Simple unification-based type inference for GADTs

Simon Peyton Jones
Microsoft Research, Cambridge

Dimitrios Vytiniotis Stephanie Weirich
Geoffrey Washburn
University of Pennsylvania

Abstract

Generalized algebraic data types (GADTs), sometimes known as "guarded recursive data types" or "first-class phantom types", are a simple but powerful generalization of the data types of Haskell and ML. Recent works have given compelling examples of the utility of GADTs, although type inference is known to be difficult. Our contribution is to show how to exploit programmer-supplied type annotations to make the type inference task almost embarrassingly easy. Our main technical innovation is *wobbly types*, which express in a declarative way the uncertainty caused by the incremental nature of typical type-inference algorithms.

1. Introduction

Generalized algebraic data types (GADTs) are a simple but potent generalization of the recursive data types that play a central role in ML and Haskell. In recent years they have appeared in the literature with a variety of names (guarded recursive data types [24], first-class phantom types [5], equality-qualified types [18], and so on), although they have a much longer history in the dependent types community. Any feature with so many names must be useful—and indeed these papers and others give many compelling examples.

It is time to pluck the fruit. We seek to turn GADTs from a specialized hobby into a mainstream programming technique, by incorporating them as a conservative extension of Haskell (a similar design would work for ML). The main challenge is integrating GADTs with *type inference*, a dominant feature of Haskell and ML.

Rather than seeking a super-sophisticated inference algorithm, an increasingly popular approach is to guide type inference using programmer-supplied type annotations. With this in mind, our central focus is this: *we seek a declarative type system for a language that includes both* GADT*s and programmer-supplied type annotations, which has the property that type inference is straightforward*. Our goal is a type system that is *predictable* enough to be used by ordinary programmers; and *simple* enough to be implemented without heroic efforts. We make the following specific contributions:

- We specify a programming language that supports GADTs and programmer-supplied type annotations (Section 4). The key innovation in the type system is the notion of a *wobbly type* (Section 3), which models the places where an inference algorithm would make a "guess". The idea is that type refinements induced by GADTs never refine wobbly types, and hence type inference is insensitive to the order in which the algorithm traverses the abstract syntax tree.
- Like any system making heavy use of type annotations, we offer support for lexically scoped type variables that can be bound by both polymorphic type signatures and signatures on patterns (Section 4.4 and 5.5). There is no rocket science here, but we think our design is particularly simple and easy to specify, certainly compared to our earlier efforts.
- We explore a number of extensions to the basic system, including improved type checking rules for patterns and case expression scrutinees, and nested patterns (Section 5).
- We prove that our type system is sound, and that it is a conservative extension of a standard Hindley-Milner type system (Section 6). Moreover our language can express all programs that an explicitly-typed language could express.
- We sketch a type inference algorithm for our type system that is a modest variant of the standard algorithm for Hindley-Milner type inference. We have proven that this algorithm is sound and complete (Section 6.3). An companion technical report gives details of the algorithm and proofs of its properties [22].

We have implemented type inference for GADTs, using wobbly types, in the Glasgow Haskell Compiler (GHC). GHC's type checker is already very large; not only does it support Haskell's type classes, but also numerous extensions, such as functional dependencies, implicit parameters, arbitrary-rank types, and more besides. An extension that required all this to be re-engineered would be a non-starter and it is here that the simplicity of our GADT inference algorithm pays off.

Our implementation has already received heavy use. To be fair, the released implementation in GHC uses a more complicated scheme than that described here, originally given in an earlier version of this paper (see Section 7). We are in the midst of re-engineering the implementation to match what we describe in this revised, simpler version.

2. Background

By way of background, we use a standard example to remind the reader of the usefulness of GADTs. Here is a declaration of a `Term` data type that represents terms in a simply-typed language:

```
data Term a where
   Lit  :: Int -> Term Int
   Inc  :: Term Int -> Term Int
   IsZ  :: Term Int -> Term Bool
   If   :: Term Bool -> Term a -> Term a -> Term a
   Pair :: Term a -> Term b -> Term (a,b)
   Fst  :: Term (a,b) -> Term a
   Snd  :: Term (a,b) -> Term b
```

Submitted to PLDI'06: November 11, 2005

`Term` has a type parameter `a` that indicates the type of the term it represents, and the declaration enumerates the constructors, giving each an explicit type signature. These type signatures only allow one to construct well-typed terms; for example, the term `(Inc (IsZ (Lit 0)))` is rejected as ill-typed, because `(IsZ (Lit 0))` has type `Term Bool` and that is incompatible with the argument type of `Inc`.

An evaluator for terms becomes stunningly direct:

```
eval :: Term a -> a
eval (Lit i)    = i
eval (Inc t)    = eval t + 1
eval (IsZ t)    = eval t == 0
eval (If b t e) = if eval b then eval t else eval e
eval (Pair a b) = (eval a, eval b)
eval (Fst t)    = fst (eval t)
eval (Snd t)    = snd (eval t)
```

It is worth studying this remarkable definition. Note that the right hand side of the first equation patently has type `Int`, not `a`. But, if the argument to `eval` is a `Lit`, then the type parameter `a` must be `Int` (for there is no way to construct a `Lit` term other than to use the typed `Lit` constructor), and so the right hand side has type `a` also. Similarly, the right hand side of the third equation has type `Bool`, but in a context in which `a` must be `Bool`. And so on.

The key ideas are these:

- A generalized data type is declared by enumerating its constructors, giving an explicit type signature for each. In conventional data types in Haskell or ML, a constructor has a type of the form $\forall \overline{a}.\overline{\tau} \rightarrow \mathsf{T}\ \overline{a}$, where the result type is the type constructor T applied to all the type parameters $\overline{a}$. In a generalized data type, the result type must still be an application of T, but the argument types are arbitrary. For example `Lit` mentions no type variables, `Pair` has a result type with structure `(a,b)`, and `Fst` mentions some, but not all, of its universally-quantified type variables.
- The data constructors are functions with ordinary polymorphic types. There is nothing special about how they are used to construct terms, apart from their unusual types.
- All the excitement lies in pattern matching. Matching against a constructor may induce a *type refinement* in the case alternative. For example, in the `Lit` branch of `eval`, we can refine `a` to `Int`.
- The dynamic semantics is unchanged. Pattern-matching is done on data constructors only, and there is no run-time type passing.

The `eval` function is a somewhat specialized example, but earlier papers have given many other applications of GADTs, including generic programming, modeling programming languages, maintaining invariants in data structures (e.g. red-black trees), expressing constraints in domain-specific embedded languages (e.g. security constraints), and modeling objects [8, 24, 5, 18, 16, 17]. The interested reader should consult these works; meanwhile, for this paper we simply take it for granted that GADTs are useful.

3. The key idea

Our goal is to combine the flexibility of Hindley-Milner type inference with the expressiveness and power of GADTs, by using the programmer's annotations to guide type inference. Furthermore, we seek a system that gives as much freedom as possible to the inference algorithm. For example, we would like to retro-fit GADT inference to existing compilers, as well as using it for new ones.

The difficulty with type inference for GADTs is well illustrated by the `eval` example of Section 2. In the absence of the type signature for `eval`, a type inference engine would have to anti-refine the `Int` result type for the first two equations, and the `Bool` result type of the third (etc.), to guess that the overall result should be of type `a`. Such a system would certainly lack principal types. Furthermore, polymorphic recursion is required: for example, the recursive call to `eval` in the second equation is at type `Int`, not `a`. All of these problems go away when the programmer supplies us with the type of `eval`. Here is another variant:

```
f x y = case x of
          Lit i -> i + y
          other -> 0
```

There are at least two types one could attribute to `f`, namely `Term` $\alpha \rightarrow$ `Int` $\rightarrow$ `Int` and `Term` $\alpha \rightarrow \alpha \rightarrow$ `Int`. The latter works because the type refinement induced by the pattern match on `x` will refine the type of `y`. Alas, neither is more general than the other. Again, a programmer-supplied type signature would solve the problem. Thus motivated, our main idea is the following:

Type refinement applies only to user-specified types.

In the case of `f`, since there are no type annotations, no type refinement will take place: `y` will get type `Int`. However, if the programmer adds a type annotation, the situation is quite different:

```
f :: Term a -> a -> Int
f x y = case x of
          Lit i -> i + y
          other -> 0
```

Now it is "obvious" that `x` has type `Term a` and `y` has type `a`. Because the scrutinee of the `case` has the user-specified type `Term a`, the `case` expression does type refinement, and in the branch the type system knows that `a` $=$ `Int`. Because the type of `y` is also user-specified, this type refinement is applied when `y` occurs in the right hand side.

To summarise, our general approach is this:

- Instead of "user-specified type", we use the briefer term *rigid type* to describe a type that is completely specified, in some direct fashion, by a programmer-supplied type annotation.
- A *wobbly type* is one that is not rigid. There is no such thing as a partly-rigid type; if a type is not rigid, it is wobbly[1].
- A variable is assigned a rigid type if it is clear, *at its binding site*, precisely what its type should be.
- A `case` expression will perform type refinement in each of its alternatives only if its scrutinee has a rigid type.
- The type of a variable occurrence is refined by the current type refinement only if the variable has a rigid type.

But exactly when is a type "completely specified by a type annotation"? After all, no type annotation decorates the binding for `x` in the definition of `f` above, nor is the `case` expression adorned with a result type, and yet we argued above that both should be rigid. Would it make any difference if we had written `case (id x) of ...`, where `id` is the identity function?

To answer these questions, we need a precise and predictable description of what it means for a type to be rigid.

4. The type system

The syntax of the source language is shown in Figure 1, and is entirely conventional. We assume that data types are declared by simply enumerating the constructors and their types (as in Section 2), and those typings are used to pre-populate the type environment Γ. The `let` binding form is recursive. Pattern matching is performed only by `case` expressions, but we will occasionally take the liberty of writing `\p.t` instead of `\x.case x of p` $\rightarrow$ `t`.

Programmer-supplied type annotations appear in the syntax of the source language in two places. First, a `let` definition may

[1] In an earlier version of this paper, types were allowed to have both rigid and wobbly components (Section 7).

—Source language syntax—

$$
\begin{array}{rrcl}
\text{Atoms} & v & ::= & x \mid C \\
\text{Terms} & t, u & ::= & v \mid \lambda x.t \mid t\ u \\
 & & \mid & \texttt{let } x = u \texttt{ in } t \\
 & & \mid & \texttt{let } x :: \sigma = u \texttt{ in } t \\
 & & \mid & \texttt{case } t \texttt{ of } \overline{p \texttt{ -> } t} \\
\text{Patterns} & p & ::= & x \mid C\ \overline{p} \mid p :: \overline{a}.\tau \\
\text{Polytypes} & \sigma & ::= & \forall \overline{a}.\tau \\
\text{Monotypes} & \tau, \upsilon & ::= & \tau \to \tau \mid a \mid T\ \overline{\tau}
\end{array}
$$

—Meta language syntax—

$$
\begin{array}{rrcl}
\text{Environments} & \Gamma, \Delta & ::= & \cdot \mid \Gamma, v :^m \sigma \mid \Gamma, a \\
\text{Modifiers} & m, n & ::= & w \mid r \\
\text{Refinements} & \theta, \psi & ::= & [\overline{a \mapsto \tau}] \\
\text{Triples} & K, L & ::= & (\overline{a},\ \Delta,\ \theta)
\end{array}
$$

—Refinement application—

$$
\begin{array}{rcl|rcl}
\theta^r(\sigma) & = & \theta(\sigma) & \theta(\cdot) & = & \cdot \\
\theta^w(\sigma) & = & \sigma & \theta(\Gamma, x :^m \sigma) & = & \theta(\Gamma), x :^m \theta^m(\sigma) \\
 & & & \theta(\Gamma, a) & = & \theta(\Gamma), a
\end{array}
$$

Figure 1: Syntax of source language and types

be annotated with a polytype, thus: `let x::σ = t`. Second, a pattern may be annotated with a monotype, as $p :: \overline{a}.\tau$. Haskell also allows an expression to be annotated with a type, thus $(e :: \sigma)$, and we will occasionally use this form; it is syntactic sugar for (`let x::σ = e in x`). We often use the term *type signature* for a programmer-supplied type annotation.

The language of types is also entirely conventional, stratified into *polytypes* σ and quantifier-free *monotypes* τ. The type environment Γ is more unusual. Each variable binding $v :^m \sigma$ is annotated with a modifier, m, which indicates whether the type is rigid (r) or wobbly (w). Furthermore, the type environment also binds lexically scoped type variables, a, as we discuss in Section 4.4.

A *type refinement*, θ, is simply a substitution mapping type variables to monotypes. (One can also represent the type refinement as a set of constraints, an alternative that we discuss in Section 4.3.) The operation $\theta(\Gamma)$ applies the type refinement θ to the context Γ, and is also defined in Figure 1. The key thing to note is that only the rigid bindings in Γ are affected.

4.1 Typing rules: overview

The typing rules for the language are syntax-directed, and are given in Figure 2. The main judgement has the conventional form $\Gamma \vdash t :^m \tau$. The unusual feature is the modifier m, which indicates whether the type τ is rigid or not. In algorithmic terms, $\Gamma \vdash t :^r \tau$ checks that t has type τ *when we know* τ *completely in advance*, whereas $\Gamma \vdash t :^w \tau$ checks that t has type τ without that assumption – τ may be partially or entirely unknown in advance.

The modifier m propagates information about type rigidity. In rule LET-R we see that a *type-annotated* `let` binding causes the right hand side u to be type-checked in a rigid context ($\ldots \vdash u :^r \tau$). (The notation $\overline{a} \sharp \Gamma$ means that the type variables $\overline{a}$ do not appear in Γ.) Furthermore, when typechecking u, the environment Γ is extended with a rigid binding for x, giving its specified, polymorphic type, thereby permitting polymorphic recursion, which is very often necessary in GADT programs (e.g. `eval` in Section 2).

Then, in rule ABS we see that to typecheck an abstraction $\backslash x.t$ we extend the environment Γ with a binding for x that is rigid if the context is rigid, and vice versa. So, for example, consider the term

```
let f::(∀a.Term a → a → a) = \x.\y.u in t
```

The body u of the abstraction will be typechecked in an environment that has rigid bindings for both `x` and `y` (as well as `f`).

$$\boxed{\Gamma \vdash t :^m \tau}$$

$$\frac{v :^n \forall \overline{a}.\tau \in \Gamma}{\Gamma \vdash v :^m [\overline{a \mapsto \upsilon}]\tau}\ \text{VAR}$$

$$\frac{\Gamma \vdash t :^w \tau_1 \to \tau_2 \quad \Gamma \vdash u :^w \tau_1}{\Gamma \vdash t\ u :^m \tau_2}\ \text{APP} \qquad \frac{\Gamma, x :^m \tau_1 \vdash t :^m \tau_2}{\Gamma \vdash \backslash x.t :^m \tau_1 \to \tau_2}\ \text{ABS}$$

$$\frac{\Gamma, x :^w \tau \vdash u :^w \tau \quad \overline{a} = ftv(\tau) - ftv(\Gamma) \quad \Gamma, x :^w \forall \overline{a}.\tau \vdash t :^m \tau}{\Gamma \vdash (\texttt{let } x = u \texttt{ in } t) :^m \tau}\ \text{LET-W}$$

$$\frac{ftv(\forall \overline{a}.\tau) \in dom(\Gamma) \quad \overline{a} \sharp \Gamma \quad \Gamma, x :^r \forall \overline{a}.\tau, \overline{a} \vdash u :^r \tau \quad \Gamma, x :^r \forall \overline{a}.\tau \vdash t :^m \tau}{\Gamma \vdash (\texttt{let } x :: \forall \overline{a}.\tau = u \texttt{ in } t) :^m \tau}\ \text{LET-R}$$

$$\frac{\Gamma \vdash u : \tau_p \upharpoonright^{m_p} \quad \Gamma \vdash \overline{p \to t} :^{\langle m_p, m_t \rangle} \tau_p \to \tau_t}{\Gamma \vdash (\texttt{case } u \texttt{ of } \overline{p \texttt{->} t}) :^{m_t} \tau_t}\ \text{CASE}$$

$$\boxed{\Gamma \vdash t : \tau \upharpoonright^m}$$

$$\frac{v :^m \tau \in \Gamma}{\Gamma \vdash v : \tau \upharpoonright^m}\ \text{SCR-VAR} \qquad \frac{\Gamma \vdash t :^w \tau}{\Gamma \vdash t : \tau \upharpoonright^w}\ \text{SCR-OTHER}$$

$$\boxed{\Gamma \vdash p \to t :^{\langle m_p, m_t \rangle} \tau_p \to \tau_t}$$

$$\frac{\begin{array}{c} C :^r \forall \overline{a}.\overline{\tau}_1 \to T\ \overline{\tau}_2 \in \Gamma \quad \overline{a} \sharp ftv(\Gamma, \overline{\tau}_p, \tau_t) \\ \overline{a}_c = \overline{a} \cap ftv(\overline{\tau}_2) \quad \theta = [\overline{a_c \mapsto \upsilon}] \quad \theta(\overline{\tau}_2) = \overline{\tau}_p \\ \Gamma, \overline{x :^w \theta(\tau_1)} \vdash t :^m \tau_t \end{array}}{\Gamma \vdash C\ \overline{x} \to t :^{\langle w, m \rangle} T\ \overline{\tau}_p \to \tau_t}\ \text{PCON-W}$$

$$\frac{\begin{array}{c} C :^r \forall \overline{a}.\overline{\tau}_1 \to T\ \overline{\tau}_2 \in \Gamma \quad \overline{a} \sharp ftv(\Gamma, \overline{\tau}_p, \tau_t) \\ \theta = mgu(\overline{\tau}_p \doteq \overline{\tau}_2) \\ \theta(\Gamma, \overline{x :^r \tau_1}) \vdash \theta(t) :^m \theta^m(\tau_t) \end{array}}{\Gamma \vdash C\ \overline{x} \to t :^{\langle r, m \rangle} T\ \overline{\tau}_p \to \tau_t}\ \text{PCON-R}$$

Figure 2: Typing rules for scrutinees and case alternatives

The APP rule always typechecks both function and argument in a wobbly context: even if the result type is entirely known, the function and arguments types are not. One might wonder whether the function might provide a rigid context for its argument, or vice versa, but APP does not attempt such sophistication (but see Section 5.1).

Rule VAR does not use the modifier n of the variable type in the environment. It merely checks that the type of the variable in the environment can be instantiated to the type given by the judgement.

The really interesting rule is, of course, that for `case`, which we discuss next. For the moment we restrict ourselves to flat patterns, of form $C\ \overline{x}$, leaving nested patterns for Section 5.4.

4.2 Pattern matching

The key idea of the paper is that a `case` expression only performs type refinement if the scrutinee has a rigid type. The auxiliary judgement $\Gamma \vdash t : \tau \upharpoonright^m$ which is defined in Figure 2, figures out whether or not the scrutinee is rigid. Rather than pushing the modifier inwards as the main judgement does, it *infers* the modifier.

The judgement is almost trivial, because it has just one interesting case, for variables. Rule SCR-VAR returns the modifier found for the variable in Γ; otherwise the judgement conservatively returns a wobbly modifier w (SCR-OTHER)[2]. We will extend this judgement later, in Section 5.1.

Rule CASE first uses this new judgement to typecheck the scrutinee u, and then typechecks each alternative of the `case`, passing in both the rigidity of the scrutinee, m_p, and the rigidity of the result type, m_t.

The `case`-alternative rules are also given in Figure 2. There are two cases to consider. Rule PCON-W is used when the scrutinee has a wobbly type. In that case we need ordinary Hindley-Milner type checking. We look up the constructor C in the type environment, α-rename its quantified type variables to avoid ones that are in use, and then find a substitution θ that makes the result type of the constructor $T\ \overline{\tau}_2$ match the type of the pattern $T\ \overline{\tau}_p$. Finally, we extend Γ with wobbly bindings for the variables $\overline{x}$ (obtained by instantiating the constructor's type), and type check the right hand side of the alternative, t.

One subtle point is that the constructor may bind *existential* type variables. For example, suppose $\texttt{MkT} :: \forall a\,b.a \to (a \to b) \to T\ b$. Then the type variable a is existentially bound by a pattern for `MkT`, because a does not appear in the result type $T\ b$. Clearly we must not substitute for a; for example, this term is ill-typed, if x : `MkT Bool`:

```
case x of { MkT r s -> r+1 }
```

We must form the substitution $[b \mapsto \texttt{Bool}]$ to make the result type of the constructor match that of `x`, but a is simply a fresh skolem constant. That is why PCON-W first computes $\overline{a}_c$, the subset of C's quantified variables that appear in its result type, and permits only these variables in the domain of θ.

Rule PCON-W gives a wobbly type to *all* the bound variables $\overline{x}$, which is safe but pessimistic; for example above, `r` could surely have a rigid type. We return to this question in Section 5.3.

4.3 Type refinement

Now we consider rule PCON-R, which is used when the scrutinee of the `case` has a rigid type. In that case we compute a type refinement: we find a substitution θ that *unifies* the type of the pattern, $T\ \overline{\tau}_p$, and the result type of the constructor, $T\ \overline{\tau}_2$. Unlike rule PCON-W, θ can bind type variables free in the type of the pattern, $T\ \overline{\tau}_p$, as well as the quantified type variables of the constructor. Now we apply the type refinement to the environment, to the term itself, and to the result type, before type-checking the right hand side of the branch. Of course, when applying the refinement to the environment, we only refine rigid bindings (see Figure 1), and similarly we only refine the result type τ_t if it is rigid (hence $\theta^m(\tau_t)$). We always apply the refinement to the term because its type annotations (certainly rigid!) may mention a type variable that is in $\mathrm{dom}(\theta)$.

If unification fails to compute a type refinement, then the `case` alternative cannot possibly match, and the type system rejects the program. Another possible design choice is to accept statically-inaccessible alternatives without even type-checking the right hand side (since it can never be reached) but that takes an extra rule or two, and it seems quite acceptable to reject a program that contains a completely useless alternative.

The appearance of unification, mgu, is unusual for a declarative type system, although not without precedent [9]. The best way to think about it is this: a successful pattern match implies the truth of an equality constraint of form $T\ \overline{\tau_p} \doteq T\ \overline{\tau_t}$, and the case alternative should be checked under that constraint. We express this idea by solving the constraint to get its most general unifier, and applying the unifier to the entire judgement (modulo the rigid/wobbly distinctions).

Most other authors choose to deal with the constraint sets explicitly, using a judgement of form $C, \Gamma \vdash t : \tau$, where C is a set of constraints, and type equality is taken modulo these constraints [24, 5, 19, 15]. That approach is more general, but it is less convenient in our context, because by the time that type equality is invoked, the provenance of the types (and in particular whether or not they are rigid) has been lost. For example, we do not want this judgement to hold:

$$a \doteq \texttt{Int}, \texttt{xs}:^w \texttt{[a]} \vdash (\texttt{3:xs}) : \texttt{[Int]}$$

It should not hold because `xs` has a wobbly type. But the type equality arises from instantiating the call to cons `(:)`, and by that time the fact that its second argument had a wobbly type has been lost. A solution would be to embody wobbliness in the types themselves, as an earlier version of this paper did, but the approach we give here is significantly simpler.

Would *any* unifier do, other than the most general one? No, it would not. We must not make any substitution beyond those justified by the constraints. For example, consider the program

```
f::∀a.(a,b) -> Int
  = \x. case x of { (p,q) -> p+1 }
```

It would obviously be utterly wrong to substitute `Int` for `a` in the case alternative! Nor, just as in rule PCON-W, can we refine the types of existential variables.

There is also a dual question: must θ be a unifier at all? The answer here is more nuanced: "no" for soundness, but "yes" for completeness. To avoid getting bogged down in detail here, we will discuss this point in Section 6.4.

4.4 Lexically scoped type variables

Any polymorphic language that exploits user type annotations, as we do here, must support lexically scoped type variables, so that a type signature can mention type variables that are bound "further out". This feature is curiously absent from Haskell 98, and its absence is often awkward. For example:

```
prefix :: a -> [[a]] -> [[a]]
prefix x yss = map xcons yss
  where
    xcons :: [a] -> [a]
    xcons ys = x : ys
```

This program is rejected by Haskell, because the type signature for `xcons` is implicitly quantified to mean $\forall a.[a] \to [a]$. What we want here is an *open* type signature for `xcons` that mentions a type variable bound by the definition of `prefix`.

In our small language, we therefore allow the programmer to annotate a `let` definition with a polymorphic type, $\forall\overline{a}.\tau$. The quantified type variables $\overline{a}$ scope over *the entire right hand side*, as well as over the type τ. Any type variables that are free in the type annotation $\forall\overline{a}.\tau$ must be already in scope. So we could write the `prefix` example like this:

```
prefix::∀a.[[a]] → [[a]]
  = \x.\yss. let xcons :: [a] → [a]
                 = \ys. x:ys
             in map xcons yss
```

The type variables that are lexically in scope are bound by the environment Γ (see the syntax in Figure 1); in a full-blown system, the environment would also record their kinds. The environment is extended in rule LET-R, to bring new lexically-scoped type variables into scope. The same rule insists that (a) the user-supplied annotation is well-formed ($\mathrm{ftv}(\forall\overline{a}.\tau) \in \Gamma$), and (b) the lexically-scoped type

[2] To be truly syntax-directed, SCR-OTHER would need a side condition to exclude the variable case.

variables do not shadow ($\overline{a} \sharp \Gamma$). The latter restriction is easily lifted by slightly complicating the rules.

The idea that the quantified type variables of a type signature should scope over the right hand side of its definition is not new: it is used in Mondrian [12] and Chameleon [20]. It seems a little peculiar, and we resisted it for a long time, but it is extremely direct and convenient, and we now regard it as the Right Thing.

The job is not done, though. We still need a way to name *existentially-bound* type variables. For example, consider this (slightly contrived) example:

```
data T where MkT :: [a] -> (a->Int) -> T
f::T -> Int
  = \x. case x of
          MkT ys g -> let y::?? = head ys
                      in g y
```

What type can we attribute to `y` in the inner `let` binding? We need a name for the existential type variable that is bound by the pattern `(MkT ys g)`. We achieve this by allowing the programmer to annotate any pattern with a monotype. For example:

```
f::T -> Int
  = \x. case x of
          MkT (ys::a.[a]) g -> let y::a = head ys
                               in g y
```

The pattern `(ys::a.[a])` brings the type variable `a` into scope, so that it can be used in the `let` binding for `y`. In general, a type-annotated pattern $(p :: \overline{a}.\tau)$ brings into scope the type variables $\overline{a}$, which then scope over τ, p, all patterns to the right of the binding site, and the right hand side of the case alternative. The typing rules of Figure 2 only deal with simple flat patterns; we will formalize type-annotated patterns when we discuss nested patterns in Section 5.4.

In a real programming language such as Haskell, quantification is often implicit. For example, the "$\forall\overline{a}$" quantification in `let` binding might be determined by taking $\overline{a}$ to the type variables that are mentioned in the type, but are not already in scope. (Indeed, we adopt this convention for many of types we write in this paper.) A similar convention could be used for the the "$\overline{a}$." quantification in a pattern type signature. However, for our formal material we assume that quantification is explicit.

4.5 Type inference

It is very straightforward to perform type inference for our system. One algorithm that we have worked out in detail is based on the standard approach to inference for Hindley-Milner systems [4, 13]. The inference engine maintains an ever-growing substitution mapping meta type variables to monotypes. Whenever the inference engine needs to guess a type (for example in rule ABS) it allocates a fresh meta type variable; and whenever it must equate two types (such as rule APP) it unifies the types and extends the substitution.

Modifying this algorithm to support GADTs is simple. Bindings in the type environment Γ carry a boolean rigid/wobbly flag, as does the result type. The implementation of pattern-matching can be read directly from rules PCON-R and PCON-W. In fact the only difference between our actual implementation and the one suggested by the rules is that we apply the type refinement to Γ lazily: we pass down the current type refinement and apply it in the VAR rule.

There is one subtlety, which lies in the implementation of $\mathtt{mgu}$, in PCON-R. Consider the possible type derivations for

$$\mathtt{x} :^{\mathrm{r}} (a, b) \vdash (\mathtt{case\ x\ of\ (p,q)\ ->\ p}) :^{w} a$$

The pair constructor has type $\forall c\,d.\,c \to d \to (c, d)$. So the unification problem in PCON-R will be $\mathtt{mgu}(a \doteq c, b \doteq d)$. There are several most general unifiers of this constraint, and not all of them are useful to us. For example, while the substitution $[a \mapsto c, b \mapsto d]$ is a most general unifier, it will not type this program, because the type of `p` will be `c` which does not match result type a. Instead, *we want the unifier to eliminate the freshly-introduced type variables*, in this case `c` and `d`.

Our inference engine therefore uses a "biased" $\mathtt{mgu}$ algorithm that preferentially eliminates freshly-introduced type variables. We have proven that using this biased $\mathtt{mgu}$ in PCON-R does not change what programs are typeable. Therefore the implementation still remains complete with respect to the specification (Section 6.3).

5. Variations on the theme

The type system we have described embodies a number of somewhat *ad hoc* design choices, which aim to balance expressiveness with predictability and ease of type inference. In this section we explore the design space a bit further, explaining several variations on the basic design that we have found useful in practice.

5.1 Smart application

The rules we have presented will type many programs, but there are still some unexpected failures. Here is an example (c.f. [3])[3]:

```
data Equal a b where
  Eq :: Equal a a
data Rep a where
  RI :: Rep Int
  RP :: Rep a -> Rep b -> Rep (a,b)

test :: Rep a -> Rep b -> Maybe (Equal a b)
test RI RI = Just Eq
test (RP s1 t1) (RP s2 t2)
  = case (test s1 s2) of
      Nothing -> Nothing
      Just Eq -> case (test t1 t2) of
                   Nothing -> Nothing
                   Just Eq -> Eq
```

A non-bottom value `Eq` of type `Equal a b` is a witness that the types `a` and `b` are the same; that is why the constructor has type $\forall a.$`Equal a a`. Consider the outer `case` expression in `test`. The programmer reasons that since the types of `s1` and `s2` are rigid, *then so is the type of* `(test s1 s2)`, and hence the `case` should perform type refinement; and indeed, `test` will only pass the type checker if both its `case` expressions perform type refinement.

The difficulty is that the scrutinee-typing rules of Figure 2 conservatively assume that an application has a wobbly type, so neither `case` expression will perform type refinement. We could solve the problem by adding type annotations, but that is clumsy:

```
test :: Rep a -> Rep b -> Maybe (Equal a b)
test RI RI = Just Eq
test (RP (s1::a1.Rep a1) (t1::b1.Rep b1))
     (RP (s2::a2.Rep a2) (t2::b2.Rep b2))
  = let r1 :: Eq a1 a2 = test a1 a2
        r2 :: Eq b1 b2 = test b1 b2
    in case r1 of
      Nothing -> Nothing
      Just Eq -> case r2 of
                   Nothing -> Nothing
                   Just Eq -> Eq
```

(However, note the importance of pattern-binding the type variables `a1`, `a2` etc, so that they can be used to attribute a type to `s1`, `t1` etc.) To avoid this clumsiness, we need a way to encode the programmer's intuition that if `test`'s argument types are rigid, then so is its result type. More precisely, if all of the type variables in `test`'s *result* appear in an *argument* type that is rigid, then the result type should be rigid. Here is the rule, which extends the scrutinee

[3] We take the liberty of using pattern matching on the left-hand side and separate type signatures, but they are just syntactic sugar.

typing rules of Figure 2:

$$\frac{\begin{array}{c} \nu :^{r} \forall \overline{a}.\overline{\tau} \to \tau_r \in \Gamma \qquad \Gamma \vdash u_i : [\overline{a \mapsto v}]\tau_i \restriction^{m_i} \\ \overline{a}_r = \{a \in \overline{a} \mid \exists i. a \in ftv(\tau_i) \wedge m_i = r\} \\ m = \begin{cases} r & \text{if } ftv(\tau_r) \subseteq \overline{a}_r \\ w & \text{otherwise} \end{cases} \end{array}}{\Gamma \vdash \nu\ \overline{u} : [\overline{a \mapsto \tau_a}]\tau_r \restriction^{m}} \text{ SCR-APP}$$

The rule gives special treatment to applications $\nu\ \overline{u}$ of an atom ν to zero or more arguments $\overline{u}$, where ν has a rigid type in Γ. In that case, SCR-APP recursively uses the scrutinee typing judgement to infer the rigidity m_i of each argument u_i. Then it computes the subset $\overline{a}_r$ of ν's quantified type variables that appear in at least one rigid argument. We can deduce (rigidly) how these variables should be instantiated. Hence, if all the type variables free in the result of ν are in a_r then the result type of the call is also known rigidly.

One could easily imagine adding further scrutinee-typing rules. Notably, if the language supported type annotations on terms, $(t::\sigma)$, then one would definitely also want to add a scrutinee-typing rule to exploit such annotations:

$$\frac{\Gamma \vdash t :^{r} \tau}{\Gamma \vdash (t::\tau) : \tau \restriction^{r}} \text{ SCR-SIG}$$

Now, in any place where a `case` expression has a wobbly scrutinee, the programmer can make it rigid by adding an annotation, thus: (`case t::τ of ...`). Beyond that, we believe that there is little to be gained by adding further rules to the scrutinee-typing rules.

5.2 Smart let

Consider these two terms, where `(f x)` is determined to be rigid by SCR-APP:

```
case f x of                 let s = f x
  MkT a b -> ..             in case s of
                                 MkT a b -> ...
```

With the rules so far, the left-hand `case` would do refinement, but the right hand `case` would not, because `s` would get a wobbly type. This is easily fixed by re-using the scrutinee judgement for the right hand side of a `let`:

$$\frac{\begin{array}{c} \Gamma, x :^{w} \tau \vdash u : \tau \restriction^{n} \\ \overline{a} = ftv(\tau) - ftv(\Gamma) \qquad \Gamma, x :^{n} \forall \overline{a}.\tau \vdash t :^{m} \tau \end{array}}{\Gamma \vdash (\texttt{let } x = u \texttt{ in } t) :^{m} \tau} \text{ LET-W}$$

This change means that introducing a `let` does not gratuitously lose rigidity information. An interesting property is that if LET-W infers a rigid type for x, then x is monomorphic and $\overline{a}$ is empty:

THEOREM 5.1. *If* $\Gamma \vdash u : \tau \restriction^{r}$ *then* $ftv(\tau) \subseteq ftv(\Gamma)$.

Why is this true? Because the only way u could get a rigid type is by extracting it from Γ.

5.3 Smart patterns

Consider rule PCON-W in Figure 2, used when the scrutinee has a wobbly type. It gives a wobbly type to *all* the variables $\overline{x}$ bound by the pattern. However, if some of the fields of the constructor have purely existential types, then these types are definitely rigid, and it is over-conservative to say they are wobbly.

This observation motivates the following variant of PCON-W

$$\frac{\begin{array}{c} C :^{r} \forall \overline{a}.\overline{\tau}_1 \to T\ \overline{\tau}_2 \in \Gamma \quad \overline{a} \sharp ftv(\Gamma, \overline{\tau}_p, \tau_t) \\ \overline{a}_c = \overline{a} \cap ftv(\overline{\tau}_2) \quad \theta = [\overline{a_c \mapsto v}] \quad \theta(\overline{\tau}_2) = \overline{\tau}_p \\ m_i = \begin{cases} r & \text{if } ftv(\tau_{1i}) \sharp \overline{a}_c \\ w & \text{otherwise} \end{cases} \\ \Gamma, \overline{x :^{m_i} \theta(\tau_{1i})} \vdash t :^{m} \tau_t \end{array}}{\Gamma \vdash C\ \overline{x} :^{\langle w, m \rangle} T\ \overline{\tau}_p \to \tau_t} \text{ PCON-W}$$

$$\boxed{\Gamma \vdash p \to t :^{\langle m_p, m_t \rangle} \tau_p \to \tau_t}$$

$$\frac{\Gamma, (\emptyset,\ \cdot,\ \emptyset) \vdash p :^{m_p} \tau_p \blacktriangleright (\overline{a},\ \Delta,\ \theta) \qquad ftv(\Gamma, \tau_p, \tau_t) \sharp \overline{a} \quad \theta(\Gamma \cup \Delta) \vdash \theta(t) :^{m_t} \theta^{m_t}(\tau_t)}{\Gamma \vdash p \to t :^{\langle m_p, m_t \rangle} \tau_p \to \tau_t} \text{ PAT}$$

$$\boxed{\Gamma, K_1 \vdash p :^{m} \tau \blacktriangleright K_2}$$

$$\frac{x \notin dom(\Delta)}{\Gamma, (\overline{a},\ \Delta,\ \theta) \vdash x :^{m} \tau \blacktriangleright (\overline{a},\ \Delta, x :^{m} \tau,\ \theta)} \text{ PVAR}$$

$$\frac{\begin{array}{c} C :^{r} \forall \overline{b}.\overline{\tau}_1 \to T\ \overline{\tau}_2 \in \Gamma \quad \overline{b} \sharp \overline{a} \\ \overline{b}_c = \overline{b} \cap ftv(\overline{\tau}_2) \quad \psi = [\overline{b_c \mapsto v}] \quad \overline{\tau}_3 = \psi(\overline{\tau}_2) \\ m_i = \begin{cases} r & ftv(\tau_{1i}) \sharp \overline{b}_c \\ w & \text{otherwise} \end{cases} \\ \Gamma, (\overline{a}\overline{b},\ \Delta,\ \theta) \vdash^{fold} \overline{p_i :^{m_i} \psi(\tau_{1i})} \blacktriangleright K \end{array}}{\Gamma, (\overline{a},\ \Delta,\ \theta) \vdash C\ \overline{p} :^{w} T\ \overline{\tau}_3 \blacktriangleright K} \text{ PCON-W}$$

$$\frac{\begin{array}{c} C :^{r} \forall \overline{b}.\overline{\tau}_1 \to T\ \overline{\tau}_2 \in \Gamma \quad \overline{b} \sharp \overline{a} \\ \theta(\tau) = T\ \overline{\tau}_3 \quad \psi = mgu(\overline{\tau}_3 \doteq \overline{\tau}_2) \\ \Gamma, (\overline{a}\overline{b},\ \Delta,\ \psi \cdot \theta) \vdash^{fold} \overline{p_i :^{r} \tau_{1i}} \blacktriangleright K \end{array}}{\Gamma, (\overline{a},\ \Delta,\ \theta) \vdash C\ \overline{p} :^{r} \tau \blacktriangleright K} \text{ PCON-R}$$

$$\frac{\begin{array}{c} \overline{b} \sharp dom(\Gamma \cup \Delta) \\ m = w \Rightarrow \overline{b} = \{\} \quad \theta^{m}(\tau_s) = \theta^{m}(\tau) \\ \Gamma, (\overline{a},\ (\Delta, \overline{b}),\ \theta) \vdash p :^{m} \tau \blacktriangleright K \end{array}}{\Gamma, (\overline{a},\ \Delta,\ \theta) \vdash (p::\overline{b}.\tau_s) :^{m} \tau \blacktriangleright K} \text{ PANN}$$

$$\boxed{\Gamma, K_1 \vdash^{fold} \overline{p_i :^{m_i} \tau_i} \blacktriangleright K_2}$$

$$\frac{}{\Gamma, K \vdash^{fold} \cdot \blacktriangleright K} \text{ F-BASE}$$

$$\frac{\Gamma, K_1 \vdash p :^{m} \tau \blacktriangleright K_2 \qquad \Gamma, K_2 \vdash^{fold} \overline{p_i :^{m_i} \tau_i} \blacktriangleright K_3}{\Gamma, K_1 \vdash^{fold} (p :^{m} \tau), \overline{p_i :^{m_i} \tau_i} \blacktriangleright K_3} \text{ F-REC}$$

Figure 3: Source language pattern typing

Here we attribute a rigid type to x_i if x_i's type does not mention any of the type variables a_c that are contaminated by appearing in the result type of the constructor; that is, x_i is rigid if its type is purely existential.

To be honest, this elaboration of PCON-W is motivated more by the fact that it is easy to describe and implement, and its symmetry with SCR-APP, rather than because we know of useful programs that would require more annotation without it.

5.4 Nested patterns

In Section 4 we treated only flat patterns, and we did not handle pattern type signatures (introduced in Section 4.4). Handling nested patterns introduces no new technical or conceptual difficulties, but the rules look substantially more intimidating, which is why we have left them until now. The rules for nested patterns are given in Figure 3. The main new judgement typechecks a nested pattern, p:

$$\Gamma, K_1 \vdash p :^{m} \tau \blacktriangleright K_2$$

Here K is a triple $(\overline{a}, \Delta, \theta)$, with three components (Figure 1):

- $\overline{a}$ is the set of type variables bound by the pattern. We need to collect these variables so that we are sure to choose unused variables when instantiating a constructor, and so that we can ensure that none of the existential variables escape.
- Δ gives the typings of term variables bound by the pattern, and the lexically-scoped type variables brought into scope by pattern type signatures; we use Δ to extend Γ before typing the body of the case alternative.
- θ is the type refinement induced by the pattern.

This triple K is threaded through the judgement: K_1 gives the bindings from patterns to the left of p, and K_2 is the result of augmenting K_1 with the bindings from p.

With that in mind, rule PAT is easy to read (compare it with PCON-R from Figure 2): it invokes the pattern-checking judgement, starting with an empty K, checks that none of the existential type variables escape, and typechecks the body t of the case alternative after extending the type environment with Δ and applying the type refinement θ. Rule PVAR is also straightforward; the test $x \notin dom(\Delta)$ prevents a single variable being used more than once in a single pattern match.

The constructor rules PCON-W and PCON-R are similar to those in Figure 2, with the following differences. First, the sub-patterns are checked using an auxiliary judgement $\vdash^{fold}$, which simply threads the K triple through a vector of patterns. Second, in PCON-R the incoming substitution θ is *composed* with the unifier, ψ,f to obtain $(\psi \cdot \theta)$. In PCON-W, however, the instantiation ψ has only the fresh variables b_c in its domain, so there is no need to extend the global type refinement θ.

There is one tricky point. Consider the following example:

```
data T where C :: Rep a -> a -> T
data Rep a where  RI :: Rep Int
                 RB :: Rep Bool

f :: T -> Bool
f (C RB True) = False
f (C RI 0)    = False
f other       = True
```

Should this program typecheck? The constructor `C` binds an existential variable `a`. The pattern `RB` induces a type refinement that refines `a` to `Bool`; and hence, in our system, the pattern `True` typechecks, and the program is accepted. There is a left-to-right order implied here, and our system would reject the definition if the order of arguments to `C` were reversed. Furthermore, accepting the program requires that the operational order of pattern matching must also be left-to-right. In a lazy language like Haskell, termination considerations force this order anyhow, so no new compilation constraints are added by our decision. In a strict language, however, one might argue for greater freedom for the compiler, and hence less type refinement.

This left-to-right ordering shows up in the way that the type refinement is threaded through the sub-patterns of a constructor by $\vdash^{fold}$. It also requires one subtlety in PCON-R. Notice that the conclusion of PCON-R does not say $C\ \overline{p} : T\ \overline{\tau}_3$, as in PCON-W; instead, the conclusion says simply $C\ \overline{p} : \tau$, with $\theta(\tau) = T\ \overline{\tau}_3$ as a premise. The reason is apparent from the above example. When typechecking the pattern `True`, we must establish the judgement

$$\Gamma, (a, \cdot, [a \mapsto \texttt{Bool}]) \vdash \texttt{True} : a \blacktriangleright (a, \cdot, [a \mapsto \texttt{Bool}])$$

That is, we must check that the pattern `True` has type `a` (not `Bool`). Hence the need to apply the current substitution (coming from patterns to the left) before requiring the pattern type to be of the form $T\ \overline{\tau}_3$.

5.5 Pattern type signatures

Figure 3 also enhances the type checking of patterns to accommodate pattern type signatures, which we introduced informally in Section 4.4. Rule PANN does the job, in the following stages:

- First we check that the new lexically-scoped type variables $\overline{b}$ are not already in scope.
- Next, the premise $m = w \Rightarrow \overline{b} = \{\}$ ensures that the signature binds new type variables only if the pattern type τ is rigid. This is necessary to ensure that each lexically-scoped type variable is rigid. For example, this definition is rejected:

  ```
  f = \(x::a.a) -> x
  ```

  Why? Because the pattern has a wobbly type, so the scoped type variable `a` would be wobbly.
- Next, we check that the type τ of the pattern and the signature τ_s are identical when the current type refinement is applied. We may need to α-rename $\overline{a}.\tau_s$ to make τ_s match τ precisely.
- Lastly, we extend the type environment Δ with the new type variables, and typecheck p.

Many other design choices could be made here without fundamentally changing our system. We insist that a lexically-scoped type variable gives a name to a (rigid) type *variable*, and that distinct lexical type variables name distinct type variables. One could instead say that a lexical type variable names an arbitrary *type*. That could readily be achieved by requiring only that τ_s *matches* τ via a substitution for the $\overline{b}$, rather than requiring equality. The environment Γ could record the binding between each lexical type variable and the type it names.

At present, scoped type variables are always rigid, so that any programmer-written type annotation specifies a rigid type. One could instead allow lexical type variables to name *wobbly* types, rather than just rigid ones, and refine them selectively just as we do term variable bindings. The type system remains tractable, but becomes noticeably more complicated, because we must now infer the rigidity of both scoped type variables (or, rather, of the types they stand for), and of type annotations.

The choice among these designs is a matter of taste. We present here the design that we have found to be simplest to specify and describe.

6. Properties of our system

We have proven that our system enjoys the usual desirable properties: it is sound (Section 6.1); it can express anything that an explicitly-typed language can (Section 6.2); we have a sound and complete type inference algorithm (Section 6.3); and it is a conservative extension of the standard Hindley-Milner type system (Section 6.6). Although these properties are standard, they are easily lost, as we elaborate in this section. All of the results in this section hold for the most elaborate version of the rules we have presented, including all of the extensions in Section 5. We do not present the proofs of these results here, but refer interested readers to the companion technical report [22][4].

6.1 Soundness

We prove soundness by augmenting our typing rules with a type-directed translation to the predicative fragment of System F augmented with GADTs. As usual, type abstractions and applications are explicit, and every binder is annotated with its type. In addition, in

[4] The development in the technical report is for a slightly different and somewhat more complicated system than the one defined here, but there are no deep differences between the two systems. Furthermore, we are working on an updated report that includes the simplifications of this paper.

support of GADTs, we annotate each `case` expression with its result type. In the accompanying technical report we give this intermediate language in full detail including its operational semantics, and show that the type system is sound.

We augment each source-language typing judgement with a translation into the target language; for example the main term judgement becomes $\Gamma \vdash t :^m \tau \rightsquigarrow t'$, where t' is the translation of t. For example, here is the VAR rule, whose translation makes explicit the type application that is implicit in the source language:

$$\frac{v :^n \forall \overline{a}.\tau \in \Gamma}{\Gamma \vdash v :^m [\overline{a \mapsto v}]\tau \rightsquigarrow v\ \overline{v}}\ \text{VAR}$$

The semantics of the source language is defined by this translation. The soundness theorem then states that if a program is well-typed in our system then its translation is well-typed in our extended System F, and hence its execution cannot "go wrong".

THEOREM 6.1 (Type safety). *If* $\vdash t :^m \tau \rightsquigarrow t'$ *then* $\vdash^F t' : \tau$.

6.2 Expressiveness

The basic idea of guiding inference using programmer-supplied type annotations suggests the following question: are the annotations expressive enough that *any* program that can be expressed by the explicitly-typed System-F-style intermediate language can also be expressed in the source language?

Yes, it can. We can give a systematic translation from the intermediate language into the source language, such that any typeable intermediate-language program translates to a typeable source-language program. The translation is straightforward: type applications are merely erased, type abstractions are replaced with annotations that bring into scope the abstraction's quantified type variables, every binder is annotated with a signature, and annotations are added to every `case` expression.

6.3 Soundness and completeness of inference

We have developed a sound and complete type inference algorithm for our system, as sketched in Section 4.5. Again, the accompanying technical report presents the algorithm in full detail, and we content ourselves with a short sketch here.

The algorithm uses notation α, β for unification variables. Unifiers, that is, idempotent substitutions from unification variables to monotypes, are denoted with δ. The algorithm also makes use of infinite sets of fresh names, which we denote with $\mathcal{A}$, and call *symbol supplies*. The main inference algorithm can be presented as a deterministic relation:

$$(\delta_0, \mathcal{A}_0) \succ \Gamma \vdash t :^m \tau \succ (\delta_1, \mathcal{A}_1)$$

The judgement should be read as: "given an initial unifier δ_0 and an initial symbol supply $\mathcal{A}_0$, check that t has the type τ with the modifier m under Γ, returning an extended unifier δ_1 and the rest of the symbol supply $\mathcal{A}_1$". Everything is an input except δ_1 and $\mathcal{A}_1$ which are results. A precondition of the algorithm is that whenever $m = r$ then $\nexists \alpha \in \tau$; that is, τ is fully known. This way we enforce a clean separation between refinement and unification. For example, consider the algorithmic rule for application:

$$\frac{(\mathcal{A}_0, \delta_0) \succ \Gamma \vdash t :^w \beta \to \tau_2 \succ (\mathcal{A}_1, \delta_1) \quad (\mathcal{A}_1, \delta_1) \succ \Gamma \vdash u :^w \beta \succ (\mathcal{A}_2, \delta_2)}{(\mathcal{A}_0 \beta, \delta_0) \succ \Gamma \vdash t\ u :^m \tau_2 \succ (\mathcal{A}_2, \delta_2)}\ \text{AAPP}$$

The function and the argument types contain the unification variable β and therefore should be checked with the wobbly modifier.

The algorithm is *sound*; that is, if a term is shown to be well-typed by the algorithm, there should exist a typing derivation in the specification that witnesses this fact.

THEOREM 6.2 (Type inference soundness). *Let* $\mathcal{A}_0$ *be a supply of fresh symbols. If* $(\mathcal{A}_0, \cdot) \succ \vdash t :^w \alpha \succ (\mathcal{A}_1, \delta)$ *then* $\vdash t :^w \delta(\alpha)$. *If* $(\mathcal{A}_0, \cdot) \succ \vdash t :^r \tau \succ (\mathcal{A}_1, \delta)$ *and* τ *does not contain unification variables, then* $\vdash t :^r \tau$.

Since unification variables live only in wobbly parts of a judgement, and unification is incremental, Theorem 6.2 relies on the following substitution property.

LEMMA 6.3 (Substitution). *If* $\mathrm{dom}(\phi)$ *is disjoint from the variables appearing in the rigid parts of the judgement* $\Gamma \vdash t :^m \tau$, *and the scoped variables of* Γ, *then* $\phi[\Gamma] \vdash t :^m \phi(\tau)$, *where* $\phi[\Gamma]$ *means the application of* ϕ *in both rigid and wobbly parts of* Γ.

The other important property of the algorithm is *completeness*; that is for all the possible types that the type system can attribute to a term, the algorithm can infer (i.e. check against a fresh unification variable) one such that all others are instances of that type.

THEOREM 6.4 (Type inference completeness). *Let* $\mathcal{A}_0$ *be a supply of fresh symbols. If* $\vdash t :^r \tau$ *then* $(\mathcal{A}_0, \cdot) \succ \vdash t :^r \tau \succ (\mathcal{A}_1, \delta)$. *If* $\vdash t :^w \tau$, *and* α *is a fresh unification variable then* $(\mathcal{A}_0, \cdot) \succ \vdash t :^w \alpha \succ (\mathcal{A}_1, \delta)$ *and* $\exists \delta_r$ *such that* $\delta_r \delta(\alpha) = \tau$.

Soundness and completeness, along with determinacy of the algorithm give us a principal types property.

THEOREM 6.5 (Principal types). *If* $\vdash t :^w \tau$ *then there exists a principal type* τ_p *such that* $\vdash t :^w \tau_p$, *and for every* τ_1 *such that* $\vdash t :^w \tau_1$ *it is the case that* $\tau_1 = \delta(\tau_p)$ *for some substitution* δ.

A principal types property for the rigid judgement is less interesting, as rigid types are always known.

6.4 Pre-unifiers and completeness

We remarked in Section 4.3 that in PCON-R it would be unsound to use any unifier other than a most-general one. But must the refinement be a unifier at all? For example, even though the `case` expression *could* do refinement, no refinement is *necessary* to typecheck this function:

```
f :: Term a -> Int
f (Lit i) = i
f other   = 0
```

That motivates the following definition:

DEFINITION 6.6 (Pre-unifier). *A substitution* θ *is a pre-unifier of types* τ_1 *and* τ_2 *iff for every unifier* ψ *of* τ_1 *and* τ_2, *there exists a substitution* θ' *s.t.* $\psi = \theta' \cdot \theta$.

That is, a pre-unifier is a substitution that can be extended to be any unifier. A most-general unifier is precisely characterized by being both (a) a unifier and (b) a pre-unifier. However, a pre-unifier need not itself be a unifier; for example, the empty substitution is a pre-unifier of any two types.

In our explicitly-typed System F language (Section 6.1), it is sound for rule PCON-R[5] to use any pre-unifier, rather than using a most-general unifier. To our surprise, however, leaving the same flexibility in the source language would preclude a complete type inference algorithm. To see why consider this program:

```
data T a where C :: T Int

g :: T a -> a -> a
g x y = let v = case x of { C -> y }
        in v
```

[5] In the explicitly-typed language everything is rigid, so there is nothing corresponding to PCON-W.

One would expect that this program would be ill-typed: `v` would get `Int`, due to the refinement of `y`'s type inside the `case` expression, and the type `Int` does not match the return type `a` of `g`.

But suppose that the specification was allowed to choose the *empty* pre-unifier for the `case` expression (thereby performing no refinement). Then `v` would get the type `a`, and the definition of `g` would typecheck! There would be nothing unsound about doing this, but it is difficult to design a type inference algorithm that will succeed on the above program. In short, completeness of type inference becomes much harder to achieve.

This was a surprise to us. Our initial system used a pre-unifier instead of a most-general unifier in PCON-R, on the grounds that $\mathtt{mgu}$ over-specifies the system, and we discovered the above example only through attempting a (failed) completeness proof for our inference algorithm. The same phenomenon as been encountered by others, albeit in a very different guise [15, section 5.3,6]. Our solution is simple: we use $\mathtt{mgu}$ in the specification of type refinement, as well as in the implementation.

6.5 Wobbliness and completeness

Our initial intuition was that if a term typechecks in a wobbly context then, *a fortiori*, it would typecheck in a rigid context. But not so; here is a counter-example. Suppose we have

```
data T a where C :: T Int
```

Then the following judgement holds:

$$\mathtt{x}:^{\mathrm{r}} \mathtt{T}\ \mathtt{a}, \mathtt{y}:^{\mathcal{W}} \mathtt{a} \vdash \mathtt{case\ x\ of\ \{\ C\ ->\ y\ \}} :^{\mathcal{W}} \mathtt{a}$$

However, if we made the binding for `y` rigid, then the type of `y` would be refined to `Int`, and the judgement would not hold any more. (It can be made to hold again by making the return type rigid as well.) This implies that there may be some programs that become untypeable when (correct!) type annotations are added, which is clearly undesirable. Again this unexpected behavior is not unique to our system [15, section 5.3], and we believe that the examples that demonstrate this situation are rather contrived.

What this means is that our specification must be careful to specify *exactly* when a type is wobbly and when it is rigid. We cannot leave any freedom in the specification about which types are rigid and which are wobbly. If we did, then again inference would become much harder and, by the same token, it would be harder for the programmer to predict whether the program would typecheck.

Since our system is (with one small exception) deterministic, it already has the required precision. The exception is rule SCR-OTHER in Figure 2, which overlaps with SCR-VAR. This is easily fixed by adding to SCR-OTHER a side condition that $\mathtt{t}$ is not an atom ν.

6.6 Relationship to Hindley-Milner

Our type system is a conservative extension of the conventional Hindley-Milner type system.

THEOREM 6.7 (Conservative extension of HM). *Assume that the algebraic datatypes are conventional. If* $\Gamma \vdash \mathtt{t} :^{\mathrm{m}} \tau$ *then* $\Gamma \vdash^{\mathrm{HM}} \mathtt{t} : \tau$. *Conversely, if* $\Gamma \vdash^{\mathrm{HM}} \mathtt{t} : \tau$ *then* $\Gamma \vdash \mathtt{t} :^{\mathrm{m}} \tau$ *for any* m.

To prove this theorem, first, we use the version of PCON-R that uses a biased $\mathtt{mgu}$ (Section 4.5). As mentioned in Section 4.5 any program that is typeable with the original system is typeable in the system with biased $\mathtt{mgu}$. In the latter system, it follows that, under the Hindley-Milner restrictions, the pattern judgement will return a substitution that only refines freshly-introduced type variables, because each equality passed to $\mathtt{mgu}$ will be of form $\tau_{\mathrm{p}} \doteq \mathtt{a}$, where $\mathtt{a}$ is a freshly-introduced type variable:

LEMMA 6.8 (Shapes of refinements under HM restrictions). *If algebraic datatypes are conventional, biased* $\mathtt{mgu}$*s are used, and* $\Gamma \vdash \mathtt{p} :^{\mathrm{m}} \tau \blacktriangleright (\overline{\mathtt{a}},\ \Delta,\ \theta)$, *then* $\mathtt{dom}(\theta) \subseteq \overline{\mathtt{a}}$.

For the proof of Theorem 6.7 we additionally rely on the fact that if we apply the refinement returned by the pattern typing judgement to the extra part of the environment that the pattern introduces, we get back the part of the environment that the Hindley-Milner system would introduce. The "conservativity" part is proved using the intermediate system that uses biased $\mathtt{mgu}$s and the fact that this system is equivalent to the original that uses arbitrary $\mathtt{mgu}$s.

7. Related work

In the dependent types community, GADTs have played a central role for over a decade, under the name *inductive families of data types* [7]. Coquand in his work on dependently typed pattern matching [6] also uses a unification based mechanism for implementing the refinement of knowledge gained through pattern matching. These ideas were incorporated in the ALF proof editor [10], and have evolved into dependently-typed programming languages such as Cayenne [1] and Epigram [11]. In the form presented here, GADTs can be regarded as a special case of dependent typing, in which the separation of types from values is maintained, with all the advantages and disadvantages that this phase separation brings.

The idea of GADTs in practical programming languages dates back to Zenger's system of indexed types [26], but Xi *et al* were perhaps the first to suggest including GADTs in an ML-like programming language [24]. (In fact, an earlier unpublished work by Augustsson and Petersson proposed the same idea [2].) Xi's subsequent work adopts more ideas from the dependent-type world [25, 23]. Cheney and Hinze examine numerous uses of what they call *first class phantom types* [5, 8]. Sheard and Pasalic use a similar design they call *equality-qualified types* in the language Ωmega [18]. All of these works employ sets of (equality) constraints to describe the type system. We use unification instead, for reasons we discussed in Section 4.3.

Jay's *pattern calculus* [9] also provides the same kind type refinement via pattern matching as ours does, and it inspired our use of unification as part of the declarative type-system specification. The pattern calculus aims at a different design space than ours, choosing to lump all all data type constructors into a single pool. This allows Jay to relax his rule for typing constructors. As our intended target is Haskell, where for historical and efficiency reasons constructors for different datatypes can have overlapping representations in memory, we cannot make this same design choice.

Most of this work concerns type *checking* for GADTs. Much less has been done on type *inference*. An unpublished earlier version of this paper originally proposed the idea of wobbly types, but in a more complicated form than that described here [14]. In that work, the wobbly/rigid annotations were part of the syntax of *types* whereas, in this paper, a type is either entirely rigid or entirely wobbly. For example, in the present system, `case (x,y) of ...` will do no type refinement if either `x` or `y` has a wobbly type, whereas before the rigidity of either `x` or `y` would lead to type refinement of the corresponding sub-pattern. However, this fine-grain attribution of wobbliness gave rise to significant additional complexity (such as "wobbly unification"), which is not necessary here, and we believe that the gain is simply not worth the pain. We believe, but have not formally proved, that every program in the language of the earlier draft is typeable in the current system—perhaps with the addition of a few more type annotations.

Inspired by the wobbly-type idea, Pottier and Régis-Gianas found a way to factor the complexity into three parts: a *shape-inference* phase that works out which types are rigid and which are wobbly, a straightforward *constraint generation* phase that turns the program text into a set of constraints, followed by a *constraint-solving* phase [15]. They call this process *stratified type inference*. The novelty is in the shape inference part; the constraint generation/solving idea is now well established. The result is a rather sophisticated type system

that can infer types for some programs that our system would reject. Here is an example, taken from their paper, using the `Rep` type defined in Section 5.4:

```
double :: Rep a -> [a] -> [a]
  = \r xs. map (\x. case r of { RI -> x+x }) xs
```

In our system, `x` would be given a wobbly type, and hence would not enjoy the type refinement induced by the `case` on `r`, so the program would be rejected. To fix the problem is easy: annotate the binding of `x`:

```
double :: Rep a -> [a] -> [a]
  = \r xs. map (\(x::a). case r of { RI -> x+x }) xs
```

Stratified type inference can accept the program without the `(x::a)` annotation. The price to be paid is that the system is much more complicated than the one we present here; for example, it is non-trivial to figure out whether the annotation on `x` is required or not. Our gut feeling is that the extra annotation burden that we impose, compared to stratified inference, is barely noticeable, but we need more experience to be sure.

Stuckey and Sulzmann also tackle the problem of type inference for GADTs [19]. They generate constraints and then solve them, but unlike Pottier *et al* they do not require a shape inference phase to precisely describe necessary type annotations. Instead, their inference algorithm, which also attacks polymorphic recursion, is incomplete. To assist users whose code does not type check, they develop a set of heuristics to identify where more type annotations are required. As a result, their compiler will accept programs with fewer type annotations than our system (or stratified type inference) requires, but these programs must be developed with the assistance of their compiler.

8. Conclusions and further work

We believe that expressive languages will shift increasingly towards type systems that exploit and propagate programmer annotations. Polymorphic recursion and higher-rank types are two established examples, and GADTs is another. We need tools to describe such systems, and the wobbly types we introduce here seem to offer a nice balance of expressiveness with predictabilty and simplicity of type inference. Furthermore, the idea of distinguishing programmer-specified types from inferred ones may well be useful in applications beyond GADTs. The main shortcoming of our implementation in GHC is that the interaction between GADTs and type classes is not dealt with properly. We plan to address this, along the lines proposed by Sulzmann [21].

Acknowledgements We thank François Pottier for his particularly detailed and insightful feedback on our draft. We also thank Martin Sulzmann for many fruitful conversations on related topics as well as comments on this paper. Matthew Fluet gave us helpful comments on an early draft.

References

[1] Lennart Augustsson. Cayenne — a language with dependent types. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 239–250, Baltimore, 1998. ACM.

[2] Lennart Augustsson and Kent Petersson. Silly type families. Available as `http://www.cs.pdx.edu/~sheard/papers/silly.pdf`, 1994.

[3] AL Baars and SD Swierstra. Typing dynamic typing. In *ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 157–166, Pittsburgh, September 2002. ACM.

[4] Luca Cardelli. Basic polymorphic typechecking. *Polymorphism*, 2(1), January 1985.

[5] James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

[6] Thierry Coquand. Pattern matching with dependent types. In *Proc workshop on Logical Frameworks, Bastaad*, pages 66–79, 1992.

[7] Peter Dybjer. Inductive Sets and Families in Martin-Lf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.

[8] Ralf Hinze. Fun with phantom types. In Jeremey Gibbons and Oege de Moor, editors, *The fun of programming*, pages 245–262. Palgrave, 2003.

[9] Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems*, 26:911–937, November 2004.

[10] Lena Magnusson. *The implementation of ALF - a proof editor based on Martin-Löf's monomorhic type theory with explicit substitution*. PhD thesis, Chalmers University, 1994.

[11] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[12] Erik Meijer and Koen Claessen. The design and implementation of Mondrian. In J Launchbury, editor, *Haskell workshop*, Amsterdam, 1997.

[13] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[14] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Microsoft Research, 2004.

[15] François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *ACM Symposium on Principles of Programming Languages (POPL'06)*, Charleston, January 2006. ACM.

[16] Tim Sheard. Languages of the future. In *ACM Conference on Object Orientated Programming Systems, Languages and Applicatioons (OOPSLA'04)*, 2004.

[17] Tim Sheard. Putting Curry-Howard to work. In *Proceedings of ACM Workshop on Haskell, Tallinn*, pages 74–85, Tallinn, Estonia, September 2005. ACM.

[18] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proc 4th international workshop on logical frameworks and meta-languaegs (LFM'04), Cork*, July 2004.

[19] Peter Stuckey and Martin Sulzmann. Type inference for guarded recursive data types. Technical report, National University of Singapore, 2005.

[20] Martin Sulzmann. A Haskell programmer's guide to Chameleon. Available at `http://www.comp.nus.edu.sg/~sulzmann/chameleon/download/haskell.html`, 2003.

[21] Martin Sulzmann, Jeremy Wazny, and Peter Stuckey. A framework for extended algebraic data types. Technical report, National University of Singapore, 2005.

[22] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Simple unification-based type inference for GADTs, Technical Appendix. Technical Report MS-CIS-05-22, University of Pennsylvania, November 2005.

[23] Hongwei Xi. Applied type system. In *Proceedings of TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer Verlag, 2004.

[24] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235. ACM Press, 2003.

[25] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227, San Antonio, January 1999. ACM.

[26] Christoph Zenger. Indexed types. *Theoretical Computer Science*, pages 147–165, 1997.