Wobbly types: type inference for generalised algebraic data types

Simon Peyton Jones
Microsoft Research, Cambridge

Geoffrey Washburn
University of Pennsylvania

Stephanie Weirich
University of Pennsylvania

Abstract

Generalised algebraic data types (GADTs), sometimes known as "guarded recursive data types" or "first-class phantom types", are a simple but powerful generalisation 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.

It is time to pluck the fruit. Can GADTs be added to Haskell, without losing type inference, or requiring unacceptably heavy type annotations? Can this be done without completely rewriting the already-complex Haskell type-inference engine, and without complex interactions with (say) type classes? We answer these questions in the affirmative, giving a type system that explains just what type annotations are required, and a prototype implementation that implements it. 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

Generalised algebraic data types (GADTs) are a simple but potent generalisation 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, first-class phantom types, equality-qualified types, and so on), although they have a much longer history in the dependent types community (see Section 6). Any feature with so many names must be useful — and indeed these papers and others give many compelling examples, as we recall in Section 2.

Submitted to POPL'05: July 19, 2004

We seek to turn GADTs from a specialised hobby into a mainstream programming technique. To do so, we have incorporated them as a conservative extension of Haskell (a similar design would work for ML). The main challenge is the question of *type inference*, a dominant feature of Haskell and ML. It is well known that GADTs are too expressive to admit type inference in the absence of any programmer-supplied type annotations; on the other hand, when enough type annotations are supplied, type inference is relatively straightforward.

One approach, then, is to implement a compiler that takes advantage of type annotations. If the algorithm succeeds, well and good; if not, the programmer adds more annotations until it does succeed. The difficulty with this approach is that there is no guarantee that another compiler for the same language will also accept the annotated program, nor does the programmer have a precise specification of what programs are acceptable and what are not. With this in mind, our central focus is this: we give a declarative type system for a language that includes GADTs and programmer-supplied type annotations, which has the property that type inference is straightforward for any well-typed program. More specifically, we make the following contributions:

- We describe an *explicitly-typed* target language, in the style of System F, that supports GADTs (Section 3). This language differs in only minor respects from that of Xi [XCC03], but our presentation of the type system is rather different and, we believe, more accessible to programmers. In addition, some design alternatives differ and, most important, it allows us to introduce much of the vocabulary and mental scaffolding to support the main payload. We prove that the type system is sound.
- We describe an *implicitly-typed* source language, that supports GADTs and programmer-supplied type annotations (Section 4), and explore some design variations, including lexically-scoped type variables (Section 4.7). The key innovation in the type system is the notion of a *wobbly type*, which models the places where an inference algorithm would make a "guess". The idea is that the type refinement induced by GADTs never "looks in-

side” a wobbly type, and hence is insensitive to the order in which the inference algorithm traverses the tree. We prove various properties of this system, including soundness.

- We have built a prototype implementation of the system as an extension to the type inference engine described by Peyton Jones and Shields’s tutorial [PS04]. We discuss the interesting aspects of the implementation in Section 5.

Our focus is on type *inference* rather than *checking*, unlike most previous work on GADTs. The exception is an excellent paper by Simonet and Pottier, written at much the same time as this one, and which complements our work very nicely [SP03]. Their treatment is more general (they use HM(X) as the type framework), but we solve two problems they identify as particularly tricky. First, we support lexically-scoped type variables and open type annotations; and second we use a single set of type rules for all data types, rather than one set for “ordinary” data types and another for GADTs. We discuss this and other important related work in Section 6.

Our goal is to design a system that is *predictable* enough to be used by ordinary programmers; and *simple* enough to be implemented without heroic efforts. In particular, we are in the midst of extending the Glasgow Haskell Compiler (GHC) to accommodate GADTs. GHC’s type checker is already very large; not only does it support Haskell’s type classes, but also numerous extensions, such as multi-parameter type classes, functional dependencies, scoped type variables, arbitrary-rank types, and more besides. An extension that required all this to be re-engineered would be a non-starter but, by designing our type system to be type-inference-friendly, we believe that GADTs can be added as a more or less orthogonal feature, without disturbing the existing architecture.

More broadly, we believe that the goal of annotation-free type inference is a mirage; and that expressive languages will shift increasingly towards type systems that exploit 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 meet that need.

2 Background

We begin with a brief review of the power of GADTs — nothing in this section is new. Consider the following data type for terms in a simple language of arithmetic expressions:

```
data Term = Lit Int  | Inc Term
          | IsZ Term | If Term Term Term
          | Fst Term | Snd Term | Pair Term Term
```

We might write an evaluator for this language as follows:

```
data Val = VInt Int | VBool Bool | VPr Val Val

eval :: Term -> Val
eval (Lit i) = VInt i
eval (Inc t) = case eval t of
                  VInt i -> VInt (i+1)
eval (IsZ t) = case eval t of
                  VInt i -> VBool (i==0)
eval (If b t e) = case eval b of
                    VBool True -> eval t
                    VBool False -> eval e
..etc..
```

There are obvious infelicities in both the data type and the evaluator. The data type allows the construction of nonsense terms, such as `(Inc (IfZ (Lit 0)))`; and the evaluator does a good deal of fruitless tagging and un-tagging.

Now suppose that we could instead write the data type declaration like this:

```
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
```

Here we have added a type parameter to `Term`, which indicates the type of the term it represents, and we have enumerated the constructors, giving each an explicit type signature. These type signatures already rule out the nonsense terms; in the example above, `(IfZ (Lit 0))` has type `Term Bool` and that is incompatible with the argument type of `Inc`.

Furthermore, the evaluator 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 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. Under the dictum “well typed programs do not go wrong”, this program is definitely well-typed.

The key ideas are these:

- A generalised 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 $\forall\overline{\alpha}\,.\,\overline{\tau} \to \mathsf{T}\ \overline{\alpha}$, where the result type is the type constructor `T` applied to all the type parameters $\overline{\alpha}$. In a generalised 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. Pattern-matching against a constructor may allow a *type refinement* in the case alternative. For example, in the `Lit` branch of `eval`, we can refine `a` to `Int`.

- Type inference is only practicable when guided by type annotations. For example, in the absence of the type signature for `eval`, a type inference engine would have to miraculously 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.

- The dynamic semantics is unchanged. Pattern-matching is done on data constructors only, and there is no run-time type passing.

This simple presentation describes GADTs as a modest generalisation of conventional data types. One can generalise still further, by regarding the constructors as having *guarded*, or *qualified* types [XCC03, SP03]. For example:

$$\mathtt{Lit} :: \forall\alpha\,.\,(\alpha = \mathtt{Int}) \Rightarrow \mathtt{Term}\ \alpha$$

This use of explicit constraints has the advantage that it can be generalised to more elaborate constraints, such as subtype constraints. But it has the disadvantage that it exposes programmers to a much richer and more complicated world. In keeping with our practical focus, our idea is to see how far we can get without mentioning constraints to the programmer at all – indeed, they barely show up in the presentation of the type system. Our approach is less general, but it has an excellent power-to-weight ratio.

The `eval` function is a somewhat specialised example, but earlier papers have given many other applications of GADTs, including generic programming, modelling programming languages, maintaining invariants in data structures (e.g. red-black trees), expressing constraints in domain-specific embedded languages (e.g security constraints), and modelling objects [Hin03, XCC03, CH03, SP04, She04]. The interested reader should consult these works; meanwhile, for this paper we simply take it for granted that GADTs are useful.

$$\begin{array}{ll}
\text{Variables} & x, y, z \\
\text{Type variables} & \alpha, \beta \\
\text{Type constructors} & \mathsf{T} \\
\text{Data constructors} & C
\end{array}$$

$$\begin{array}{lrcl}
\text{Programs} & \mathit{prog} & ::= & \overline{d}\ t \\
\text{Data types} & d & ::= & \mathtt{data}\ \mathsf{T}\overline{\alpha}\ \mathtt{where}\ \overline{C\mathtt{::}\forall\overline{\alpha}\,.\,\overline{\sigma} \to \mathsf{T}\ \overline{\xi}} \\
\text{Atoms} & \nu & ::= & x \mid C \\
\text{Terms} & t, u & ::= & \nu \mid \lambda p\,.\,t \mid \Lambda\alpha\,.\,t \mid t\ u \mid t\ \sigma \\
 & & \mid & \mathtt{let}\ x_\sigma\ \mathtt{=}\ u\ \mathtt{in}\ t \\
 & & \mid & \mathtt{case}(\sigma)\ t\ \mathtt{of}\ \overline{p\mathtt{->}t} \\
\text{Patterns} & p, q & ::= & x_\sigma \mid C\ \overline{\alpha}\ \overline{p}
\end{array}$$

$$\begin{array}{lrcl}
\text{Types} & \sigma, \phi, \xi & ::= & \forall\alpha.\sigma \mid \sigma \to \phi \mid \mathsf{T}\ \overline{\sigma} \mid \alpha \\
\text{Type contexts} & \Gamma, \Delta & ::= & \epsilon \mid \Gamma, \alpha \mid \Gamma, \nu : \sigma \\
\text{Constraint lists} & \Pi & ::= & \overline{\pi} \\
\text{Constraint} & \pi & ::= & \sigma_1 \doteq \sigma_2 \\
\text{Substitutions} & \theta & ::= & \emptyset \mid \bot \mid \theta, \alpha \mapsto \sigma
\end{array}$$

$$\begin{array}{lcll}
ftv(\sigma) & :: & \overline{\alpha} & \text{Free type variables of } \sigma \\
ftv(\sigma) & = & \textit{...standard...} & \\
ftv(\Gamma) & :: & \overline{\alpha} & \text{Type variables in domain of } \Gamma \\
ftv(\Gamma) & = & \bigcup\{ftv(\sigma) \mid (x : \sigma) \in \Gamma\} &
\end{array}$$

Figure 1: Syntax of the core language

3 The core language

Our first step is to describe an explicitly-typed language, in the style of System F, that supports GADTs. This language allows us to introduce much of our terminology and mental scaffolding, in a context that is relatively simple and constrained. This so-called *core language* is more than a pedagogical device, however. GHC uses a System-F-style language as its intermediate language, and the language we describe here is very close to that used by GHC (extended to support GADTs). Furthermore, the source-language type system that we describe in Section 4 gives a type-directed translation into the core language, and that is the route by which we prove soundness for the source language.

3.1 Syntax of the core language

Figure 1 gives the syntax of the core language and its types. As in System F, each binder is annotated with its type, and type abstraction $(\Lambda\alpha\,.\,t)$ and application $(t\ \sigma)$ is explicit. The `let` binding form is recursive. In this paper, every type variable has kind "*"; the extension to higher kinds is straightforward, but increases clutter. The system is impredicative, however; for example, the type application $(f\ (\forall\alpha\,.\,\alpha \to \alpha))$ is legitimate.

We use overbar notation extensively. The notation $\overline{\alpha}^n$ means the sequence $\alpha_1 \cdots \alpha_n$; the "n" may be omitted when it is unimportant. The notation $\overline{a} \# \overline{b}$ means that the two sequences have no common elements. Although we give the syntax of function types in the conventional curried way, we also sometimes use an equivalent overbar notation, thus:

$$\overline{\sigma}^n \to \phi \quad \equiv \quad \sigma_1 \to \cdots \to \sigma_n \to \phi$$

We will sometimes decompose sequences one element at a time, using the following grammar.

$$\overline{a} \quad ::= \quad \epsilon \mid a, \overline{a}$$

This notation is used in the declaration of data types (Figure 1), which are given by explicitly enumerating the constructors, giving an explicit type signature for each. The declaration of `Term` in Section 2 was an example.

Pattern-matching over these data types is done by `case` expressions. Each such expression ($\texttt{case}(\sigma)\ t\ \texttt{of}\ \overline{p\texttt{->}t}$) is decorated with its result type σ, and patterns p may be nested. (GHC's intermediate language does not have nested patterns, but the source language certainly does, and nested patterns turn out to have interesting interactions with GADTs, so it is worth exploring them here.)

Notice that a constructor pattern $(C\ \overline{\alpha}\ \overline{x_\sigma})$ binds type variables $\overline{\alpha}$ as well as term variables $\overline{x_\sigma}$. For example, here is part of the definition of `eval`, expressed in the core language:

```
eval :: forall a. Term a -> a
     = /\a. \(x:Term a).
       case(a) x of
         Lit (i:Int) -> i
         Pair b c (s::Term b) (t::Term c)
           -> (eval b s, eval c t)
         Fst b c (t:Term (b,c))
           -> fst (b,c) (eval (b,c) t)
         ...etc...
```

Each constructor pattern binds a (fresh) type variable for each universally-quantified type variable in the constructor's type. Then, as we shall see, the same type refinement that refines `a` to `Int` in the `Lit` case will refine `a` to `b` (or vice versa) in the `If` case, and will refine `a` to `(b,c)` in the `Fst` case. Notice, too, the essential use of polymorphic recursion: the recursive calls to `eval` are at different types than `a`. To be useful, a language that offers GADTs must also support polymorphic recursion.

$$\begin{array}{rcl}
\mathcal{U}(\overline{\alpha};\Pi) & :: & \theta \\
\mathcal{U}(\overline{\alpha};\epsilon) & = & \emptyset \\
\mathcal{U}(\overline{\alpha};\sigma \doteq \sigma,\Pi) & = & \mathcal{U}(\overline{\alpha};\Pi) \\
\mathcal{U}(\overline{\alpha};\beta \doteq \sigma_\in,\Pi) & = & \mathcal{U}(\overline{\alpha};\Pi[\sigma_\in/\beta]) \circ [\beta \mapsto \sigma_\in] \\
 & & \beta \notin \overline{\alpha}, \beta \notin \sigma_2 \\
\mathcal{U}(\overline{\alpha};\sigma_\infty \doteq \beta,\Pi) & = & \mathcal{U}(\overline{\alpha};\Pi[\sigma_\infty/\beta]) \circ [\beta \mapsto \sigma_\infty] \\
 & & \beta \notin \overline{\alpha}, \beta \notin \sigma_1 \\
\multicolumn{3}{l}{\mathcal{U}(\overline{\alpha};\sigma_\infty \to \sigma'_\infty \doteq \sigma_\in \to \sigma'_\in,\Pi)} \\
 & = & \mathcal{U}(\overline{\alpha};\sigma_\infty \doteq \sigma_\in, \sigma'_\infty \doteq \sigma'_\in,\Pi) \\
\mathcal{U}(\overline{\alpha};\mathcal{T}\ \overline{\sigma_\infty} \doteq \mathcal{T}\ \overline{\sigma_\in},\Pi) & = & \mathcal{U}(\overline{\alpha};\overline{\sigma_\infty \doteq \sigma_\in},\Pi) \\
\mathcal{U}(\overline{\alpha};\forall\beta\ .\ \sigma_\infty \doteq \forall\beta\ .\ \sigma_\in,\Pi) & = & \mathcal{U}(\beta,\overline{\alpha};\sigma_\infty \doteq \sigma_\in,\Pi) \\
\mathcal{U}(\overline{\dashv};\tau_\infty \doteq \tau_\in,\Pi) & = & \bot
\end{array}$$

Figure 3: Core-language most general unification

3.2 Type system of the core language

Figure 2 gives the type system of the core language. We omit rules for data type declarations d, because they simply populate the environment Γ with the types of the constructors. The main typing judgement $\Gamma \vdash t : \sigma$ is absolutely standard. The auxiliary judgement $\Gamma \overset{k}{\vdash} \sigma$ checks that σ is well-kinded; since all type variables have kind `*`, the judgement checks only that the type variables of σ are in scope, and that applications of type constructors T are saturated. We omit the details.

Pattern-matching is where all the interest lies. The judgement

$$\Gamma \overset{a}{\vdash} p \to u : \sigma_1 \to \sigma_2$$

checks each alternative of a `case` expression. Intuitively, both σ_1 and σ_2 should be considered as inputs to this judgement; the former is the type of the scrutinee, while the latter annotates the `case` (see rule CASE). The first thing that $\overset{a}{\vdash}$ does is to typecheck the (possibly nested) pattern, using a judgement of form

$$\Gamma;\Delta;\theta \overset{p}{\vdash} p : \sigma;\Delta';\theta'$$

Here, Γ is the type environment giving the types of the constructors, p is the pattern, and σ is the type of the pattern. The judgement also takes a mini-environment Δ describing the type and term variables bound to the left of the pattern p in the current pattern-match, and extends it with the bindings in p to give Δ'. The bindings Δ are "threaded" top-down and left-to-right through the patterns, starting off empty, ϵ, in rule ALT. The threading work is done by the auxiliary judgement $\overset{ps}{\vdash}$, which simply invokes $\overset{p}{\vdash}$ successively on a sequence of patterns. This threading makes it easy to check for repeated variables — the test $x \# dom(\Delta)$ in rule PVAR — but our real motivation for threading Δ will not be apparent until Section 4.7.

The least conventional feature of the pattern judgement is a *substitution*, or *type refinement*, θ, to which we have already referred informally; again, the judgement takes a type refinement θ and extends it with type refinements from p to give θ'. The type refinement is threaded in exactly the same way as Δ, for reasons we discuss in Section 3.4.

Let us focus on rule PCON, which deals with a constructor pattern $C\overline{\alpha}\ \overline{p}$. First, we look up the type of C in Γ; it has argument types $\overline{\sigma}^c$ (where c is the arity of the constructor)

$$\boxed{\text{Terms } \Gamma \vdash t : \sigma}$$

$$\frac{(\nu : \sigma) \in \Gamma}{\Gamma \vdash \nu : \sigma}\ \text{ATOM} \qquad \frac{\Gamma \vdash t : \sigma' \to \sigma \qquad \Gamma \vdash u : \sigma'}{\Gamma \vdash t\ u : \sigma}\ \text{TERM-APP} \qquad \frac{\Gamma \vdash^{k} \sigma \qquad \Gamma \vdash t : \forall \alpha\,.\,\sigma'}{\Gamma \vdash t\ \sigma : \sigma'[\sigma/\alpha]}\ \text{TYPE-APP}$$

$$\frac{\Gamma \vdash^{a} p\texttt{->}t : \sigma \to \sigma'}{\Gamma \vdash (\lambda p\,.\,t) : \sigma \to \sigma'}\ \text{TERM-LAM} \qquad \frac{\Gamma, \alpha \vdash t : \sigma \qquad \alpha \# \mathrm{dom}(\Gamma)}{\Gamma \vdash (\Lambda \alpha\,.\,t) : \forall \alpha\,.\,\sigma}\ \text{TYPE-LAM} \qquad \frac{\Gamma, x : \sigma \vdash u : \sigma \qquad \Gamma, x : \sigma \vdash t : \sigma'}{\Gamma \vdash (\texttt{let}\ x_{\sigma}\texttt{=}u\ \texttt{in}\ t) : \sigma'}\ \text{LET}$$

$$\frac{\Gamma \vdash^{k} \sigma_2 \qquad \Gamma \vdash t : \sigma_1 \qquad \Gamma \vdash^{a} \overline{p\ \texttt{->}\ u} : \sigma_1 \to \sigma_2}{\Gamma \vdash (\texttt{case}(\sigma_2)\ t\ \texttt{of}\ \overline{p\ \texttt{->}\ u}) : \sigma_2}\ \text{CASE}$$

$$\boxed{\text{Case alternatives } \Gamma \vdash^{a} p \to u : \sigma_1 \to \sigma_2}$$

$$\frac{\Gamma; \epsilon; \emptyset \vdash^{p} p : \sigma_1; \Delta; \theta \qquad \theta(\Gamma, \Delta) \vdash \theta(u) : \theta(\sigma_2)}{\Gamma \vdash^{a} p \to u : \sigma_1 \to \sigma_2}\ \text{ALT} \qquad \frac{\Gamma; \epsilon; \emptyset \vdash^{p} p : \phi; \Delta; \bot}{\Gamma \vdash^{a} p \to u : \phi \to \sigma}\ \text{ALT-FAIL}$$

$$\boxed{\text{Patterns } \Gamma; \Delta; \theta \vdash^{p} p : \sigma; \Delta'; \theta'}$$

$$\frac{}{\Gamma; \Delta; \bot \vdash^{p} p : \sigma; \epsilon; \bot}\ \text{PFAIL} \qquad \frac{x \# \mathrm{dom}(\Delta) \qquad \theta(\sigma) = \theta(\phi)}{\Gamma; \Delta; \theta \vdash^{p} x_{\sigma} : \phi; \Delta, (x : \sigma); \theta}\ \text{PVAR}$$

$$\frac{\begin{array}{c}(C : \forall \overline{\alpha}\,.\,\overline{\sigma}^{c} \to T\ \overline{\xi}^{t}) \in \Gamma \qquad \overline{\alpha} \# \mathrm{dom}(\Gamma, \Delta) \\ \theta(\phi) = T\ \overline{\xi'}^{t} \qquad \theta' = \mathcal{U}(\mathcal{T}\ \overline{\xi}^{\sqcup} \doteq \mathcal{T}\ \overline{\xi'}^{\sqcup}) \\ \Gamma; (\Delta, \overline{\alpha}); \theta' \circ \theta \vdash^{ps} \overline{p : \sigma}^{c}; \Delta''; \theta''\end{array}}{\Gamma; \Delta; \theta \vdash^{p} C\ \overline{\alpha}\ \overline{p}^{c} : \phi;\ \Delta''; \theta''}\ \text{PCON}$$

$$\boxed{\text{Pattern sequences } \Gamma; \Delta; \theta \vdash^{ps} \overline{p : \sigma}; \Delta'; \theta'}$$

$$\frac{}{\Gamma; \Delta; \theta \vdash^{ps} \epsilon; \Delta; \theta}\ \text{PEMPTY} \qquad \frac{\Gamma; \Delta; \theta \vdash^{p} p : \sigma; \Delta'; \theta' \qquad \Gamma; \Delta'; \theta' \vdash^{ps} \overline{p : \sigma}; \Delta''; \theta''}{\Gamma; \Delta; \theta \vdash^{ps} p : \sigma, \overline{p : \sigma}; \Delta''; \theta''}\ \text{PCONS}$$

Figure 2: Core language typing rules

and result type $T\ \overline{\xi}^{t}$ (where t is the arity of T). We require that the binding type variables $\overline{\alpha}$ are not already in scope, and we quietly alpha-rename the constructor's type to use these variables. Now comes the interesting bit: we must match up the constructor's result type $T\ \overline{\xi}$ with the pattern's type in the conclusion of the rule, ϕ. We do this in two steps. First, the test $\theta(\phi) = T\ \overline{\xi'}$ checks that the result type in the conclusion, ϕ, when refined by the type refinements induced by "earlier" patterns, is an application of T to some types $\overline{\xi'}$. Second, we unify the constructor's result type $T\ \overline{\xi}$ with the pattern type $T\ \overline{\xi'}$, using the function $\mathcal{U}$ that computes the most-general unifier, and use the resulting substitution to extend the type refinement. The definition of $\mathcal{U}$ is given in Figure 3, and is standard, apart from a straightforward extension to handle polymorphic types.

Returning to rule ALT, once $\vdash^{p}$ has type-checked the pattern, we typecheck the right-hand side of the alternative, u, *under the type refinement* θ. To achieve this effect, we simply apply θ to Γ, Δ, u, and the result type σ_2, before checking the type of u.

A subtlety of rule PCON is the separation of the third and fourth preconditions. It is possible to combine them, to obtain the single condition $\theta' = \mathcal{U}(\theta(\phi) \doteq \mathcal{T}\ \overline{\xi}^{\sqcup})$ However, doing so makes too many programs typeable. For example:

```
/\a. \x:a. case x of { (a,b) -> (b,a) }
```

This program would be regarded as ill-typed by ML or Haskell, because a variable of polymorphic type a is treated as a pair type, and it is indeed rejected by the third precondition of rule PCON. Under the above modification, though, it would be regarded as acceptable, because `a` is refined to a pair type by θ; and indeed, such a modification is perfectly sound provided the dynamic semantics can distinguish between constructors of different types (`False` and `Nil`, say). Precisely this design choice is made by Jay [Jay03]. Nevertheless, we adopt the Haskell/ML view here, for reasons both of principle (more errors are discovered at compile time) and practice (implementation of the new dynamic semantics would be difficult).

3.3 An example

It is instructive to see how the rules work in an example. Here is the first part of the body of `eval`:

```
/\a. \(x:Term a). case(a) x of
                    Lit (i:Int) -> i
```

Since `Lit` has type `Int -> Term Int`, the pattern binds no type variables. Rule CASE invokes the judgement

$$\Gamma \vdash^{a} \texttt{Lit (i:Int) -> i} : \texttt{Term a} \rightarrow \texttt{a}$$

In rule PCON we unify `Term Int` (the result type of `Lit`) with `Term a`, to obtain the type refinement $[\texttt{a} \mapsto \texttt{Int}]$. Then rule ALT applies that substitution to the right-hand side `i` and result type `a`, and type-checks the right-hand side, which succeeds.

The next alternative is more interesting:

```
Pair b c (t1::Term b) (t2::Term c)
  -> (eval b t1, eval c t2)
```

The pattern binds two new type variables `b` and `c`, and generates the type refinement $[\texttt{a} \mapsto \texttt{(b,c)}]$; again, the right hand side type-checks once this substitution is applied to the result type `a`. We discuss just one other constructor, `Fst`, which has the interesting property that it has an existential type variable (because one of the quantified type variables does not appear in the result type of the constructor):

```
Fst b c (t:Term (b,c))
   -> fst (b,c) (eval (b,c) t)
```

As with `Pair`, the pattern binds two fresh type variables `b` and `c`. The result type of the constructor is just `Term b` – it does not mention `a` – so rule PCON forms $\mathcal{U}(\texttt{Term b} \doteq \texttt{Term a})$, yielding either the substitution $[\texttt{a} \mapsto \texttt{b}]$ or $[\texttt{b} \mapsto \texttt{a}]$. The reader may want to confirm that in either case the right hand side is well typed.

3.4 Nested patterns

Consider these two functions (we omit big lambdas and some type annotations):

```
f1 (x::Term a) (y::a)
   = case(a) x of
        Pair p q -> case(a) y of
                       (r,s) -> (p,s)
f2 (x::Term a) (y::a)
   = case(a) (x,y) of
        (Pair p q, (r,s)) -> (p,s)
```

It should be clear that `f1` is well-typed, because the type refinement induced by the pattern match on `x` is "seen" by the inner `case`; in that context, `y` has a pair type so the case makes sense. If the two cases were reversed, the function would be ill-typed. But what about `f2`? Here, the two cases are merged into one; but is the left-hand match "seen" by the right-hand one?

This is an open design choice, and other things being equal the more refinement the better, so we provide for left-to-right refinement. This is the reason that we "thread" the substitution in our pattern judgement. A consequence is that the compiler must generate code that matches patterns 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, as indeed Simonet and Pottier do [SP03]

3.5 Meta-theory

We have proved that the type system of Figure 2 is sound with respect to the obvious small-step dynamic semantics (omitted here for lack of space).

THEOREM 3.1 (TYPE SOUNDNESS FOR CORE LANGUAGE). *If* $\epsilon \vdash \texttt{t} : \sigma$ *then* e *either evaluates to a value or diverges.*

Our dynamic semantics does not depend on type information at run time – one may erase all types without affecting execution. Our definition of values is also standard. We prove type soundness using the standard progress and preservation lemmas.

We have also proved that type checking is decidable. That is, given a well-formed context Γ and an expression `t`, it is decidable whether there exists a σ such that $\Gamma \vdash \texttt{t} : \sigma$. Because our rules are syntax-directed, showing that type checking is decidable is straightforward, given that $\mathcal{U}$ is decidable [Rob65]. The type checking algorithm may be read from the inference rules.

We are more used to seeing unification in type *inference* algorithms, and it is unusual to see it in declarative type checking rules. The best way to think about it is this. A successful pattern match implies the truth of certain equality constraints among types, all of form $\mathsf{T}\ \overline{\xi} \doteq \mathsf{T}\ \overline{\xi'}$, and the case alternative should be checked under these constraints. However, rather than add a set of constraints to the environment, and reason about type equality modulo those constraints, we solve the constraints to get their most general unifier, and apply the resulting substitution. We find that it is much easier to explain the type system to programmers in terms of an eagerly-applied substitution than by speaking of constraint sets – and the usual question of whether or not one can abstract over constraints simply does not arise. In effect, we are exploiting the special case of equality constraints to simplify the technology.

This use of $\mathcal{U}$ is not new – it was used in essentially the same way by Coquand [Coq92] and by Jay [Jay03] (from whom we learned the trick). It is essential for soundness that the $\mathcal{U}$ function indeed delivers the *most general* unifier. (In contrast, in the conventional use of unification for type inference, any unifier is sound.) Why? Because the constraints gathered in the patterns are treated as *facts* in the case alternative, and we cannot soundly invent new facts – for example, we cannot suddenly assume that α is `Int`.

Technically, the fact that $\mathcal{U}$ must be a most general unifier shows up in the type substitution lemma of the type soundness proof. In this case, we must show that even though a type substitution θ may produce a different refinement for branches of a case alternative, those branches are still well typed after substitution with θ and this new refinement. However, the new refinement composed with θ is a unifier for the original types, and the original refinement was the most general unifier. Therefore, we know that the new refinement and θ is some substitution θ' composed with the original refinement. We know the branch was well-typed with the original refinement, so by induction, it must be well-typed after substitution by θ'.

Since the most-general unifier contains exactly the same information as the original constraints, the choice of whether to pass constraints around or apply a unifier is a matter of taste. Both choices lead to languages that type check the same terms. However, the two choices require different meta-theory – the proofs of type soundness and the decidability of type checking are a bit different for the two alternatives, although the proofs for both alternatives seem to be of equivalent complexity.

Most other authors have chosen the alternative path, of dealing with constraint sets explicitly. For example, Xi et al. [XCC03] and Cheney and Hinze [CH03] design explicitly-typed versions of System F that include equality constraints in the typing environment. These equality constraints admit a straightforward type soundness proof. However, showing that type checking is decidable (done by Cheney and Hinze, but not by Xi et al.) is more difficult. Because the typing rules are not syntax directed (one rule allows any expression to be typed with any equivalent type), decidable type checking requires putting these typing derivations into a normal form, which requires computing their most general unifier.

4 The source language

We now move on to consider the source language. The language is implicitly typed in the style of Haskell, but the type system is far too rich to permit type inference in the absence of any help from the programmer, so type inference is guided by programmer-supplied type annotations. The type system specifies precisely what annotations are required to make a program well-typed.

If the type system accepts too many programs, it may be effectively un-implementable, either in theory (by being undecidable) or in practice (by being too complicated). Since we want to implement it, we must be careful to write typing rules that reject hard-to-infer programs. A good example of this principle is the treatment of polymorphic recursion in Haskell 98. A program that uses polymorphic recursion might in principle have a valid typing derivation, but it is hard to find it. So we reject the program unless the offending function has a type signature. It follows, of course, that programmer-supplied type annotations play a key role in the typing judgements.

Since our goal is tractable inference, we must speak, at least informally, about inference algorithms. A potent source of confusion is that, as in Section 3, unification forms part of the *specification* of the type system (when pattern-matching on GADTs), and also forms part of the *implementation* of the type inference algorithm. We keep the two rigorously separate. Where confusion may arise we call the former *match-unification* and the latter *inference-unification*. We also describe the substitution arising from match-unification as a *type refinement*.

4.1 Syntax

The syntax of the source language is in Figure 4. Unlike the core language, binders have no compulsory type annotation, nor are type abstractions or applications present. Instead, the programmer may optionally supply a type annotation on a term, thus (`t::ty`). For example, here is part of `eval` again, in our source language:

```
eval :: forall a. Term a -> a
     = \x. case x of
            Lit i    -> i
            Pair s t -> (eval s, eval t)
            Fst t    -> fst (eval t)
            ...etc...
```

Compared to the core-language version of the same function in Section 3, the source language has implicit type abstraction and application, and variables are not annotated with their types (except in `letrec`).

4.2 Source types and internal types

We distinguish between *source types*, `ty`, and *internal types*, σ, and similarly between source type variables `a` and internal

$$
\begin{array}{llll}
\text{Variables} & & & x, y, z \\
\text{Type constructors} & & & T \\
\text{Data constructors} & & & C \\
\text{Source type variables} & & & a, b, c \\
\text{Target type variables} & & & \alpha, \beta \\
\\
\text{Atoms} & \nu & ::= & x \mid C \\
\text{Terms} & t, u & ::= & \nu \mid \lambda p.t \mid t\,u \mid t\texttt{::}ty \\
 & & \mid & \texttt{let } x \texttt{ = } u \texttt{ in } t \\
 & & \mid & \texttt{letrec } x\texttt{::}ty \texttt{ = } u \texttt{ in } t \\
 & & \mid & \texttt{case}\ t\ \texttt{of}\ \overline{p\ \texttt{->}\ t} \\
\text{Patterns} & p, q & ::= & x \mid C\,\overline{p} \\
\text{Source types} & ty & ::= & a \mid ty_1\texttt{->}ty_2 \mid T\,\overline{ty} \\
 & & \mid & \texttt{forall}\,\overline{a}\,\texttt{.}\ ty \\
\\
\text{Polytypes} & \sigma, \phi & ::= & \forall\overline{\alpha}\,.\,\tau \\
\text{Monotypes} & \tau, \upsilon & ::= & T\overline{\tau} \mid \tau_1 \to \tau_2 \mid \alpha \mid \boxed{\tau} \\
\\
\text{Type contexts} & \Gamma & ::= & \epsilon \mid \Gamma, \alpha \mid \Gamma, (\nu : \sigma) \mid \Gamma, a = \tau \\
 & \text{Note:} & \multicolumn{2}{l}{a = \tau \text{ form unused until Section 4.7}} \\
\text{Constraint lists} & \Pi & ::= & \overline{\pi} \\
\text{Constraint} & \pi & ::= & \tau_1 \doteq \tau_2 \\
\text{Substitutions} & \theta & ::= & \emptyset \mid \theta, \alpha \mapsto \tau \mid \bot
\end{array}
$$

Figure 4: Syntax of source types and terms

type variables α. The former constitute part of a source program, while the latter are used in typing judgements *about* the program. The syntax of both is given in Figure 4. An auxiliary judgement $\Gamma \vdash^{t} ty \rightsquigarrow \sigma$ checks that ty is well-kinded, and gives the internal type τ corresponding to ty. We omit the details of this judgement which is standard. For the present, we assume that ty is a closed type, a restriction we lift in Section 4.7.

The syntax of internal types is mostly conventional. It is stratified into *polytypes* (σ, ϕ), and *monotypes* (τ, υ); and it is *predicative*: type variables range over monotypes (types with no $\forall$'s within them), and the argument(s) of a type constructor are always monotypes. Predicativity makes type inference considerably easier. In our implementation in GHC, types can also have *higher kinds*, exactly as in Haskell 98, and can be of *higher rank* [PS04]. These two features turn out to be largely orthogonal to generalised data types, so we omit them here to keep the base system as simple as possible.

There is one brand-new feature, unique to this system: the "wobbly" monotypes, $\boxed{\tau}$. The intuition is this: *the un-wobbly parts of a type can all be traced to programmer-supplied type annotations, whereas the wobbly parts cannot.* Wobbly types address the following challenge. When we want to type a `case` expression that scrutinises a GADT, we must apply a different type refinement to each alternative, just as in the explicitly-typed language of Section 3. The most straightforward way to do this is to follow the type rules rather literally:

$$
\begin{array}{rcll}
ftv(\sigma) & :: & \overline{\alpha} & \text{Free type vars of } \sigma \\
ftv(\tau) & :: & \overline{\alpha} & \text{Free type vars of } \tau \\
ftv(\Gamma) & :: & \overline{\alpha} & \text{Free type vars of } \Gamma \\
ftv(\Gamma) & = & \bigcup\{ftv(\sigma) \mid (x : \sigma) \in \Gamma\} & \\
\\
\mathcal{S}(\tau) & :: & \tau & \text{Strip boxes from } \tau \\
\mathcal{S}(\alpha) & = & \alpha & \\
\mathcal{S}(T\,\overline{\tau}) & = & T\,\overline{\mathcal{S}(\tau)} & \\
\mathcal{S}(\tau_1 \to \tau_2) & = & \mathcal{S}(\tau_1) \to \mathcal{S}(\tau_2) & \\
\mathcal{S}(\boxed{\tau}) & = & \mathcal{S}(\tau) & \\
\\
\mathcal{S}(\Gamma) & :: & \Gamma & \text{Strip boxes from } \Gamma \\
\mathcal{S}(\epsilon) & = & \epsilon & \\
\mathcal{S}(\Gamma, \alpha) & = & \mathcal{S}(\Gamma), \alpha & \\
\mathcal{S}(\Gamma, (\nu : \sigma)) & = & \mathcal{S}(\Gamma), (\nu : \mathcal{S}(\sigma)) & \\
\mathcal{S}(\Gamma, (a = \tau)) & = & \mathcal{S}(\Gamma) & \\
\\
push(\tau) & :: & \tau & \text{Push boxes down one level} \\
push(\boxed{T\,\overline{\tau}}) & = & T\,\overline{\boxed{\tau}} & \\
push(\boxed{\tau_1 \to \tau_2}) & = & \boxed{\tau_1} \to \boxed{\tau_2} & \\
push(\boxed{\boxed{\tau}}) & = & push(\boxed{\tau}) & \\
\\
\theta(\tau) & :: & \tau & \text{Apply a type refinement} \\
\theta(\alpha) & = & \alpha & \text{if } \alpha \notin dom(\theta) \\
 & = & \tau & \text{if } [\alpha \mapsto \tau] \in \theta \\
\theta(T\,\overline{\tau}) & = & T\,\overline{\theta(\tau)} & \\
\theta(\tau_1 \to \tau_2) & = & \theta(\tau_1) \to \theta(\tau_2) & \\
\theta(\boxed{\tau}) & = & \boxed{\tau} & \\
\\
\theta(\Gamma) & :: & \Gamma & \text{Apply a type refinement to } \Gamma \\
\theta(\epsilon) & = & \epsilon & \\
\theta(\Gamma, \alpha) & = & \theta(\Gamma) & \text{if } \alpha \in dom(\theta) \\
 & = & \theta(\Gamma), \alpha & \text{otherwise} \\
\theta(\Gamma, (\nu : \sigma)) & = & \theta(\Gamma), (\nu : \theta(\sigma)) & \\
\theta(\Gamma, (a = \tau)) & = & \theta(\Gamma), (a = \theta(\tau)) & \\
\\
\theta_{\Uparrow}(\tau) & = & \tau & \\
\theta_{\Downarrow}(\tau) & = & \theta(\tau) &
\end{array}
$$

Figure 5: Functions over types

type-check the pattern, perform the match-unification, apply the substitution to the environment and the result type, and then type-check the right hand side. There are two potential difficulties for inference: (a) the types to be unified may not be fully known; and (b) the types to which the substitution is applied may not be fully known. Type inference typically proceeds by using meta-variables to represent as-yet-unknown monotypes, relying on inference-unification to fill them out as type inference proceeds. At some intermediate point, this filling-out process may not be complete; indeed, just how it proceeds depends on the order in which the inference algorithm traverses the syntax tree.

As a concrete example, consider this:

```
f x y = (case x of { ... }, x==[y])
```

Initially, `x` and `y` will be assigned distinct meta variables, α and β, say. If the algorithm processes the syntax tree right-to-left, the term `x==[y]` will force α to be unified to `[`β`]`, and that information might influence how match-unification takes place in the `case` expression (if it involved a GADT). On the other hand, if the algorithm works left-to-right, this information might not be available when examining the `case` expression.

Our goal is that the specification of the type system should not constrain the inference algorithm to a particular traversal order. Wobbly types are our mechanism for achieving this goal. The intuition is this:

- In the places where an inference algorithm would have to "guess" a type, we use a wobbly type to indicate that fact. For example, if lambda abstractions bound only a simple variable, the rule for abstraction would look like this:
  $$\frac{\Gamma,(x:\boxed{\tau_1})\vdash t:\tau_2}{\Gamma\vdash(\backslash x.t):(\boxed{\tau_1}\rightarrow\tau_2)}$$
  The argument type τ_1 is "presciently guessed" by the rule; an inference algorithm would use a meta type variable. So in the typing rule we use a wobbly $\boxed{\tau_1}$ to reflect the fact that x's type may be developed gradually by the inference algorithm.
- When performing match-unification in the alternatives of `case` expression scrutinising a GADT, we make no use of information inside wobbly types. We will see how this is achieved in Section 4.5.
- When applying the substitution from that match-unification, to refine the type environment and return type, we do not apply the substitution to a wobbly type: $\theta(\boxed{\tau})=\boxed{\tau}$ (Figure 5).

Wobbly types ensure the type refinements arising from GADTs are derived only from, and applied only to, types that are directly attributable to programmer-supplied type annotations. We describe a type with no wobbly types inside it as *rigid*.

4.3 Directionality

Wobbly types allow us to record where the type system "guesses" a type but, to complete the picture, we also need a way to explain when the type system must guess, and when a programmer-supplied type annotation specifies the necessary type. Our intuition is this: when the programmer can point to a simple "audit trail" that links the type of a variable to a programmer-supplied annotation, then the system should give the variable a rigid type, so that it will participate in type refinement. For example, if the programmer writes:

```
foo :: a -> T a -> T a
foo x xs = ...
```

then he might reasonably expect the system to understand that the the type of `x` is `a`, and the type of `xs` is `T a`. To make this intuitive idea precise, we annotate our typing judgements with a *directionality flag*, δ. For example, the judgement $\Gamma\vdash_{\Uparrow} t:\tau$ means "in environment Γ the term t has type τ, regardless of context", whereas $\Gamma\vdash_{\Downarrow} t:\tau$ means "in environment Γ, the term t in context τ is well-typed". The up-arrow $\Uparrow$ suggests pulling a type up out of a term ("guessing mode"), whereas the down-arrow $\Downarrow$ suggests pushing a type down into a term ("checking mode"). We use checking mode when we know the expected type for a term, because it is given by a programmer-supplied type annotation.

To see how the directionality works, here is how we split the rule for abstraction given in Section 4.2 into two, one for each direction:

$$\frac{\Gamma,(x:\boxed{\tau_1})\vdash_{\Uparrow} t:\tau_2}{\Gamma\vdash_{\Uparrow}(\backslash x.t):(\boxed{\tau_1}\rightarrow\tau_2)}\qquad\frac{\Gamma,(x:\tau_1)\vdash_{\Downarrow} t:\tau_2}{\Gamma\vdash_{\Downarrow}(\backslash x.t):(\tau_1\rightarrow\tau_2)}$$

The first rule, used when we do not know the way in which the term is to be used, is just as given in Section 4.2. On the other hand, if the type is supplied by the context, the second rule does not make τ_1 wobbly – if there is any uncertainty about it, that uncertainty should already be expressed by wobbly types inside τ_1.

The idea of adding a directionality flag to typing judgements was first published by Pierce & Turner, who called it *local type inference*. We used directionality flags to support type inference for higher-rank types in [PS04], and it is a bonus that exactly the same technology is useful here.

4.4 The typing rules

The type rules for the source language are given in Figure 6. They give a type-directed translation into the core language of Section 3. For example, the judgement $\Gamma\vdash_{\delta} t\rightsquigarrow t':\tau$ says that term t translates into the core term t', with type τ.

We begin with rule ATOM, which deals with a variable or constructor by instantiating its type, σ, using $\vdash^{inst}_{\delta}$. The latter chooses arbitrary, well-kinded wobbly types $\boxed{\overline{\upsilon}}$ to instantiate σ, the wobbliness indicating that the system must "guess" the types $\overline{\upsilon}$. Their well-kindedness is checked by $\vdash^{k}$, whose details we omit as usual. In guessing mode we are now done, but in checking mode the wobbliness may get in the way. For example, it is perfectly acceptable for a function with type $\boxed{\texttt{Int}}\rightarrow\texttt{Bool}$ to be given an argument of type `Int`, and vice versa. Hence, the auxiliary judgement $\vdash\tau\sim\tau'$ checks for equality modulo wobbliness. The function $\mathcal{S}(\tau)$, defined in Figure 5, removes wobbliness from an internal type.

Rule ANNOT invokes $\vdash^{gen}_{\Downarrow}$ to "push" the known type signature into the term, as discussed in Section 4.3. Rule LET is quite conventional, and uses $\vdash^{gen}_{\Uparrow}$ in guessing mode, while

$$\boxed{\text{Terms }\ \Gamma \vdash_{\delta} t \rightsquigarrow t' : \tau}$$

$$\frac{(\nu : \sigma) \in \Gamma \qquad \Gamma \vdash^{inst}_{\delta} \sigma \leq \tau \rightsquigarrow \overline{\upsilon}}{\Gamma \vdash_{\delta} \nu \rightsquigarrow \nu\, \overline{\upsilon} : \tau}\ \textsc{Atom} \qquad \frac{\Gamma \vdash^{t} ty \rightsquigarrow \sigma \qquad \Gamma \vdash^{gen}_{\Downarrow} t \rightsquigarrow t' : \sigma \qquad \Gamma \vdash^{inst}_{\delta} \sigma \leq \tau \rightsquigarrow \overline{\upsilon}}{\Gamma \vdash_{\delta} (t\texttt{::}ty) \rightsquigarrow t'\, \overline{\upsilon} : \tau}\ \textsc{Annot}$$

$$\frac{\Gamma \vdash^{gen}_{\Uparrow} u \rightsquigarrow u' : \sigma \qquad (\Gamma, x : \sigma) \vdash_{\delta} t \rightsquigarrow t' : \tau}{\Gamma \vdash_{\delta} (\texttt{let } x\texttt{=}u \texttt{ in } t) \rightsquigarrow (\texttt{let } x_{\mathcal{S}(\sigma)}\texttt{=}u' \texttt{ in } t') : \tau}\ \textsc{Let} \qquad \frac{\Gamma \vdash^{t} ty \rightsquigarrow \sigma \qquad \Gamma, x : \sigma \vdash^{gen}_{\Downarrow} u \rightsquigarrow u' : \sigma \qquad \Gamma, x : \sigma \vdash_{\delta} t \rightsquigarrow t' : \tau}{\Gamma \vdash_{\delta} (\texttt{letrec } x\texttt{::}ty\texttt{=}u \texttt{ in } t) \rightsquigarrow (\texttt{let } x_{\mathcal{S}(\sigma)}\texttt{=}u' \texttt{ in } t') : \tau}\ \textsc{Rec}$$

$$\frac{\Gamma \vdash^{k} \tau_1 \qquad \Gamma \vdash^{a}_{\Uparrow} p\texttt{->}t \rightsquigarrow p'\texttt{->}t' : \boxed{\tau_1} \to \tau_2}{\Gamma \vdash_{\Uparrow} \lambda p.t \rightsquigarrow \lambda p'.t' : \boxed{\tau_1} \to \tau_2}\ \textsc{Abs}\Uparrow \qquad \frac{\Gamma \vdash^{a}_{\Downarrow} p\texttt{->}t \rightsquigarrow p'\texttt{->}t' : \tau_1 \to \tau_2}{\Gamma \vdash_{\Downarrow} \lambda p.t \rightsquigarrow \lambda p'.t' : \tau_1 \to \tau_2}\ \textsc{Abs}\Downarrow$$

$$\frac{\Gamma \vdash_{\Uparrow} u \rightsquigarrow u' : \tau_1 \qquad \Gamma \vdash^{a}_{\delta} \overline{p\texttt{->}t \rightsquigarrow p'\texttt{->}t' : \tau_1 \to \tau_2}}{\Gamma \vdash_{\delta} (\texttt{case } u \texttt{ of } \overline{p\texttt{->}t}) \rightsquigarrow (\texttt{case}(\tau_2)\ u' \texttt{ of } \overline{p'\texttt{->}t'}) : \tau_2}\ \textsc{Case} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\Uparrow} t \rightsquigarrow t' : \tau \qquad push(\tau) = \tau_1 \to \tau_2 \\ \Gamma \vdash_{\Downarrow} u \rightsquigarrow u' : \tau_1 \qquad \vdash_{\delta} \tau_2 \sim \tau_2'\end{array}}{\Gamma \vdash_{\delta} t\, u \rightsquigarrow t'\, u' : \tau_2'}\ \textsc{App}$$

$$\boxed{\Gamma \vdash^{gen}_{\delta} t \rightsquigarrow t' : \sigma} \qquad \boxed{\Gamma \vdash^{inst}_{\delta} \sigma \leq \tau \rightsquigarrow \overline{\upsilon}} \qquad \boxed{\vdash_{\delta} \tau_1 \sim \tau_2}$$

$$\frac{\Gamma, \overline{\alpha} \vdash_{\delta} t \rightsquigarrow t' : \tau \qquad \overline{\alpha} \# dom(\Gamma)}{\Gamma \vdash^{gen}_{\delta} t \rightsquigarrow \Lambda \overline{\alpha}\,.\,t' : \forall \overline{\alpha}.\tau}\ \textsc{Gen} \qquad \frac{\Gamma \vdash^{k} \overline{\upsilon} \qquad \vdash_{\delta} [\,\overline{\alpha \mapsto \boxed{\upsilon}}\,]\tau_1 \sim \tau_2}{\Gamma \vdash^{inst}_{\delta} \forall \overline{\alpha}.\tau_1 \ \leq\ \tau_2 \rightsquigarrow \overline{\mathcal{S}(\upsilon)}}\ \textsc{Inst} \qquad \frac{}{\vdash_{\Uparrow} \tau \sim \tau}\ \textsc{Inst}\Uparrow \qquad \frac{\mathcal{S}(\tau_1) = \mathcal{S}(\tau_2)}{\vdash_{\Downarrow} \tau_1 \sim \tau_2}\ \textsc{Inst}\Downarrow$$

$$\boxed{\text{Case alternatives }\ \Gamma \vdash^{a}_{\delta} p\texttt{->}u \rightsquigarrow p'\texttt{->}u' : \tau_1 \to \tau_2}$$

$$\frac{\Gamma; \epsilon; \emptyset \vdash^{p} p \rightsquigarrow p' : \tau_1;\ \Delta;\ \theta \qquad dom(\Delta) \# (ftv(\tau_2)) \qquad \theta(\Gamma, \Delta) \vdash_{\delta} u \rightsquigarrow u' : \theta_{\delta}(\tau_2)}{\Gamma \vdash^{a}_{\delta} p\texttt{->}u \rightsquigarrow p'\texttt{->}u' : \tau_1 \to \tau_2}\ \textsc{Alt}$$

$$\boxed{\text{Patterns }\ \Gamma; \Delta; \theta \vdash^{p} p \rightsquigarrow p' : \tau; \Delta'; \theta'} \qquad \boxed{\text{Unification }\ \vdash^{u} \Pi \rightsquigarrow \theta}$$

$$\frac{x \# dom(\Delta)}{\Gamma; \Delta; \theta \vdash^{p} x \rightsquigarrow x_{\mathcal{S}(\tau)} : \tau;\ \Delta, (x : \tau);\ \theta}\ \textsc{PVar} \qquad \frac{\begin{array}{c}\theta(\Pi') = \Pi \qquad dom(\theta) \# ftv(\Pi) \\ \Pi' \text{ is rigid} \qquad \theta' \text{ is a most general unifier of } \Pi'\end{array}}{\vdash^{u} \Pi \rightsquigarrow (\theta \circ \theta')\,|_{ftv(\Pi)}}\ \textsc{Unif}$$

$$\frac{\begin{array}{c}(C : \forall \overline{\alpha}\,.\,\overline{\tau}^{c} \to T\, \overline{\upsilon}^{t}) \in \Gamma \qquad \overline{\alpha} \# dom(\Gamma, \Delta) \\ push(\theta(\upsilon')) = T\, \overline{\upsilon''}^{t} \qquad \vdash^{u} (T\, \overline{\upsilon}^{t} \doteq T\, \overline{\upsilon''}^{t}) \rightsquigarrow \theta' \\ \Gamma; \Delta, \overline{\alpha}; (\theta' \circ \theta) \vdash^{ps} \overline{p : \tau}^{c} \rightsquigarrow \overline{p'}^{c}; \Delta''; \theta''\end{array}}{\Gamma; \Delta; \theta \vdash^{p} C\, \overline{p}^{c} \rightsquigarrow C\, \overline{\alpha}\, \overline{p'}^{c} : \upsilon'; \Delta''; \theta''}\ \textsc{PCon}$$

$$\boxed{\text{Pattern sequences }\ \Gamma; \Delta; \theta \vdash^{ps} \overline{p : \tau} \rightsquigarrow \overline{p'}; \Delta'; \theta'}$$

$$\frac{}{\Gamma; \Delta; \theta \vdash^{ps} \epsilon \rightsquigarrow \epsilon; \Delta; \theta}\ \textsc{PEmpty} \qquad \frac{\Gamma; \Delta; \theta \vdash^{p} p_1 : \tau_1 \rightsquigarrow p_1';\ \Delta';\ \theta' \qquad \Gamma; \Delta'; \theta' \vdash^{ps} \overline{p : \tau} \rightsquigarrow \overline{p'};\ \Delta'';\ \theta''}{\Gamma; \Delta; \theta \vdash^{ps} p_1 : \tau_1, \overline{p : \tau} \rightsquigarrow p_1', \overline{p'};\ \Delta'';\ \theta''}\ \textsc{PCons}$$

Figure 6: Typing rules for the source language

rule REC uses checking mode to support polymorphic recursion.. The rule for application (APP), is a little unusual, however. What one normally sees is something like this:

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\,u : \tau_2}$$

However, in our system, it is possible that the function t will turn out to have a wobbly type, or perhaps a doubly-wobbly type, or more. What we need is that it can be cajoled into the form $\tau_1 \to \tau_2$, for some types τ_1 and τ_2, a predicate we capture in the partial function $push(\tau)$ (see Figure 5). This function takes a possibly-wobbly type, and guarantees to return a type with a type constructor (such as a function arrow) at the outermost level. The function $push$ is partial; for example, $push(\alpha)$ gives no result, and the APP rule fails. Now we can check that u has the desired type, with $\vdash_{\Downarrow}$. Finally, we must check that the function's result type τ_2 matches the expected result τ_2', just as we did in rule ATOM.

Rule CASE handles `case` expressions. First it derives the type τ_1 of the scrutinee u, and then it uses the auxiliary judgement $\vdash^{a}$ to type-check the alternatives. Since a lambda abstraction binds a pattern, it behaves very like a single case alternative, so rules ABS$\Uparrow$ and ABS$\Downarrow$ simply invoke the same auxiliary judgement as for `case` expressions, the former with a wobbly type as discussed in Section 4.3.

The judgements $\vdash^{a}$, for alternatives, and its auxiliary judgement $\vdash^{p}$ for patterns, take a very similar form to those of the explicitly-typed language (Section 3). One significant difference is the use of $push$ in rule PCON, which appears for exactly the same reason as in rule APP. Another important difference, also in PCON, is that we invoke an auxiliary judgement $\vdash^{u} \Pi \leadsto \theta$ to unify Π; we discuss this judgement in Section 4.5. The third major difference is that in rule ALT we apply θ_δ to τ_2, rather than θ (Figure 4 defines θ_δ). The idea is that in guessing mode, we cannot expect to guess a single type that can be refined for each alternative by that alternative's substitution.

4.5 Type refinement with wobbly types

Rule PCON uses an auxiliary judgement $\vdash^{u} \Pi \leadsto \theta$ to unify the constraints Π. The key idea is that this judgement performs a kind of partial unification of Π, *based only on information in* Π *outside wobbly types*. This simple intuition is surprisingly tricky to formalise.

Rule UNIF in Figure 6 gives the details. It splits Π into two parts: a rigid Π', in which arbitrary types within Π have been replaced with fresh type variables $\overline{\gamma}$; and a substitution θ that maps the $\overline{\gamma}$ onto the excised types. These two parts fit together, so that $\theta(\Pi')$ is the original Π. Now we find a most general unifier of Π', namely θ'. Finally, we re-insert the excised types into θ', by composing it with θ, and restricting the domain of the resulting substitution to the free type variables of Π — or, equivalently, discarding any substitutions for the intermediate $\overline{\gamma}$. (The notation $\theta|_{\overline{\alpha}}$ means the substitution θ with domain restricted to $\overline{\alpha}$.)

The judgement leaves a lot of freedom about how to split Π into pieces, but a complete type inference algorithm must find a solution if one exists. The way to do this is to split Π by replacing the outermost wobbly types with fresh type variables, so that as much rigid information as possible is exposed in Π'. This algorithm is quite discerning. For example, the reader is invited to verify that, using this algorithm,

$$\begin{array}{ll}\vdash^{u} & (\alpha \doteq (\boxed{\mathtt{Int}},\mathtt{Bool}), \alpha \doteq (\mathtt{Int},\boxed{\mathtt{Bool}})) \\ & \leadsto [\alpha \mapsto (\mathtt{Int},\mathtt{Bool})]\end{array}$$

where the result has gathered together the least-wobbly result that is justified by the evidence.

Notice that the substitution returned by $\vdash^{u}$ is not necessarily a unifier of the constraints. For example, here is a valid judgement:

$$\vdash^{u} ((\alpha,\beta) \doteq \boxed{(\mathtt{Int},\mathtt{Bool})}) \leadsto \emptyset$$

where the empty substitution $\emptyset$ plainly is not a unifier of the input constraints. This amounts to saying that the algorithm may do less type refinement than would be justified in an explicitly-typed program. Furthermore, $\vdash^{u}$ may succeed even when there *is* no unifier of the stripped constraints. For example,

$$\vdash^{u} ((\alpha,\mathtt{Int}) \doteq (\mathtt{Int},\boxed{\mathtt{Bool}})) \leadsto [\alpha \mapsto \mathtt{Int}]$$

This means that the algorithm may not detect all inaccessible branches of a `case`; which is perfectly sound, because such branches are admitted in the core language and need not be well-typed. Another variant of the same sort is $\vdash^{u} (\alpha \doteq \boxed{\alpha \to \mathtt{Int}}) \leadsto [\alpha \doteq \boxed{\alpha \to \mathtt{Int}}]$. Again, the stripped version would be rejected (by the occurs check), but the source-language type system may not detect the inaccessible branch.

A delicate point of rule UNIF is that we are careful to say that "θ' is a most general unifier of Π'" rather than "$\theta' = \mathcal{U}(\Pi')$", because it sometimes makes a difference which way around the unifier chooses to solve trivial constraints of form $\alpha \doteq \beta$. For example, consider this simple Haskell 98 program:

```
  f = \x. case x of { Just y -> y }
```

One valid typing derivation gives `x` the wobbly type $\boxed{\mathtt{Maybe}\ \alpha}$. Now we apply rule PCON to the `case` pattern: the `Maybe` constructor will bind a fresh type variable, say β, and we try to find θ'' such that $\vdash (\mathtt{Maybe}\ \beta \doteq \mathtt{Maybe}\ \boxed{\alpha}) \leadsto \theta''$. To do this we split the constraints into $\Pi' = (\mathtt{Maybe}\ \beta \doteq \mathtt{Maybe}\ \gamma)$, and $\theta = [\gamma \mapsto \boxed{\alpha}]$. Now here is the subtlety: there are two possible most general unifiers θ', namely

$[\beta \mapsto \gamma]$ and $[\gamma \mapsto \beta]$. The former produces the desired result $\theta'' = (\theta \circ \theta')|_{\alpha,\beta} = [\beta \mapsto \boxed{\alpha}]$. But the latter produces $\theta'' = [\gamma \mapsto \beta]|_{\alpha,\beta} = \emptyset$ which fails to type the program.

Although this is tricky stuff, an inference algorithm can easily find a "best" most general unifier: *whenever there is a choice, the unification algorithm should substitute for the type variables bound by the pattern*. Specifically, when solving a trivial constraint $\alpha \doteq \beta$, where β is bound by the pattern and α is not, the unifier should substitute $[\beta \mapsto \alpha]$; and vice versa. The intuition is that these pattern-bound type variables only scope over a single case branch, so we should do all we can to refine them into more globally-valid types.

A minor difference from Section 3 is that we treat unification failure as a type error, because rule UNIF simply fails if Π' has no unifier. This saves two rules, and makes more sense from a programming point of view, but loses subject reduction. We do not mind the latter, because we treat soundness by giving a type-directed translation into our core language, rather than by giving a dynamic semantics for the source language directly.

4.6 Smart function application

The rules we have presented will type many programs, but there are still some unexpected failures. Here is an example (c.f. [BS02]):

```
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 a1 b1) (RP a2 b2)
  = case test a1 a2 of
      Nothing -> Nothing
      Just Eq -> case test b1 b2 of
                   Nothing -> Nothing
                   Just Eq -> Eq
```

A non-bottom value `Eq` of type `Equal t1 t2` is a witness that the types `t1` and `t2` are the same; that is why the constructor has type $\forall\alpha\,.\,\texttt{Equal}\ \alpha\ \alpha$. The difficulty with typing this comes from the guessing involved in function instantiation (rule INST). Even if we know that $x : \texttt{Int}$, say, the term $(\texttt{id}\ x)$, where `id` is the identity function, will have type $\boxed{\texttt{Int}}$. In `test`, therefore, no useful information is exposed to type refinement in the `case` expressions, because both scrutinise the result of a call to a polymorphic function (`test` itself), which will be instantiated with wobbly types.

This is readily fixed by treating a function application as a whole, thus:

$$\frac{\begin{array}{cc} (f : \forall\overline{\alpha}\,.\,\overline{\tau} \to \upsilon) \in \Gamma & \Gamma \vdash_{\Uparrow} \overline{t : \tau'} \\ \Gamma \vdash^{m} \overline{\alpha}; (\overline{\tau \doteq \tau'}) \leadsto \theta & \vdash_{\delta} \theta(\upsilon) \sim \upsilon' \end{array}}{\Gamma \vdash_{\delta} f\ \overline{t} : \upsilon'}\ \text{APPN}$$

The idea is that we guess (rather than check) the arguments, obtaining types $\overline{\tau'}$ that express our certainty (or otherwise) about these types. Then we use a new judgement $\vdash^{m}$ to match the function's expected argument types $\overline{\tau}$ against the actual types. Finally, we check that the result types match up.

The judgement $\Gamma \vdash^{m} \overline{\alpha}; \Pi \leadsto \theta$ finds an instantiation of the type variables $\overline{\alpha}$ that satisfies Π. To maximise information, we would like it to find the least-wobbly instantiation that can, something that our existing unification judgement $\vdash^{u}$ also does:

$$\frac{\begin{array}{c} \vdash^{u} \Pi \leadsto \theta \qquad \Gamma \vdash^{k} \overline{\upsilon} \qquad \theta' = [\,\overline{\alpha \mapsto \boxed{\upsilon}}\,] \circ \theta|_{\overline{\alpha}} \\ \forall \tau \doteq \tau' \in \Pi,\ \mathcal{S}(\theta'(\tau)) = \mathcal{S}(\tau') \end{array}}{\Gamma \vdash^{m} \overline{\alpha}; \Pi \leadsto \theta'}\ \text{MATCH}$$

We use $\vdash^{u}$ to solve Π, then restrict the domain of the result, θ, because we want a one-way match only: the function type must not influence the environment. So $\theta|_{\overline{\alpha}}$ is a substitution that binds some, but perhaps not all, of the $\overline{\alpha}$. We expand this substitution to instantiate the remaining $\overline{\alpha}$ with guessed types $\overline{\boxed{\upsilon}}$, yielding θ'. Then we check that θ' does indeed satisfy the constraints, modulo wobbliness.

There is a strong duality between rules PCON and APPN: in patterns, the type refinement is derived from the result type, and is applied to the arguments; in function applications, the type refinement is derived from the argument types, and is applied to the result.

4.7 Lexically-scoped type variables

Thus far, all our type annotations have been closed. But in practice, as programs scale up, it is essential to be able to write open type annotations; that is, ones that mention type variables that are already in scope. In the olden days, when type annotations were optional documentation, open type signatures were desirable but not essential, but now that type annotations are sometimes mandatory, it must be possible to write them.

We therefore add lexically scoped type variables as an orthogonal addition to the language we have presented so far. The extensions are these. First, in a type-annotated term $(t\!::\!ty)$, the source type ty may be open. Second, the environment Γ is extended with a binding form, $a = \tau$, which binds a source type variable a to a type τ. These bindings are used in the obvious way when kind-checking an open source type ty. Third, patterns are extended with a new form

$$p \quad ::= \quad \ldots \mid (p\!::\!\overline{a}\,.\,ty)$$

This type-annotated pattern brings into scope the source type variables $\overline{a}$, with the same scope as the term variables bound by p, and ensures that p has type ty. Here is the extra rule for the pattern judgement:

$$\frac{\begin{array}{c}\Gamma,\Delta \vdash^{t} \texttt{forall}\,\overline{a}\,.\,ty \rightsquigarrow \forall\overline{\alpha}\,.\,\tau \\ \Gamma \vdash^{m} \overline{\alpha};\theta(\tau) \doteq \theta(\tau') \rightsquigarrow \theta' \\ \Delta' = \Delta, \overline{a = \theta'(\alpha)} \\ \Gamma;\Delta';\theta \vdash^{p} p \rightsquigarrow p' : \theta'(\theta(\tau));\Delta'';\theta''\end{array}}{\Gamma;\Delta;\theta \vdash^{p} (p\texttt{::}\overline{a}\,.\,ty) \rightsquigarrow p' : \tau';\Delta'';\theta''}\ \text{PANNOT}$$

We begin by kind-checking the source type, ty, temporarily building a `forall`'d source type so that $\vdash^{t}$ will generate a polymorphic type $\forall\overline{\alpha}\,.\,\tau$. Then we use the same judgement $\vdash^{m}$ that we introduced in Section 4.6 to instantiate this type to match the incoming type τ', being careful to first apply the current type refinement. Finally we check the nested pattern p, in the extended type environment. A lexically-scoped type variable scopes over all patterns to the right, so that the pattern `(x::a.Term a, y::a)` makes sense. That is why we "thread" the environment Δ through the pattern judgements (c.f. Section 3.2).

Notice that a lexically-scoped type variable is simply a name for an (internal) type, not necessarily a type variable. For example, the term `(\(x::a.a). x && x)` is perfectly acceptable: the source type variable `a` is bound to the type `Bool`. More precisely, it is bound to the type $\boxed{\texttt{Bool}}$, because a scoped type variable maybe be bound to a wobbly type, and the type system indeed says that `x`'s type will be wobbly in this case.

This means that pattern type signatures cannot be used to specify a completely rigid *polymorphic* type, which is a slight pity. For example, if we write

```
eval = \(x::a.Term a). (...body... :: a)
```

the type of `x` will be `Term` $\boxed{\alpha}$, to reflect the uncertainty about what type `a` will ultimately be bound to, and hence no type refinement will take place, and the definition will be rejected (at least if `..body..` requires type refinement). The only way to give a completely rigid polymorphic type is using a type signature on a term, or on a `letrec` binding:

```
eval :: forall a. Term a -> a = \x. ...body...
```

4.8 Scope and implicit quantification

A notationally awkward feature of the design we describe is the "$\overline{a}$." prefix on a pattern type signature, which brings the type variables $\overline{a}$ into scope. In our real source language, we use an implicit-quantification rule that allows us to write, for example

```
eval = \(x::Term a). ...etc...
```

with no "`a.`" prefix. The rule we use is this: any type variable that is mentioned in the pattern type annotation, and is not already in scope, is brought into scope by the pattern. This is the same rule that we use for adding implicit `forall` quantifiers to type signatures on terms. One could imagine other choices, but it is an orthogonal concern to this paper.

Haskell allows separate, declaration type signatures, thus:

```
eval :: Term a -> a
eval = ...
```

It is (arguably) attractive to allow the universally-quantified type variables of such a signature to scope, lexically, over the body of `eval` [MC97, SSW04]. Again this is an orthogonal concern, but one that readily works with our main design.

4.9 Properties of the type system

Our system is sound, in the sense that any well-typed program translates to a well-typed core-language program:

THEOREM 4.1. *If* $\Gamma \vdash_{\delta} t \rightsquigarrow t' : \tau$ *then* $\mathcal{S}(\Gamma) \vdash t' : \mathcal{S}(\tau)$

We have proved this theorem for the system of Figure 6, augmented with the smart function-application rule (Section 4.6) and lexically-scoped type variables (Section 4.7). The main tricky point in the proof is to show that the partial refinement generated by $\vdash^{u}$, if it succeeds, yields a well-typed core program. The key property is this:

LEMMA 4.2. *If* $\vdash^{u} \Pi \rightsquigarrow \theta$ *then either* $\mathcal{S}(\Pi)$ *has no unifier, or* $\mathcal{S}(\Pi)$ *has a most general unifier* θ' *such that*
$\theta' = \theta' \circ \mathcal{S}(\theta)$.

We say that $\mathcal{S}(\theta)$ is a *pre-unifier* of $\mathcal{S}(\Pi)$: it is not necessarily a unifier, but it is "on the way" to one (if one exists at all).

Furthermore, our system is a conservative extension of vanilla Haskell/ML data types. The latter have types of form $\forall\overline{\alpha}\,.\,\overline{\tau} \to T\,\overline{\alpha}$, where the $\overline{\alpha}$ are distinct. Hence rule UNIF is guaranteed to succeed, and one possible solution is always of form $[\,\overline{\alpha \mapsto \tau}\,]$, where the pattern has type $T\,\overline{\tau}$. This substitution binds only the pattern-bound type variables (i.e. does not refine the rest of the environment) and ensures that the sub-patterns have exactly the expected types. It would be straightforward, albeit tedious, to formalise this argument.

5 Implementation

We have built a prototype type inference engine for the source-language type system, starting from the executable prototype described by [PS04]. This baseline algorithm is quite conventional; most of the work is done by a unifier that implements an ever-growing substitution using side effects. "Guessed" types are implemented by "flexible" meta variables, which are fleshed out by in-place updates performed by the unifier. There is no constraint gathering; in effect, the equality constraints are solved incrementally, as they are encountered.

By design, it is quite easy to support our new type system. Some straightforward extensions are required to parse the new data type declarations, and to extend Γ with the constructors they define. Wobbly types are simple to implement: they simply *are* the flexible meta variables that the inference engine already uses, and introduction of a wobbly type in the rules (e.g. in $\vdash^{inst}$) corresponds to the allocation of a fresh flexible meta variable. Invocations of push in the rules correspond to places where the inference algorithm must force a type to have a certain outermost shape (e.g. be of form $\tau_1 \rightarrow \tau_2$), which sometimes requires the allocation of further flexible meta variables. One also has to take care that the commonly-implemented path-compression optimisation, which elminates chains of flexible meta variables, does not thereby elide the wobbliness altogether.

Match-unification implements the wobbly-type-aware algorithm of Section 4.5, and is implemented entirely separately from inference-unification. Different type refinements apply in different case branches, so in-place update is inappropriate, and the match-unification algorithm instead generates a data structure representing the type refinement explicitly. Rather than applying the type refinement eagerly to the environment, as the rules do, we perform this substitution lazily, by carrying down a pair of the type environemt and the current refinement. The inference-unifier consults (but does not extend) the type refinement during unification. One wrinkle that we missed at first is that the unifier must also consult the type refinement when performing the occurs check. There are also some occasions where we must eagerly apply the type refinement to an entire type, such as when finding the free variables of a type at a generalisation point.

On the basis of this experiment, the changes to the inference algorithm do indeed appear to be extremely localised and non-invasive, as we hoped. The only apparently-global change is the requirement to pair up the type refinement with the environment, but the monadic framework we use makes this change local as well.

6 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* [Dyb91]. Coquand in his work on dependently typed pattern matching [Coq92] also uses a unification based mechanism for implementing the refinement of knowledge gained through pattern matching. These ideas were incoporated in the ALF proof editor [Mag94], and have evolved into dependently-typed programming languages such as Cayenne [Aug98] and Epigram [MM04]. 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.

Xi, et al.'s work on guarded recursive data types closely corresponds to our work. They present a language very similar to our core language [XCC03], though with a bug that prevents them from checking some fairly useful classes of nested patterns [SP03, Remark 4.27]. Instead of using unification in the specification of their type system, type-equality constraints are propagated around the typing rules and solved as needed. Their type inference algorithm, like ours, is based upon Pierce and Turner's local type inference [PT98]. Also closely related is Zenger's system of indexed types [Zen97], and Xi's language Dependent ML [XP99]; in both cases, instead of constraints over type equalities, the constraint language is enriched to include, for example, Presburger arithmetic . Finally, Xi generalises both these languages with what he calls an *applied type system* [Xi04].

Cheney and Hinze examine numerous uses of what they call *first class phantom types* [CH03, Hin03]. Their language is essentially equivalent to ours in terms of expressiveness, but they achieve type refinement via equality constraint clauses. Sheard and Pasalic use a similar design they call *equality-qualified types* in the language Ωmega [SP04].

Most of this work concerns type checking for GADTs, but Simonet and Pottier explicitly tackle type inference [SP03]. Their work is much more general than ours: they start from the HM(X) constraint framework, and generalise it to a language in which arbitrary constraints can be used to guard quantification. Our language corresponds to instantiating theirs with type equality constraints, and exploiting this special case seems to make the system considerably simpler. In their search for tractable inference, they are forced to impose two undesirable restrictions: type annotations must be closed, and their system embodies two independent rule sets, one dealing with GADTs and the other with ordinary data types. Our system manages to avoid both these shortcomings; it would be interesting to get a more detailed insight into these trade-offs, perhaps by expressing our solution in their framework.

Our wobbly types correspond very closely to meta-variables in an implementation of type inference. Nanevski, Pientka, and Pfenning have developed an explicit account of meta-variables in terms of a modal type system [NPP03]. It would be worthwhile to examine whether their language can subsume our wobbly types. Our wobbly types also propagate uncertainty in a fashion that has the flavour of coloured types in Odersky and Zenger's coloured local type inference [OZZ01].

7 Conclusion and further work

We have much left to do. In this paper we have not formally presented an inference algorithm and proved it complete with respect to our specification. Our claim that wobbly types accurately reflect the uncertainty of real algorithms is, therefore, not formally established. We have built a prototype implementation, however, and we are confident that a formal proof is within reach. Similarly, our claim that wobbly types can co-exist smoothly with the other complexities of Haskell's type system is rooted in the authors' (rather detailed) experience of implementing the latter. We are engaged in a full-scale implementation in GHC, which will provide concrete evidence.

Nevertheless, we believe that this paper takes a significant step forward. The literature on technical aspects of GADTs is not easy going, and the system of Section 3 is the simplest we have seen. There is very little literature on type inference for GADTs, and ours is uniquely powerful. More generally, we believe that type systems will increasingly embody a blend of type inference and programmer-supplied type annotations: polymorphic recursion, higher-rank types, and GADTs, are all examples of this trend, and there are plenty of others (e.g. sub-typing). Giving a precise, predictable, and implementable specification of these blended type systems is a new challenge. Bidirectional type inference is one powerful tool, and we believe that wobbly types, our main contribution, are another.

Acknowlegements. We gratefully acknowledge generous feedback from James Cheney, Matthew Fluet, Ralf Hinze, Barry Jay, Simon Marlow, Conor McBride, Greg Morrisett, Tim Sheard, Martin Sulzmann, and Don Syme. We are particularly grateful to François Pottier for his detailed insights.

8 References

[Aug98] 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.

[BS02] 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.

[CH03] James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.

[Coq92] T Coquand. Pattern matching with dependent types. In *Proc workshop on Logical Frameworks, Bastaad*, pages 66–79, 1992.

[Dyb91] Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*. CUP, 1991.

[Hin03] Ralf Hinze. Fun with phantom types. In Jeremey Gibbons and Oege de Moor, editors, *The fun of programming*, pages 245–262. Palgrave, 2003.

[Jay03] Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems*, To appear, 2003.

[Mag94] L Magnusson. *The implementation of ALF - a proof editor based on Martin-Löf's monomorhic type theory with explicit substitution.* PhD thesis, Chalmers University, 1994.

[MC97] E Meijer and K Claessen. The design and implementation of Mondrian. In J Launchbury, editor, *Haskell workshop*, Amsterdam, 1997.

[MM04] C McBride and J McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[NPP03] Aleksandar Nanevski, Brigitte Pientka, and Frank Pfenning. A modal foundation for meta-variables. In *Proceedings of MERLIN'03, Uppsala, Sweden*, pages 159–170, 2003.

[OZZ01] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, London, January 2001. ACM.

[PS04] Simon Peyton Jones and Mark Shields. Practical type inference for higher-rank types. Unpublished manuscript, 2004.

[PT98] Benjamin C. Pierce and David N. Turner. Local type inference. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 252–265, San Diego, January 1998. ACM.

[Rob65] JA Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965.

[She04] Tim Sheard. Languages of the future. In *ACM Conference on Object Orientated Programming Systems, Languages and Applicatioons (OOPSLA'04)*, 2004.

[SP03] Vincent Simonet and François Pottier. Constraint-based type inference with guarded algebraic data types. Technical report, Inria, July 2003.

[SP04] 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.

[SSW04] Peter Stuckey, Martin Sulzmann, and Jeremy Wazny. Type annotations in Haskell. Technical report, National University of Singapore, 204.

[XCC03] 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.

[Xi04] Hongwei Xi. Applied type system. In *Proceedings of TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer Verlag, 2004.

[XP99] 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.

[Zen97] C Zenger. Indexed types. *Theoretical Computer Science*, pages 147–165, 1997.