Let Should Not Be Generalised

Dimitrios Vytiniotis
Microsoft Research Cambridge, UK
dimitris@microsoft.com

Simon Peyton Jones
Microsoft Research Cambridge, UK
simonpj@microsoft.com

Tom Schrijvers [*]
Katholieke Universiteit Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Abstract

From the dawn of time, all derivatives of the classic Hindley-Milner type system have supported implicit generalisation of local `let`-bindings. Yet, as we will show, for more sophisticated type systems implicit `let`-generalisation imposes a disproportionate complexity burden. Moreover, it turns out that the feature is very seldom used, so we propose to eliminate it. The payoff is a substantial simplification, both of the specification of the type system, and of its implementation.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Functional Languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type Structure

General Terms Algorithms, Languages

Keywords Haskell, type inference, type families, type classes, generalized algebraic data types

1. Introduction

Academic papers about type systems usually propose to *add* a new feature to an existing type system. This paper is different: we propose to *remove* a feature.

From its inception, a central feature of the influential Hindley-Milner type system (Mil78; DM82) was that `let`-bindings are generalised. For example, consider

```
wuggle x = let singleton y = [y]
           in (singleton True, singleton 'w')
```

Here the locally-bound function `singleton` is implicitly generalised, to $\forall a.a \rightarrow$ `[`a`]`. As a result, `singleton` can be called at two different types in the body of the `let`.

Although generalisation of `let` bindings is taken for granted by any red-blooded functional programmer, this paper argues for its abolition, at least for local bindings. We make the following contributions:

[*] Post-doctoral researcher of the Fund for Scientific Research - Flanders.

- While generalisation for local `let` bindings is straightforward in Hindley-Milner, we show that it becomes much more complicated in more sophisticated type systems. The extensions we have in mind include functional dependencies (Jon00), GADTs (XCC03), units of measure (Ken96), and type functions (CKPM05; CKP05) (Section 2). The only technically-straightforward way to combine these developments with `let`-generalisation has unpalatable practical consequences (Section 2.2), that appear to be unavoidable (Section 3).
- We show that generalisation for local `let` bindings is seldom used; that is, if they are never generalised, few programs fail to typecheck (Section 4). We base this claim on compiling hundreds of public-domain Haskell packages, containing hundreds of thousands lines of code. Furthermore, those programs that do fail are readily fixed by adding a type signature.
- Rather than fix on a particular set of "sophisticated extensions", we present LHM(X), a type system with qualified types, parameterised over a constraint domain X (Section 5). Thus LHM(X) stands in the tradition of HM(X) (OSW99), and Jones's OML (Jon92), but unlike HM(X) we deal with *local assumptions* introduced by pattern matching and type signatures, and unlike OML we deal with both *type equalities* and local assumptions .
- We present type inference infrastructure that deals with local assumptions and is parameterized over a constraint solver for the constraint domain X. We show that LHM(X) enjoys sound and efficient type inference, conditional on intuitive requirements from the X theory solver (Section 5). Unsurprisingly, completeness does not hold due to the *ambiguity* problem, a recurring issue in type systems for Haskell. Though we do not formally show completeness, we propose a novel way to address ambiguity (Section 5.6) and recover completeness.

In short, we argue that we should simply abandon generalisation for local `let`s entirely, thereby providing a substantial simplification at almost no cost to the programmer. The aforementioned modification only applies to *local*, or nested bindings (such as `singleton`), and not for *top-level* bindings (such as `wuggle`). In the case of top-level bindings, we can freely generalize as before, as Section 5 shows. Moreover, we still support for polymorphic local bindings by supplying explicit type signature. Only *local* bindings with *inferred types* are affected.

This simplification enables this paper to take a significant step forward compared to our two earlier works in this area. In (SPJCS08) we gave a solver for equalities (involving type functions), that we "lift" in this paper to a solver than handles implication constraints — essential for type inference with local assumptions. In (SJSV09) we proposed the **OutsideIn** approach, which deals with implication constraints, but only in the special case of GADTs. Here we generalize to an arbitrary constraint domain X, which turns out to be decidedly non-trivial.

2. Motivation

The Hindley-Milner type system is a masterpiece of design. It offered a big step forward in expressiveness (parametric polymorphism) at very low cost. The cost is low in several dimensions: the type system is technically easy to describe, and an inference algorithm is both sound and complete with respect to the specification. And it does all this for programs with no type annotations at all!

A central feature of the Hindley-Milner system is that `let`-bound definitions are *generalised*. For example, consider the slightly artificial definition

```
f x = let g y = (x,y) in ...
```

The definition for g is typed in an environment in which x :: a, and the inferred type for g is $\forall b.b \rightarrow (a, b)$. This type is polymorphic in b, but not in a, because the latter is free in the type environment at the definition of g. This side condition, that g should be generalised only over variables that are not free in the type environment, is the only tricky point in the entire Hindley-Milner type system.

Type systems advanced rapidly in the following 25 years, both in expressiveness and (less happily) in complexity. One particular way in which they have advanced is in the form of *constraints* that they admit:

- The base Hindley-Milner system admits just one constraint, namely the equality of two types, which we write $\tau_1 \sim \tau_2$. For example, the application of a function of type $\tau_1 \rightarrow \tau_2$ to an argument of type τ_3 gives rise to an equality constraint $\tau_1 \sim \tau_3$.
- Haskell's type classes add *class constraints* (WB89; HHPW96). For example, the constraint `Eq` τ requires that the type τ be an instance of the class `Eq`. Haskell goes further, and allows *abstraction* over constraints. For example the `member` function has type `member :: Eq a => a -> [a] -> Bool`, which says that `member` may be called at any type τ, but that the constraint `Eq` τ must be satisfied at the call site.
- Mark Jones extended multi-parameter type classes with *functional dependencies* in 2001 (Jon00). This feature turned out to be tremendously useful in practice, and gave rise to a whole cottage industry of programming techniques that amounts to programming arbitrary computation at the type level. We omit the technical details here but the underlying idea was that conjunction of two class constraints $C\ \tau\ \upsilon_1$ and $C\ \tau\ \upsilon_2$ gives rise to an additional equality constraint $\upsilon_1 \sim \upsilon_2$.
- Generalised Algebraic Data Types (GADTs) added a new twist to equality constraints by supporting *local* equalities (XCC03; PVWW06). We will discuss GADTs further in Section 2.1.
- Kennedy's thesis (Ken96) describes how to accommodate *units of measure* in the type system so that one may write

  ```
  calcDistance :: num (m/s) -> num s -> num m
  calcDistance speed time = speed * time
  ```

  thereby ensuring that the first argument is a speed in metres/second, and similarly for the other argument and result. The system supports polymorphism, for example

  ```
  (*) :: num u1 -> num u2 -> num (u1*u2)
  ```

  There is, necessarily, a non-structural notion of type equality. For example, to typecheck the definition of `calcDistance` the type engine must reason that `(m/s)*s` $\sim$ `m` This is an ordinary equality constraint, but there is now a non-standard equality theory so the solver becomes more complicated.
- More recently, inspired by the notion of associated types in object-oriented languages, we have proposed and implemented a similar notion in Haskell (CKPM05; CKP05). The core feature is that of a *type-level function*. We elaborate in Section 3.3.

It is not our purpose to argue the case for these individual features. Rather, we simply observe (a) that they are popular with programmers, and (b) that they all affect the language of type constraints, and in particular, the equality theory on types. Our main point is to show that the "small tricky point" of Hindley-Milner, the implicit generalisation of local `let` bindings, becomes a major obstacle when combined with features such as those we have mentioned above.

2.1 Abstracting over constraints

Consider this definition:

```
data R a where
  RBool :: (a ~ Bool) => R a
  RInt  :: (a ~ Int)  => R a
```

Here `R` is a GADT. Pattern-matching on a value of type `(R a)` gives information about the type `a` — in the case of `RBool` we learn that `a` is equal to `Bool` (because of the constraint `a`$\sim$`Bool` and in the case of `RInt` we learn that `a` is equal to `Int` (because of the constraint `a`$\sim$`Int`). For example, this function is well-typed:

```
h1 :: R a -> a
h1 RBool = True
h1 RInt  = 42
```

In the `RBool` branch we know that `a`$\sim$`Bool`, so it is right to return a `Bool`, namely `True`; in the `RInt` branch `a`$\sim$`Int` so we return an `Int`, namely 42. Now consider this function definition:

```
fr :: a -> R a -> Bool
fr x y = let g z = not x  -- not :: Bool -> Bool
         in case y of
            RBool -> g ()
            RInt  -> True
```

The reader is urged to pause for a moment to consider whether `fr`'s definition is well-typed. After all, `x` clearly has type `a`, and it is passed as an argument to the boolean function `not`. Any normal Hindley-Milner type checker would unify `a` with `Bool` and produce a type error.

Yet there is a type for g that makes the program typecheck, namely

```
g :: forall b. (a ~ Bool) => b -> Bool
```

That is, rather than *rejecting* the constraint `a`$\sim$`Bool`, we *abstract over it*, thereby deferring the (potential) type error to g's call site. At any such call site, we must provide evidence that `a`$\sim$`Bool`, and indeed we can do so in this case, since we are in the `RBool` branch of the match on `y`.

In short, to find the most general type for g, we must abstract over the equality constraints that arise in g's right hand side. We do not *seek* this outcome: in our opinion, most programmers would expect `fr`'s definition to be rejected, as we discuss shortly (Section 2.2). But the fact is that in a system admitting equality constraints, the principal type for g is the one written above.

The very same issue arises with type-class constraints. Consider:

```
data S a where
  MkS :: Show a => a -> S a
```

The data constructor `MkS` takes a `(Show a)` constraint as its argument, and, dually, makes it available inside a pattern match. Hence, for example, this function is well-typed:

```
h2 :: S a -> String
h2 (MkS x) = show x
      -- show :: Show a => a -> String
```

The (`Show a`) constraint that arises from the call of `show` is discharged by the pattern match, so the type of `h2` can be fully polymorphic. (Haskell 98 does not offer data constructors that behave like `MkS` — i.e. where pattern matching can discharge type-class constraints from the body of the match — but GHC does, and they are very useful in practice.)

Now consider this definition of `fs`:

```
fs :: a -> S a -> Bool
fs x y = let h z = show x
         in case y of
            MkS v -> show v ++ h ()
```

Again, the most general type of `h` is

```
h :: forall b. (Show a) => b -> String
```

where we abstract over the (`Show a`) constraint even though `g` is not polymorphic in `a`. Given this type, the call to `h` is well typed, as is the whole definition of `fs`. It should be obvious that the two examples differ only in the kind of constraint that is involved.

2.2 So what is the problem?

So what is the problem? In general, type inference can usefully be viewed as a process of (a) generating and (b) solving constraints (PR05). For a `let` binding, we infer the type τ of the right-hand side, gathering its type constraints Q at the same time. Then we typically *generalise* the type, by universally quantifying over the type variables $\overline{a}$ that are free in τ but are not mentioned in the type environment. But what about Q? One robust and consistent choice (made, for example, by Pottier (PR05; SP07)) is this:

GenAll: Abstract over *all* the constraints Q, regardless of whether the constraint mentions the quantified type variables $\overline{a}$, to form the type $\forall\overline{a}.Q \Rightarrow \tau$.

However **GenAll** has serious disadvantages, of two kinds. First, and most important, there are costs to the programmer:

- It leads to unexpectedly complicated types, such as those for function `g` in Section 2.1. The larger the right-hand side, the more type constraints will be gathered and abstracted over. For type-class constraints this might be acceptable, but equality constraints are generated in large numbers by ordinary unification.

  Although they do not appear in the program text, these types may be *shown* to the programmer by an IDE; and must be *understood* by the programmer if she is to know which programs will typecheck and which will not.
- There are strong software-engineering reasons not to generalise constraints unnecessarily, because doing so postpones type errors from the definition of `g` to (each of) its occurrences. If, for example, `g` had been called in the `RInt` branch of `fr`, as well as the `RBool` branch, a mystifying error would ensue: "Cannot unify `Int` with `Bool`". Why? Because the call to `g` would require `a`$\sim$`Bool` to be satisfied and, and combined with the local knowledge that `a`$\sim$`Int`, the unsatisfiable constraint `Int`$\sim$`Bool` ensues. To understand such errors the programmer will have to construct in her head the principal type for `g`, which is no easy matter. Moreover, one such incomprehensible error will be reported for each call of `g`.
- In an inference algorithm, it turns out that we need a new form of constraint, an *implication constraint*, that embodies deferred typing problems (Section 5.2). Under **GenAll** it is necessary to abstract over implication constraints too, which further complicates the programmer's life (because she sees these weird types). This raises the question of whether implication constraints should additionally be allowed as valid type signatures, which in turn leads to open research problems in tractable solver procedures for implication constraints with implications in their assumptions (SP07).

Second, there are costs to the type inference engine:

- At *each call site* of a generalized expression, the previously abstracted large constraints have to be shown satisfiable. This makes efficient type inference harder to implement.
- Almost all existing Haskell type inference engines (with the exception of Helium (HLvI03)) use the standard Hindley-Milner algorithm, whereby unification (equality) constraints are solved "on the fly" using in-place update of mutable type variables (PVWS07). This is simple and efficient, which is important since equality constraints are numerous. (In contrast the less-common type-class constraints are gathered separately, and solved later.)

  Under **GenAll**, we can no longer eagerly solve *any unification constraint whatsoever* on the fly. An equality $a \sim \tau$ must be suspended (i.e. not solved) if a is free in the environment at some enclosing `let` declaration. Moreover, in compilers with a typed intermediate language, such as GHC, each abstracted constraint leads to an extra type or value parameter to the function, and an extra type or value argument at its occurrences.

These costs might be worth bearing if there was a payoff. But in fact the payoff is close to zero:

- Programmers do not expect `fr` and `fs` to typecheck, and will hardly be delirious if they do so in future. (Indeed, GHC currently rejects both `fr` and `fs`, with no complaints.)
- The generality of `fr` and `fs` made a difference only because the occurrence of `g` was under a pattern-match that bound a new, local constraint. Such pattern matches are rare, so in almost all cases the additional generalisation is fruitless. But it cannot be omitted (at least not without a rather ad-hoc pre-pass) because when processing the perfectly vanilla definition of `g` the type-checker does not know whether or not `g`'s occurrences are under pattern-matches that bind constraints.

In short, we claim that generalising over all constraints carries significant costs, and negligible benefits. Probably the only true benefit is that **GenAll** validates *let-expansion*; that is, `let` $x = e$ `in` b typechecks if and only if $b[e/x]$ typechecks. The reader is invited to return to `fr` and `fs`, to observe that both do indeed typecheck with no complications if `g` is simply inlined. Let-expansion is a property cherished of type theorists and sometimes useful for automatic code refactoring tools, but we believe that its price has become too high.

2.3 Our proposal

If **GenAll** is a poor choice, what else can we do? Our proposal is simple and radical:

NoGen Do not generalise un-annotated local `let`-bindings *at all*. That is, simply omit the entire generalisation step; the definition is completely monomorphic. For *annotated* local `let`-bindings, `let` x`::`$\sigma = e_1$ `in` e_2, where the programmer supplies a (possibly polymorphic) type signature σ, use that type signature. Use **GenAll** for top-level bindings.

Notice that **NoGen** applies only to *local* (i.e. non-top-level) `let`-bindings. For top-level bindings, the type environment is empty, and it turns out that all the difficulties described in Section 2.2 disappear.

As we show in Section 3, the typing rules for **NoGen** are simple. Better still, its implementation is simple: generalisation can simply be omitted, and unification can be eager just as in Hindley-Milner.

Under **NoGen**, both `fr` and `fs` are rejected, which is fine; we did not seek to accept them in the first place. But hang on! **NoGen** means that some vanilla ML or Haskell 98 functions that use polymorphic local definitions, such as `wuggle` in the Introduction, will be rejected. That is **NoGen** is not conservative over Haskell 98. Surely programmers will hate that?

Actually not. In Section 4 we will present evidence that programmers almost never use locally-defined values in a polymorphic way. In the rare cases where a local value *is* used polymorphically, the programmer can readily evade **NoGen** by simply supplying a type signature.

In short, generalisation of local `let` bindings is a device that is almost never used, and its abolition yields a dramatic simplification in both the specification and implementation of a typechecker. The situation is strongly reminiscent of the debate over ML's *value restriction*. In conjunction with assignment, unconditional generalisation is unsound. Tofte proposed a sophisticated work-around (Tof90). But Wright subsequently proposed the value restriction, whereby only syntactic values are generalised (Wri95). The reduction in complexity was substantial, and the loss of expressiveness was minor, and Wright's proposal was adopted.

The rest of this paper fleshes out our argument in several ways:

- Can we be less Draconian than **NoGen**? We explore (and ultimately reject) other choices intermediate between **GenAll** and **NoGen** (Section 3).
- **NoGen** rejects some Haskell 98 programs. How bad is that? We give quantitative evidence that the damage is negligible (Section 4).
- Does **NoGen** solve the problems of Section 2.2? We show that it does (Section 5).

3. Type system options for let-bindings

In this section we systematically analyse a range of options for typing `let`-bound expressions in the type system specification. The typing relation takes the conventional form $Q\,;\Gamma \vdash e : \tau$, to be read as: under the constraint Q, the expression e is typeable in the typing environment Γ with type τ.

3.1 GenAll: generalise all constraints

Here is the typing rule for `let` under the **GenAll** approach:

$$\boxed{\text{Qualified types: }\textbf{Yes}\text{, Generalization: }\textbf{Yes}}$$

$$\frac{Q_1\,;\Gamma \vdash e_1 : \tau_1 \quad \overline{a} = \mathit{ftv}(\tau_1) - \mathit{ftv}(Q,\Gamma) \qquad Q\,;\Gamma,(x{:}\forall\overline{a}.Q_1 \Rightarrow \tau_1) \vdash e_2 : \tau_2}{Q\,;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{LET}$$

GenAll has no technical shortcomings, but we have argued that it is undesirable in practice, so we seek alternatives.

3.2 NoQual: Generalization without qualified types

The undesirability of **GenAll** concerned the abstraction of constraints, rather than generalisation *per se*. What if the specification simply insisted that the type inferred for a `let` binding was always of form $\forall\overline{a}.\,\tau$, with no "$Q \Rightarrow$" part? This is easy to specify:

$$\boxed{\text{Qualified types: }\textbf{No}\text{, Generalization: }\textbf{Yes}}$$

$$\frac{Q\,;\Gamma \vdash e_1 : \tau_1 \quad \overline{a} = \mathit{ftv}(\tau_1) - \mathit{ftv}(Q,\Gamma) \qquad Q\,;\Gamma,(x{:}\forall\overline{a}.\tau_1) \vdash e_2 : \tau_2}{Q\,;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{LET}$$

When Q is empty, this is the usual rule for the Hindley-Milner system. In terms of an inference algorithm, what happens is this. Equality constraints are gathered from the right-hand side, but are *completely solved* before generalisation. A unique solution is guaranteed to exist, namely the most general unifier. (In Hindley-Milner the constraints are solved on-the-fly but that is incidental.)

It turns out that this approach continues to work for a system that has GADTs only; indeed, it is precisely the one we describe in an earlier paper on that topic (SJSV09). Again, a given set of constraints can always be uniquely solved (if a solution exists) by unification.

Alas, adding type classes makes the system fail, in the sense of lacking principal types, because type-class constraints do not have unique solutions in the way that equality constraints do. For example, suppose that in the definition `let` $x = e_1$ `in` e_2, that

- The type of e_1 is $b \to b$.
- b is not free in the type environment.
- The constraints Q arising from e_1 are $Q = \texttt{Eq}\ b$.

We cannot solve Q without knowing more about b — but in this case we propose to quantify over b. If we quantify over b the only reasonable type to attribute to x is

$$x :: \forall b.\,(\texttt{Eq}\ b) \Rightarrow b \to b$$

That is illegal under **NoQual**. As a result, x has many incomparable types, such as `Int` $\to$ `Int` and `Bool` $\to$ `Bool`, but no principal type.

3.3 PartQual: Restricted qualified types and generalization

We have learned that, if we are to generalise `let`-bound variables we must quantify over their type-class constraints (**NoQual** did not work); but we have argued that it is undesirable to quantify over all constraints (i.e. **GenAll**). The obvious alternative is to quantify over type-class constraints, but *not* over equality constraints. More generally, can we identify a particular "class" of constraints over which the specification is allowed to abstract? We call this choice **PartQual**, and use a predicate $\mathit{good}(Q)$ to identify abstractable constraints:

$$\boxed{\text{Qualified types: }\textbf{Restricted}\text{, Generalization: }\textbf{Yes}}$$

$$\frac{QQ_1\,;\Gamma \vdash e_1 : \tau_1 \quad \overline{a} = \mathit{ftv}(\tau_1) - \mathit{ftv}(Q,\Gamma) \qquad \mathit{good}(Q_1) \quad Q\,;\Gamma,(x{:}\forall\overline{a}.\,Q_1 \Rightarrow \tau_1) \vdash e_2 : \tau_2}{Q\,;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{LET}$$

The problem with this approach is that it is not clear what such class of constraints would be. It is not enough to pick out equality constraints, because some equality constraints behave like type class constraints. To see this, we must introduce type functions, a recent extension to Haskell.

A *type function* is a type-level function defined by a set of non-overlapping top-level equations. For example:

```
type family F a
type instance F Int  = Int
type instance F Bool = Int
```

Type functions are most useful in conjunction with type classes (see (CKPM05; CKP05)), but here we study them in isolation.

Now, in the definition `let` $x = e_1$ `in` e_2, suppose that

- The type of e_1 is $b \to b$.
- b is not free in the type environment.
- The constraints Q arising from e_1 are $Q = F\ b{\sim}\texttt{Int}$.

We cannot solve the constraint, so we must quantify over it. Had we instead chosen a more specific type instead of b — remember, this is the declarative specification, not an algorithm — then we could type e with e : `Int` $\to$ `Int` or e : `Bool` $\to$ `Bool`, because the corresponding constraints `F Int` $\sim$ `Int` and `F Bool` $\sim$ `Int` are soluble using the top-level axioms. Alas, neither of these types for

x is more general than the other, so we lose principal types. The only reasonable type[1] to attribute to x is

$$x :: \forall b. (F\ b \sim \texttt{Int}) \Rightarrow b \rightarrow b$$

Well then, can we modify $good(Q)$ to quantify over type class constraints, and equalities involving type functions, but not over equalities that mention no type functions? No, that does not work either. Suppose the constraint was $(\texttt{G}\ b\ \texttt{Int} \sim \texttt{Int})$, where we have

```
type family G a b
type instance G b Int = b
```

Using the `type instance` to rewrite the constraint gives an *ordinary* equality $b \sim \texttt{Int}$ which we are not supposed to abstract!

Another possibility worth mentioning would be to let $good(Q_1)$ be true iff $ftv(Q_1) \subseteq \overline{a}$. This option does not work either. For example, assuming that $ftv(\Gamma, Q) = \{b\}$, a possible constraint arising for a `let`-bound expression may be $F\ b \sim G\ a$ which is not generalizable, because b cannot be quantified over. However, were the specification to choose a more specific type than b, let's say $[b]$, and in the presence of a top-level axiom $F\ [b] \sim \texttt{Int}$, another possible constraint would be $\texttt{Int} \sim G\ a$. This constraint *can* be quantified over, resulting again in lack of principal types. Similar examples can be reconstructed using multi-parameter type classes.

At this stage it is clear that **PartQual** has entered a death spiral, and we give up attempts to resuscitate it.

3.4 NoGen: no generalization

The last, and much the simplest choice, is to perform no generalisation whatsoever for inferred `let` bindings, as proposed in Section 2.3. The typing rule is very simple:

$$\boxed{\text{Qualified types: \textbf{No}, Generalization: \textbf{No}}}$$

$$\frac{Q\,;\Gamma \vdash e_1 : \tau_1 \qquad Q\,;\Gamma, (x{:}\tau_1) \vdash e_2 : \tau_2}{Q\,;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ \text{LET}$$

It seems clear that **NoQual** and **PartQual** are non-starters, and we have argued that **GenAll**, while technically straightforward is practically undesirable.

4. Experimental results

Our **NoGen** proposal will reject some programs that would be accepted by any Haskell or ML compiler. This is bad in two ways:

- Backward compatibility: existing programs will break. But how many programs break? And how easy is it to fix them?
- Convenience: even for newly-written programs, automatic generalisation is convenient. But how inconvenient is programming without it?

To get some quantitative handle on these questions we added a flag to GHC that implements **NoGen**, and performed the following two experiments.

4.1 The boot libraries

We compiled all of the Haskell libraries that are built as part of the standard GHC build process, and fixed all failures due to **NoGen**. These libraries comprise some 30 packages, containing 533 modules, and 94,954 lines of Haskell code (including comments).

[1] Even if `F` had a *unique* solution for `F` $b \sim$`Int` — for example, suppose the axiom `F Int = Int` had not been introduced — we should *still* quantify over the constraint, because we assume an "open world" in which new axioms may be introduced at any time.

In total we found that 20 modules (3.7%) needed modification. The changes affected a total of 127 lines of code (0.13%), and were of three main kinds:

- There are a few occurrences of a polymorphic function that could be defined at top level, but was actually defined locally. For example `Control.Arrow.second` has a local definition for

  ```
  swap ~(x,y) = (y,x)
  ```

- One programmer made repeated use of the following pattern

  ```
  mappend a b = ConfigFlags {
    profLib     = combine profLib,
    constraints = combine constraints,
    ...
  }
    where combine ::Monoid t => (ConfigFlags->t) -> t
          combine field = field a `mappend` field b
  ```

  (The type signature was added by ourselves.) Notice that `a` and `b` are free in `combine`, but that `combine` is used for fields of many different types; for example, `profLib::Flag Bool`, but `constraints::[Dependency]`. This pattern was repeated in many functions. We fixed the code by adding a type signature, but it would arguably be nicer to make `combine` a top-level function, and pass `a` and `b` to it.

- The third pattern was this:

  ```
  let { k = ...blah... } in gmapT k z xs
  ```

  where `gmapT` is a function with a rank-2 type:

  $$\texttt{gmapT} :: \forall a.\texttt{Data}\ a \Rightarrow (\forall b.\texttt{Data}\ b \Rightarrow b \rightarrow b) \rightarrow a \rightarrow a$$

  Here, `k` really must be polymorphic, because it is passed to `gmapT`. GHC's libraries include the Scrap Your Boilerplate library of generic-programming functions that make heavy use of higher rank functions, but in vanilla Haskell code one would expect them to be much less common.

4.2 Packages on Hackage

As a second, and much larger-scale, experiment we compiled all of the third-party Haskell packages on the Hackage library, both with and without **NoGen**, and recorded whether or not the package compiled successfully with the **NoGen** flag on.

We found 793 packages that compiled faultlessly with the baseline compiler that we used. When we disabled generalisation for local `let` bindings, 95 of the 793 (12%) failed to compile. We made no attempt to investigate what individual changes would be needed to make the failed ones compile. Since the chances of an entire package compiling without modification decreases *exponentially* with the size of the package, so one would expect a much larger proportion of *packages* to fail than of *modules* (c.f. the 3.7% of base-package modules that required modification).

4.3 Summary

Although there is more work to do, to see how many type signatures are required to fix the 95 failing third-party packages, we regard these numbers as very promising: even in the higher-rank-rich base library, only a vanishingly small number of lines needed changing. We conclude that local `let` generalisation is rarely used. Moreover, as a matter of taste, in almost all cases we believe that the extra type signatures in the modified base-library code have improved readability. Finally, although our experiments involve Haskell programs, we conjecture that the situation is similar for ML variants.

5. LHM(X)

In this section we give the details of LHM(X), a type system that implements **NoGen**. The key features of LHM(X) are these:

- Rather than choose some specific extension(s), such as GADTs or type classes or units of measure, LHM(X) is parameterised over the constraint domain X. All we require of the constraint domain is that it has a solver that satisfies certain reasonable soundness properties (Section 5.4).
- LHM(X) supports *qualified types*, of form $\forall\overline{a}.\ Q \Rightarrow \tau$.
- LHM(X) supports *local assumptions*. For example, in the branch of a GADT pattern match, we have some equality assumptions that do not hold elsewhere.
- LHM(X) supports user-supplied *type signatures*. Far from making the job easier, type signatures make type inference quite a bit harder, precisely because they constitute a second source of local assumptions.

We will often use GADTs to illustrate and motivate, but they are only illustrative: everything works for an arbitrary X.

5.1 Syntax

The syntax of LHM(X) is given in Figure 1. Expressions e are the standard λ-calculus expressions and we provide `let`-bindings in two forms: (i) ordinary un-annotated `let` bindings, and (ii) `let`-bindings annotated with the type of the bound expression. Finally, we include a pattern matching construct. Programs M are sequences of top-level annotated or un-annotated bindings. Environments Γ contain variable bindings (we assume that constructors and their types appear in some global environment Γ_0).

Types are separated into polymorphic types σ and monotypes τ, υ. In a polymorphic (quantified) type $\forall\overline{a}.\ Q \Rightarrow \tau$, $\overline{a}$ represents a set of quantified variables, Q is a constraint consisting of (at least) type equalities, and τ is a monomorphic type. Monomorphic types include variables, `Int` and `Bool`, lists. Additionally, a single type `T` with constructors introducing constraints suffices to demonstrate all we need for this paper, so our tiny language has a single baked-in such type `T` and its constructors $\overline{K}$. The types of its constructors are assumed to be in the initial environment Γ_0. For simplicity of the formal metatheory we assume that they do not bring existential variables in scope and that the constraints Q_i are non-trivial.

The "..." parts of the syntax are the "X" over which LHM(X) is parameterised. Any given X will have some form of constraint(s) (which extend Q), some new forms of types (which extend τ), and some top-level axioms Q_0. For example, to add type classes we have a new form of constraint (e.g. $\mathtt{Eq}\,\tau$), no new form of types, and top-level axioms derived from `instance` declarations. To add type functions we add no new constraints, but we do add a new form of type $\mathtt{F}\,\overline{\tau}$, and axiom schemes. For example, the Haskell definition

```
type instance F [a] = a
```

introduces an axiom scheme of the form $\forall a.F\ [a] \sim a$ in Q_0. From a technical standpoint, axiom schemes can be viewed as infinite sets of axioms, which are invariant under type substitutions.

Syntactically, we often omit empty qualifiers, writing $Q \Rightarrow \tau$ instead of $\forall\epsilon.Q \Rightarrow \tau$, and τ instead of $\epsilon \Rightarrow \tau$. In our examples, we will often make use of various literal boolean and integer constants and other functions, but we omit those from Figure 1.

$$
\begin{array}{llll}
\text{Constructors} & K & ::= & \ldots \\
 & \nu & ::= & K \mid x \\
\text{Expressions} & e & ::= & \nu \mid \lambda x\,.\,e \mid e_1\ e_2 \\
 & & \mid & \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\
 & & \mid & \mathtt{let}\ x\mathtt{::}\sigma = e_1\ \mathtt{in}\ e_2 \\
 & & \mid & \mathtt{case}\ e\ \mathtt{of}\{\overline{K\,\overline{x} \rightarrow e}\} \\
\text{Programs} & M & ::= & \epsilon \mid \mathtt{let}\ x = e, M \\
 & & \mid & \mathtt{let}\ x\mathtt{::}\sigma = e, M \\
\text{Polytypes} & \sigma & ::= & \forall\overline{a}\,.\,Q \Rightarrow \tau \\
\text{Contexts} & Q & ::= & \epsilon \mid Q_1, Q_2 \mid \tau_1 {\sim} \tau_2 \mid \ldots \\
\text{Monotypes} & \tau, \upsilon & ::= & \alpha \mid a \mid \mathtt{Int} \mid \mathtt{Bool} \mid [\tau] \mid \mathtt{T}\ \overline{\tau} \mid \ldots \\
 & \Gamma & ::= & \epsilon \mid (x{:}\sigma), \Gamma \\
\text{Free type vars.} & & & ftv(\cdot)
\end{array}
$$

Γ_0 : Types of data constructors

$$K \quad : \quad \forall\overline{a}\,.\,Q_1 \Rightarrow \overline{\upsilon}_1 \rightarrow \mathtt{T}\ \overline{a}$$

Q_0 : Top-level axiom scheme set

$$\ldots$$

$$
\begin{array}{llll}
\text{Simple constraints} & Q & ::= & \epsilon \mid Q_1, Q_2 \mid \tau_1 {\sim} \tau_2 \mid \ldots \\
\text{Implication constr.} & C & ::= & \epsilon \mid C_1, C_2 \mid [\overline{\alpha}](Q \supset C)
\end{array}
$$

Figure 1: Syntax of LHM(X)

5.2 Overview of the OutsideIn approach

Previous work has addressed the problem of type inference for GADTs, based on generating and solving implication constraints (SP07; SJSV09; SWJ06; SSS08). Our type system and algorithm for LHM(X) are based on those of the **OutsideIn** approach (SJSV09), so we begin with a review of the latter. The **OutsideIn** approach only targets GADTs, and hence our examples below are GADT examples.

The algorithm GADT pattern matching introduces local assumptions to the type checker, which in turn give rise to so-called *implication constraints*, C. Recall the `R` datatype from Section 2.1 and consider the following example:

```
trans :: forall a. R a -> a -> a
trans rx x = case rx of
               RInt  -> 3
               RBool -> True
```

In the example, the type of `rx` is the known type `R` a, and the type of `x` is a. The first branch introduces a local assumption $a \sim \mathtt{Int}$ which may be used to type the right-hand side of the branch. For the latter we must be able to show $a \sim \mathtt{Int}$, so the type checker must solve this implication constraint:

$$(a \sim \mathtt{Int}) \supset (a \sim \mathtt{Int})$$

which is trivially satisfiable. Similarly, for the second branch, the (trivially satisfiable) constraint $(a \sim \mathtt{Bool}) \supset (a \sim \mathtt{Bool})$ arises. During type inference, constraints often contain unknown *unification variables*, for which we use greek letters $\alpha, \beta, \ldots$. Consider this function, where the programmer does not supply a signature:

```
foo rx = case rx of
           RInt  -> 3
           RBool -> error
```

Assuming that `R` α is the type of `rx` and β is the return type of the expression, the constraint arising is:

$$(\alpha \sim \mathtt{Int}) \supset (\beta \sim \mathtt{Int})$$

(There is no constraint for the second branch, as `error` can have any type we wish). In this case, the constraint solver is *additionally* required to actually produce a substitution for α, β that solves the constraints. There exist various possibilities for β: it may set $[\beta \mapsto \mathtt{Int}]$ or $[\beta \mapsto \alpha]$. Depending on the choice, we get two different types for `foo`:

$$\forall a.\mathtt{R}\ a \to \mathtt{Int} \quad \text{or} \quad \forall a.\mathtt{R}\ a \to a$$

Notice, however that these two types are incomparable; and there is *no type more general than both* that does not involve quantification over the full implication constraint!

The **OutsideIn** approach rejects `foo`, because it lacks a principal type. Algorithmically it is easy to reject `foo`. We record in each implication constraint a set of "untouchable" unification variables, which are the variables through which a GADT branch may communicate with the "outside" part of the program (Figure 1). In the example, those variables are both α and β, because α belongs in the environment and β is the return type of the branch. The implication constraint becomes:

$$[\alpha,\beta](\alpha \sim \mathtt{Int} \supset \beta \sim \mathtt{Int})$$

Now, **OutsideIn** requires the constraint to be soluble *without* substituting for the untouchable variables. Hence, no ambiguity in the type of expressions can occur as a result of solving an implication constraint in two different ways.

On the other hand, if information from the "outside world" fixes the solution to the implication constraint, the **OutsideIn** approach uses it. For example, assume:

```
foo rx = (case rx of
            RInt -> 3
            RBool -> error) :: Int
```

The generated constraint is:

$$([\alpha,\beta](\alpha \sim \mathtt{Int} \supset \beta \sim \mathtt{Int}) \wedge (\beta \sim \mathtt{Int})$$

since the annotation specifies that $\beta \sim \mathtt{Int}$. Having fixed β to `Int`, the implication constraint is trivially soluble, without producing any extra substitutions. Hence, algorithmically, **OutsideIn** first solves the flat top-level constraints (the *simple constraints*), and uses that information to solve nested implication constraints (the *proper implication constraints*).

The specification It is less straightforward to provide a type system for this informally described algorithm. The basic idea of the **OutsideIn** *type system* is that GADT pattern matching gives rise to *deferred typing problems*. Formally, we make use of a typing judgement of the form $Q\,;\Gamma \vdash e : \tau \rhd P$, where Q is a constraint context, Γ is the typing environment, and P is a set of deferred typing problems, each a quadruple of the form $\langle Q'\,;\Gamma'\,;e'\,;\tau'\rangle$. The intuition behind the deferred typing problem sets P is that, no matter which individual quadruples such a P may contain, these quadraples have to form valid typing judgements; we return to the details in Section 5.5. Whenever the $Q\,;\Gamma \vdash e : \tau \rhd P$ judgement encounters a GADT match, the return type of the branch is allowed to be anything (because the "outside" part of the program should be opaque to the branch), and the typing problem that corresponds to the branch is deferred in the returned P set. For instance, pattern matching for the `R` datatype can be typed by:

$$\frac{Q\,;\Gamma \vdash e : \mathtt{R}\,\tau \rhd P \qquad P_1 = \{\langle \tau{\sim}\mathtt{Bool};\Gamma;e_1;\tau_r\rangle\} \cup \{\langle \tau{\sim}\mathtt{Int};\Gamma;e_2;\tau_r\rangle\} \cup P}{Q\,;\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\{\mathtt{RBool} \to e_1, \mathtt{RInt} \to e_2\} : \tau_r \rhd P_1}$$

Notice that after typechecking e and yielding deferred problems P, we extend P with the two deferred branch typing problems. For each branch we record the constraint that arises from the corresponding constructor ($\tau \sim \mathtt{Bool}$ and $\tau \sim \mathtt{Int}$, respectively). The rest of the rules are quite standard, gathering up additionally all GADT branch deferred problems[2].

At the top-level, a recursive judgement $Q_0\,;Q\,;\Gamma \vdash^{\mathsf{R}} e : \tau$ ensures that for any possible way to type a program with type τ and given constraint Q in an top-level axiom set Q_0, while deferring the GADT branch typings, those deferred typing problems are themselves valid recursively in the $\vdash^{\mathsf{R}}$ judgement. Hence, this specification realizes the **OutsideIn** idea.

5.3 The LHM(X) type system and constraint generation

We now formally present the proposed type system. The main typing relation takes the form $Q\,;\Gamma \vdash e : \tau \rhd P$ and is given in Figure 2.

Rules EQ and NU check the entailment of the wanted constraint from the available constraint. We hence rely on a judgement of the form $Q_1 \Vdash Q_2$, which depends on the domain X that parameterizes LHM(X). We will assume however a few properties of $Q_1 \Vdash Q_2$.

1 Definition [Proof theory requirements]: We assume that the empty constraint ϵ is treated as the always valid constraint by $\Vdash$ and Q_1, Q_2 is treated as the conjunction of Q_1 and Q_2. In addition we require that:

- If $Q_1 \Vdash Q_2$ and $Q_2 \Vdash Q_3$ then $Q_1 \Vdash Q_3$.
- If $Q_1 \Vdash Q_2$ then $\theta Q_1 \Vdash \theta Q_2$.

For example one could imagine a concrete system that includes type equalities and type class constraints. These conditions are expected; for example Jones identifies the same conditions in his thesis (Jon92).

Most of the typing rules in Figure 2 are straightforward. In accordance with our earlier discussion, rule LET does not perform any generalization and hence gives the bound variable x a monomorphic type τ_1.

The rule for annotated `let` definitions comes in two flavours. If the annotation does not bring any context in scope, i.e. it is simply a τ_1 type, then rule LETA triggers. It type checks the definition, binds the variable to the annotation type and checks the body of the definition. If the annotation does bring context in scope, i.e. is of the form $\forall\overline{a}\,.\,Q \Rightarrow \tau$ where either $\overline{a}$ or Q is non-empty, then we treat the typing of the definition as a deferred typing problem. Consider:

```
flop x = let foo :: forall a.(a ~ Int)=>a -> Bool
             foo = e
         in ...
```

Then, if Q is the constraint arising from `e`, Q has to be satisfiable in a context that provides $(a \sim \mathtt{Int})$. The situation is analogous to GADT pattern matching, and rule GLETA makes sure that we treat it the same way by creating a deferred typing problem.

Finally, notice that the judgement $Q\,;\Gamma \vdash e : \tau \rhd P$ does not use the top-level axiom set Q_0 — the Q_0 axiom set comes into play only at the typing rules for top-level bindings.

Constraint generation The typing rules in Figure 2 go hand-in-hand with the constraint generation rules in Figure 3.

Recall from Figure 1 that the language of monomorphic types is allowed to contain unification variables $\alpha, \beta, \ldots$. We introduce algorithmic constraints, that include the familiar simple constraints Q, conjunctions C_1, C_2, and additionally implication constraints of the form $[\overline{\alpha}](Q \supset C)$. Notice that the assumption of an implication constraint is always a simple constraint Q, since it is always introduced by pattern matching, or by polymorphic user annotations.

[2] In the original **OutsideIn** type system local `let` definitions get generalized but non-qualified types. It only works because the type system supports GADTs (and no type functions or type classes), which can always be discharged by means of substitutions.

$$\boxed{Q\,;\Gamma \vdash e : \tau \triangleright P}$$

$$\frac{\begin{array}{c}(\nu{:}\forall\overline{a}\,.\,Q_1 \Rightarrow \tau_1) \in \Gamma \cup \Gamma_0 \\ Q \Vdash [\overline{a \mapsto \upsilon}]\,Q_1\end{array}}{Q\,;\Gamma \vdash \nu : [\overline{a \mapsto \upsilon}]\tau_1 \triangleright \emptyset}\ \text{NU}
\qquad
\frac{\begin{array}{c}Q\,;\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \triangleright P_1 \\ Q\,;\Gamma \vdash e_2 : \tau_1 \triangleright P_2\end{array}}{Q\,;\Gamma \vdash e_1\ e_2 : \tau_2 \triangleright P_1 \cup P_2}\ \text{APP}
\qquad
\frac{Q\,;\Gamma,(x{:}\tau_1) \vdash e : \tau_2 \triangleright P}{Q\,;\Gamma \vdash \lambda x\,.\,e : \tau_1 \rightarrow \tau_2 \triangleright P}\ \text{ABS}$$

$$\frac{Q\,;\Gamma \vdash e : \tau_1 \triangleright P \quad Q \Vdash \tau_1 \sim \tau_2}{Q\,;\Gamma \vdash e : \tau_2 \triangleright P}\ \text{EQ}
\qquad
\frac{Q\,;\Gamma \vdash e_1 : \tau_1 \triangleright P_1 \quad Q\,;\Gamma,(x{:}\tau_1) \vdash e_2 : \tau_2 \triangleright P_2}{Q\,;\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2 \triangleright P_1 \cup P_2}\ \text{LET}$$

$$\frac{\begin{array}{c}Q\,;\Gamma \vdash e_1 : \tau_1 \triangleright P_1 \\ Q\,;\Gamma,(x{:}\tau_1) \vdash e_2 : \tau_2 \triangleright P_2\end{array}}{Q\,;\Gamma \vdash \mathtt{let}\ x\ \mathtt{::}\ \tau_1 = e_1\ \mathtt{in}\ e_2 : \tau_2 \triangleright P_1 \cup P_2}\ \text{LETA}
\qquad
\frac{\begin{array}{c}\sigma_1 = \forall\overline{a}\,.\,Q_1 \Rightarrow \tau_1 \quad Q_1 \neq \epsilon \vee \overline{a} \neq \epsilon \quad \overline{a}\#\mathit{ftv}(Q,\Gamma) \\ P_1 = \{\langle Q_1\,;\Gamma\,;e_1\,;\tau_1\rangle\} \quad Q\,;\Gamma,(x{:}\sigma_1) \vdash e_2 : \tau_2 \triangleright P_2\end{array}}{Q\,;\Gamma \vdash \mathtt{let}\ x\ \mathtt{::}\ \sigma_1 = e_1\ \mathtt{in}\ e_2 : \tau_2 \triangleright P_1 \cup P_2}\ \text{GLETA}$$

$$\frac{\begin{array}{c}Q\,;\Gamma \vdash e : \mathtt{T}\,\overline{\tau} \triangleright P \quad K_i{:}\forall\overline{a}\,.\,Q_i \Rightarrow \overline{\upsilon_i} \rightarrow \mathtt{T}\,\overline{a} \in \Gamma_0 \\ P_i = \{\langle[\overline{a \mapsto \tau}]\,Q_i\,;\Gamma,(\overline{x{:}[\overline{a \mapsto \tau}]\upsilon_i})\,;e_i\,;\tau_r\rangle\}\end{array}}{Q\,;\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ \{\overline{K_i\ \overline{x}_i \rightarrow e_i}\} : \tau_r \triangleright P \cup (\bigcup P_i)}\ \text{GCASE}$$

$$\boxed{Q_1 \Vdash Q_2}$$

…parameter X proof theory …

Figure 2: Main typing rules of LHM(X)

Finally, we write $\mathbf{simp}[C_1]$ for the non-implication constraints of C_1 and $\mathbf{prop}[C_1]$ for the implications of C_1.

Constraint generation is given with the judgement $\Gamma \vdash\!\!\blacktriangleright e : \tau \rightsquigarrow C$ in Figure 3, where τ and C should be viewed as outputs. The rules in Figure 3 are straightforward — we introduce fresh unification variables whenever the types are unknown and create implication constraints at rules CASE and GLETA, carefully recording the untouchable variables.

5.4 Top-level algorithmic rules

Type checking expressions is only half the story. We need to show how to type check top-level bindings and, algorithmically, what to do with the generated constraints. We start with the latter, as it provides intuitions for the top-level typing rules.

Consider the top-level algorithmic rules in Figure 4. The judgement $Q_0\,;\Gamma \vdash\!\!\blacktriangleright M$ shows how to deal with top-level bindings M, and relies on a constraint solver procedure, with the signature:

$$Q_0\ ;\ Q_{given}\ ;\overline{\tau}_{untch} \vdash \mathit{solve}(C_{wanted}) \rightsquigarrow Q_{residual}\ ;\theta$$

In this signature, the *inputs* are:

- The top-level axiom set Q_0,
- the given (simple) constraints Q_{given} that arise from type annotations (or pattern matching),
- the types $\overline{\tau}_{untch}$ whose free unification variables the solver must not substitute for, and
- the constraint C_{wanted} that the solver is requested to solve.

The *outputs* are:

- A set of (simple) constraints $Q_{residual}$ that the solver was not able to solve,
- A substitution θ for the unification variables of C_{wanted} that do not appear in $\overline{\tau}_{untch}$.

Notice that the solver is not required to always fully discharge C_{wanted} via a substitution for the unification variables; it may instead return a residual $Q_{residual}$. This behaviour may appear when inferring a type for a top-level unannotated binding (rule BIND). In that case, we first produce a constraint for the expression and then appeal to the solver, providing $Q_{given} = \epsilon$ and untouchables $\overline{\tau}_{untch} = \epsilon$. But the solver may fail to fully solve the constraint, returning $Q_{residual} = Q$ along with a substitution θ. In that case we must give the binding a type that quantifies over Q. In contrast to the examples presented in the introduction, there are no technical obstacles to this quantification — since the typing environment is closed we may simplify C_{wanted} as much as we wish.

On the other hand, in the case of annotated bindings (rule ABIND) we solve the constraint, providing as given constraint the Q from the type annotation and equating the type of the annotation (τ) to the type that we have inferred for the expression (υ). Contrary

$$\boxed{Q_0\,;\Gamma \vdash\!\!\blacktriangleright M}$$

$$\frac{}{Q_0\,;\Gamma \vdash\!\!\blacktriangleright \epsilon}\ \text{EMPTY}$$

$$\frac{\begin{array}{c}\Gamma \vdash\!\!\blacktriangleright e : \tau \rightsquigarrow C \quad Q_0\,;\epsilon\,;\epsilon \vdash \mathit{solve}(C) \rightsquigarrow Q\,;\theta \\ \overline{a}\ \text{fresh} \quad \overline{\alpha} = \mathit{fuv}(\theta\tau,\,Q) \\ Q_0\,;\Gamma,(x{:}\forall\overline{a}\,.\,[\overline{\alpha \mapsto a}](Q \Rightarrow \theta\tau)) \vdash\!\!\blacktriangleright M\end{array}}{Q_0\,;\Gamma \vdash\!\!\blacktriangleright \mathtt{let}\ x = e, M}\ \text{BIND}$$

$$\frac{\begin{array}{c}\Gamma \vdash\!\!\blacktriangleright e : \upsilon \rightsquigarrow C \\ Q_0\ ;\ Q\ ;\epsilon \vdash \mathit{solve}(C \wedge \upsilon \sim \tau) \rightsquigarrow \epsilon\,;\theta \\ Q_0\,;\Gamma,(x{:}\forall\overline{a}\,.\,Q \Rightarrow \tau) \vdash\!\!\blacktriangleright M\end{array}}{Q_0\,;\Gamma \vdash\!\!\blacktriangleright \mathtt{let}\ x\ \mathtt{::}\ (\forall\overline{a}\,.\,Q \Rightarrow \tau) = e, M}\ \text{ABIND}$$

$$\boxed{Q_0\ ;\ Q_{given}\ ;\overline{\tau}_{untch} \vdash \mathit{solve}(C_{wanted}) \rightsquigarrow Q_{residual}\ ;\theta}$$

$$\frac{\begin{array}{c}Q_0\ ;\ Q_g\ ;\overline{\tau} \vdash \mathit{simpleS}(\mathbf{simp}[C]) \rightsquigarrow Q_r\ ;\theta \\ \forall([\overline{\tau}_i](Q_i \supset C_i) \in \mathbf{prop}[\theta C]), \\ Q_0\ ;\ Q_g\,Q_r\,Q_i\ ;\overline{\tau\tau}_i \vdash \mathit{solve}(C_i) \rightsquigarrow \epsilon\,;\theta_i\end{array}}{Q_0\ ;\ Q_g\ ;\overline{\tau} \vdash \mathit{solve}(C) \rightsquigarrow Q_r\ ;\theta}\ \text{SOLVE}$$

$$\boxed{Q_0\ ;\ Q_{given}\ ;\overline{\tau}_{untch} \vdash \mathit{simpleS}(Q_{wanted}) \rightsquigarrow Q_{residual}\ ;\theta}$$

…parameter X solver …

Figure 4: Top-level algorithmic rules

$$\boxed{\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C}$$

$$\frac{\begin{array}{c}\overline{\alpha}\text{ fresh}\\ (\nu{:}\forall\overline{a}\,.\,Q_1 \Rightarrow \tau_1) \in \Gamma \cup \Gamma_0\end{array}}{\Gamma \mathrel{|\!\!\blacktriangleright} \nu : [\overline{a \mapsto \alpha}]\tau_1 \rightsquigarrow [\overline{a \mapsto \alpha}]Q_1}\ \textsc{nu} \qquad \frac{\Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau_1 \rightsquigarrow C_1 \quad \Gamma \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \mathrel{|\!\!\blacktriangleright} e_1\ e_2 : \alpha \rightsquigarrow C_1, C_2, \tau_1 \sim \tau_2 \to \alpha}\ \textsc{app} \qquad \frac{\begin{array}{c}\alpha\text{ fresh}\\ \Gamma, (x{:}\alpha) \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C\end{array}}{\Gamma \mathrel{|\!\!\blacktriangleright} \lambda x\,.\,e : \alpha \to \tau \rightsquigarrow C}\ \textsc{abs}$$

$$\frac{\Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau_1 \rightsquigarrow C_1 \quad \Gamma, (x{:}\tau_1) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1, C_2}\ \textsc{let} \qquad \frac{\Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau \rightsquigarrow C_1 \quad \Gamma, (x{:}\tau_1) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x\ \texttt{::}\ \tau_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1, C_2, \tau \sim \tau_1}\ \textsc{leta}$$

$$\frac{\begin{array}{c}\sigma_1 = \forall\overline{a}\,.\,Q_1 \Rightarrow \tau_1 \quad Q_1 \neq \epsilon \vee \overline{a} \neq \epsilon\\ \Gamma \mathrel{|\!\!\blacktriangleright} e_1 : \tau \rightsquigarrow C\\ C_1 = [\mathit{fuv}(\Gamma)](Q_1 \supset C, \tau \sim \tau_1)\\ \Gamma, (x{:}\sigma_1) \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2\end{array}}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{let}\ x\ \texttt{::}\sigma_1 = e_1\ \texttt{in}\ e_2 : \tau_2 \rightsquigarrow C_1, C_2}\ \textsc{gleta} \qquad \frac{\begin{array}{c}\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C \quad \beta, \overline{\gamma}\text{ fresh} \quad C' = (\texttt{T}\ \overline{\gamma} \sim \tau), C\\ \overline{K_i{:}\forall\overline{a}\,.\,Q_i \Rightarrow \overline{\upsilon}_i \to \texttt{T}\ \overline{a} \quad \Gamma, (\overline{x{:}[\overline{a \mapsto \gamma}]\upsilon}) \mathrel{|\!\!\blacktriangleright} e_i : \tau_i \rightsquigarrow C_i}\\ C_i' = [\beta, \overline{\gamma}, \mathit{fuv}(\Gamma)]([\overline{a \mapsto \gamma}]Q_i \supset C_i \wedge \tau_i \sim \beta)\end{array}}{\Gamma \mathrel{|\!\!\blacktriangleright} \texttt{case}\ e\ \texttt{of}\ \{\overline{K_i\ \overline{x}_i \to e_i}\} : \beta \rightsquigarrow C', \overline{C_i'}}\ \textsc{gcase}$$

Figure 3: Constraint generation

to the case of rule BIND, the wanted constraint $C \wedge (\tau \sim \upsilon)$ now has to be fully solved — it is unsound to allow the solver to return a non-trivial residual constraint.

Parameterizing over the X-solver We now turn to the internals of the main solver judgement

$$Q_0\ ;\ Q_{given}\ ;\overline{\tau}_{untch} \vdash \mathit{solve}(C_{wanted}) \rightsquigarrow Q_{residual}\ ; \theta$$

We observe that C_{wanted} may well contain implication constraints — but our proof theory for X does not deduce implications. Hence the solve procedure provides the *infrastructure* of dealing with implication constraints and internally appeals to a domain-specific solver for the X theory, $\mathit{simpleS}$.

The definition of the solve procedure is given by rule SOLVE in Figure 4. The judgement first appeals to the domain-specific $\mathit{simpleS}$ solver for the simple part of the constraint $\mathbf{simp}[C]$, producing a residual constraint Q_r and a substitution θ. Subsequently, it applies θ to each of the proper implication constraints in C and recursively solves each of the implications having updated the given constraints and the set of untouchable types.

Notice that each constraint C_i in a recursive call to solve must be completely solved (wich is ensured with condition $Q_0; Q_g\, Q_r\, Q_i;$ $\overline{\tau\tau}_i \vdash \mathit{solve}(C_i) \rightsquigarrow \epsilon\ ; \theta_i$ in rule SOLVE). The reason is that the residual constraint returned from solve may only be a simple (non-implication) constraint since we are not allowed to quantify over implication constraints. Moreover, the domain of each θ_i is necessarily disjoint from the current environment variables (since the latter belong in the untouchable variables of the corresponding implication constraint), and hence there is no point in returning those θ_i substitutions along with θ in the conclusion of rule SOLVE.

The main solver appeals to a domain-specific solver for the X theory, $\mathit{simpleS}$, with the following signature:

$$Q_0\ ;\ Q_{given}\ ;\overline{\tau}_{untch} \vdash \mathit{simpleS}(Q_{wanted}) \rightsquigarrow Q_{residual}\ ; \theta$$

Any solver with this signature can be "plugged" in our main solver, but in addition we require certain conditions on $\mathit{simpleS}$ that ensure soundness of type inference.

2 Definition [Soundness conditions]: If $Q_0\ ;\ Q_{given}\ ;\overline{\tau}_{untch} \vdash \mathit{simpleS}(Q_{wanted}) \rightsquigarrow Q_{residual}\ ; \theta$ then:

- $Q_{residual}$ and $\overline{\tau}_{untch}$ are fixpoints of θ,
- $Q_0\, Q_{given}\, Q_{residual} \Vdash \theta Q_{wanted}$ and $Q_0\, Q_{given}\, Q_{wanted} \Vdash \mathcal{E}_\theta \wedge Q_{residual}$, where $\mathcal{E}_\theta$ is the equational constraint induced by θ.

The first condition ensures that the substitution is already applied to the residual constraint returned and respects the untouchable variables. The second condition ensures that the Q_{wanted} is *equivalent* to $Q_{residual} \wedge \mathcal{E}_\theta$. For example, when $Q_{residual} = \epsilon$ then Q_{wanted} is dischargeable by θ (and additionally Q_{wanted} implies $\mathcal{E}_\theta$ and $Q_{residual}$, a point that we return to in Section 6.5).

5.5 Top-level typing rules

Section 5.4 describes an *inference algorithm* for the top-level bindings, but what should the *specification* be? Curiously, it seems to be harder to pin down than the algorithm. A plausible specification appears in Figure 5.

First of all, following the **OutsideIn** ideas, we introduce a recursive judgement $Q_0\ ;\ Q\ ; \Gamma \vdash^{\mathsf{R}} e : \tau$, which ensures that for all possible typings of the program from outside, *i.e. intuitively fixing Q, Γ, and τ*, the deferred problems are typeable. Rule RMAIN gives the details. In particular, we first type the program with the required type τ, using a constraint Q_1, which must follow from the top-level axiom set and whatever our given context is (condition $Q_0\, Q \Vdash Q_1$). Next, we require that, no matter which set of deferred problems P_2 we may produce for the same τ and Γ, all the deferred problems in P_2 be valid recursively.

The judgement $Q_0\, ; \Gamma \vdash M$ gives the type checking of top-level bindings. The case for annotated definitions, rule ABIND appeals directly to the $\vdash^{\mathsf{R}}$ judgement passing in the constraint Q and type τ provided by the annotation. Rule BIND on the other hand "guesses" Q and τ to be passed to the $\vdash^{\mathsf{R}}$ judgement, and gives the unannotated binding the quantified type $\forall\overline{a}\,.\,Q \Rightarrow \tau$.

Under sound simplifier assumptions, it is not hard to show soundness of type inference.

3 Proposition [Soundness]: If $Q_0\ ; \epsilon \mathrel{|\!\!\blacktriangleright} M$ then $Q_0\ ; \epsilon \vdash M$.

The technical development that leads up to this result is mostly straightforward – with an extra step of showing that (since we defer the "hard" problems) there exist most general constraints and types for programs typeable in the $\vdash$ judgement.

However, completeness of our simple algorithm is unfortunately not true without modifications in the specification. One difficulty has to do with the **OutsideIn** approach and is best demonstrated with an example. Assume the following top-level definitions:

```
f1 x = x
f2 rx = f1 (case rx of
             RInt -> 3
             RBool -> error)
```

Suppose that we were *not* to choose the most general type for `f1` and instead of $\forall a\,.\,a \to a$ we bind `f1` with type `Int` $\to$ `Int` in the environment. In this case, the type of the `case` expression inside the body of `f2` is "fixed" because the `case` expression is applied to

$$\boxed{Q_0\,;\Gamma \vdash M}$$

$$\frac{\mathit{ftv}(\Gamma) = \emptyset}{Q_0\,;\Gamma \vdash \epsilon}\ \textsc{Empty} \qquad \frac{\begin{array}{c} Q_0\,;\,Q\,;\Gamma \vdash^{\mathsf{R}} e : \tau \\ Q_0\,;\Gamma, (x{:}\forall\overline{a}\,.\,Q \Rightarrow \tau) \vdash M \end{array}}{Q_0\,;\Gamma \vdash \mathtt{let}\ x = e, M}\ \textsc{Bind}$$

$$\frac{\begin{array}{c} Q_0\,;\,Q, \Gamma \vdash^{\mathsf{R}} e : \tau \\ Q_0\,;\Gamma, (x{:}\forall\overline{a}\,.\,Q \Rightarrow \tau) \vdash M \end{array}}{Q_0\,;\Gamma \vdash \mathtt{let}\ x\ \mathtt{::}\ (\forall\overline{a}\,.\,Q \Rightarrow \tau) = e, M}\ \textsc{ABind}$$

$$\boxed{Q_0\,;\,Q\,;\Gamma \vdash^{\mathsf{R}} e : \tau}$$

$$\frac{\begin{array}{c} Q_1\,;\Gamma \vdash e : \tau \rhd P_1 \quad Q_0\,Q \Vdash Q_1 \\ \forall Q_2, P_2, (Q_2\,;\Gamma \vdash e : \tau \rhd P_2) \Longrightarrow \\ \forall \langle Q_i;\Gamma_i;e_i;\tau_i \rangle \in P_2, Q_0\,;\,QQ_i\,;\Gamma_i \vdash^{\mathsf{R}} e_i : \tau_i \end{array}}{Q_0\,;\,Q\,;\Gamma \vdash^{\mathsf{R}} e : \tau}\ \textsc{RMain}$$

Figure 5: Top-level typing rules

f1. As a consequence, the program would be typeable. However, the algorithm only produces $\forall a\,.\,a \rightarrow a$ for f1 by looking at its definition. But the application f1 (case...) *does not* fix the type of the case expression, and the algorithm rejects this program.

Fortunately, there exists an easy solution to this problem — we can require that each un-annotated binding in rule BIND gets its most general type, so that no guessed type information can be used to fix the type of GADT branches in the scope of that binding.

5.6 Lack of completeness due to ambiguity

Even if we insist on inferring most-general types, as argued in the previous section, our inference algorithm remains incomplete — or rather our specification is too liberal — a property it shares with every other type system for Haskell known to us. The reason is the celebrated *ambiguity* problem, illustrated by the following example:

4 Example [Type class ambiguity]: Consider the classical type class example below:

```
-- show :: forall a. Show a => a -> String
-- read :: forall a. Show a => String -> a

flop :: String -> String
flop s = show (read s)
```

The constraint solver is left with a type class constraint Show α, where α is otherwise unconstrained. The constraint can be solved by arbitrarily choosing $[\alpha \mapsto \mathtt{Int}]$, or $[\alpha \mapsto \mathtt{Bool}]$, since both types are instances of Show, *but this arbitrary choice changes the dynamic semantics of the program.* The Haskell 98 Report therefore requires the program to be rejected, and the algorithm to do so is easy: simply refrain from "guessing" the instantiation of an unconstrained unification variable.

Alas, it is hard for the *specification*, the declarative typing rules, to reject the program. There are many perfectly sensible typing derivations for flop, one choosing the result of (read s) to be Int, one choosing it to be Bool, and so on. We know of no elegant specification that excludes such typings, and indeed the rules of Figure 2 do not. In the Haskell jargon, this is the *ambiguity problem* (e.g. see discussion in (Jon92)).

Similar ambiguity problems arise with type functions. If the resulting constraint is $F\ \beta \sim \mathtt{Int}$ with the top-level axioms $F\ \mathtt{Int} \sim \mathtt{Int}$ and $F\ \mathtt{Bool} \sim \mathtt{Int}$ no reasonable algorithm can simply "choose" either Int or Bool. Even if there is only one declared axiom for F, for example $F\ \mathtt{Int} \sim \mathtt{Int}$, we should not expect the algorithm to deduce that $[\beta \mapsto \mathtt{Int}]$. Under an "open world" assumption new axioms could later be introduced that no longer justify our choice to make β equal to Int. More worryingly, such ambiguous constraints may appear nested inside implication constraints, for example $[\epsilon](Q \supset \ldots, F\ \beta \sim \mathtt{Int}, \ldots)$.

The ambiguity problem reveals a mis-match between the algorithm (which does the Right Thing) and the specification (which wrongly admits some ambiguous programs). This problem is quite orthogonal to the contributions of this paper: every type system for Haskell shares the same difficulty. It is unclear whether there exists an elegant, perspicuous specification that excludes ambiguous programs, even for Haskell 98; certainly we have not seen one.

6. Discussion

6.1 Top-level bindings

Many Haskell programmers believe that *every* top-level declaration should also have a type signature, and GHC implements a warning flag to test for precisely this. Our **NoGen** proposal enforces this practice for local bindings, but arguably things would be more uniform if we refrained from generalizing top-level bindings as well, quite similarly to C# or Java. On the other hand, such a choice would reject many Haskell 98 programs, and hence we did not adopt it in this paper.

6.2 Polymorphic but unqualified type signatures

The careful reader will observe that if the annotation type is $\forall\overline{a}\,.\,\tau_1$ then GLETA triggers, rather than rule LETA. This in turn makes the environment unification variables (denoted with $\mathit{fuv}(\Gamma)$) untouchable — contrary to what one would expect for a vanilla Haskell 98 polymorphic type signature. This is not a serious limitation; we have chosen on purpose to treat $\forall\overline{a}.\tau_1$ in the same way as $\forall\overline{a}\,.\,Q_1 \Rightarrow \tau_1$ because otherwise the constraint language would have to be slightly more complex, and so would be the equational theory of constraints and the metatheory. For example, the original **OutsideIn** paper which does allow this extra freedom, requires additionally constraints of the form $[\overline{\beta}](\forall\overline{a}\,.\,C)$ that look like implications but have to be treated as simple constraints, with an additional escape check. Here, we have avoided this complication since it is completely irrelevant for our purposes.

6.3 The monomorphism restriction

Haskell 98 already restricts generalisation of let bindings in one case: the notorious and complicated Monomorphism Restriction (Section 4.5 in (Pey03)). The MR has always been a wart on the language design, and many have argued for its abolition, although we will not rehearse these arguments here.

Our **NoGen** proposal completely subsumes the MR for *local* let bindings, so the MR would then only apply to top-level bindings. In that case the arguments for its abolition, or for generating a warning rather than rejecting the program, would become very strong. A nasty wart would thereby be removed from the face of Haskell.

6.4 Units of measure

Kennedy's system of units of measure (Ken96) was briefly sketched in Section 2. Lacking qualified types, Kennedy adopted **NoQual** but then, quite unexpectedly, discovered a type inference completeness problem:

```
div :: forall u1 u2.
         num (u1*u2) -> num u1 -> num u2

main x = let f = div x
         in (f this, f that)
```

In the program above, `div` is a typed division function. Let us assume that `x` gets type `num u` in the environment, for some unknown `u`. From type checking the body of `f` we get the constraint `u` $\sim$ `u1` $*$ `u2` for some unknown instantiations of `div`. If the unifier naïvely substitutes `u` away for `u1` $*$ `u2`, those variables become bound in the environment, and hence we are not allowed to apply `f` polymorphically in the body of the definition.

Kennedy found a technical fix, by exploiting the fact that units of measure happen to form an Abelian group, and adapting an algebraic normalization procedure to types. For example, the normal form type for `f` above is: `forall u. num u -> num (u/u1)` His technique is ingenious, but leads to a significant complexity burden in the inference algorithm. More seriously, it does not generalise because it relies on special algebraic properties of units of measure. In particular, his solution fails for arbitrary type functions.

6.5 Ambiguity

From our discussion in Section 5.6, it is not to be anticipated that any reasonable algorithm will achieve completeness with respect to the specification of Figure 5. In the rest of this section we sketch a modification of the typing rules that excludes programs that algorithmically would demonstrate these problems.

Our starting point is Example 4.

```
flop :: String -> String
flop s = show (read s)
```

The specification may instantiate the quantified variable of `show` to `Int` or `Bool`, which both could make the program typeable. However, the $Q\,;\Gamma \vdash$ `show (read s)` $: \tau \triangleright \emptyset$ judgement does not use any of the axioms of Q_0 (such as `Show Int` or `Show Bool`) and there exists a most general constraint Q for the expression `show (read s)`. That constraint is simply `Show` a for some type variable a.

In the specification, it is possible to detect whether a constraint and a type are the most general possible for an expression:

$$\frac{\begin{array}{c} Q\,;\Gamma \vdash e : \tau \triangleright P \quad dom(\theta)\#\Gamma \\ \forall Q_1, \tau_1, P_1, (Q_1\,;\Gamma \vdash e : \tau_1 \triangleright P_1) \Longrightarrow Q_1 \Vdash \theta Q \wedge \theta\tau{\sim}\tau_1 \end{array}}{Q\,;\Gamma \vdash^{max} e : \tau \triangleright P}\ \textsc{max}$$

The MAX rule ensures that that Q and τ are the most general constraint and type for the expression e in the environment Γ.

Having gotten "hold" of the most general constraint and type in the typing rules, the $\vdash^{R}$ judgement can be modified to require that the most general constraint inferred for an expression should be solvable by *some* substitution for the free variables of the constraint. Formally, there must exist a substitution θ such that $Q_0\,Q_g \Vdash \theta Q_{max}$, where Q_{max} is the most general constraint and Q_g is the given constraint. For example, for the expression `show (read s)` we would have either $Q_0 \Vdash [a \mapsto \texttt{Int}](\texttt{Show}\ a)$ or $Q_0 \Vdash [a \mapsto \texttt{Bool}](\texttt{Show}\ a)$.

So far we have not achieved much – how could we reject the program if we knew the most general constraint that it can be typed with? The next step is to observe that neither $a \sim \texttt{Int}$ nor $a \sim \texttt{Bool}$ follow from the constraint `Show` a. In other words, the constraint `Show` a is *solvable* but not *unambiguously solvable*. This idea is in the heart of the definition below.

1 Definition [Unambiguously solvable constraints]: A constraint Q is *solvable* in Q_0 with given constraints Q_g iff there exists a θ with $dom(\theta)\#ftv(Q_g)$ and such that $Q_0\,Q_g \Vdash \theta Q$. The constraint Q is *unambiguously* solvable if additionally $Q_0\,Q_g\,Q \Vdash \mathcal{E}_\theta$.

Hence, a constraint Q is unambiguously solvable in a set of given constraints and an axiom set whenever it is equivalent to a substitution for a set of variables that do not belong in the given constraints. A reasonable solver (such as the one described with the conditions in Definition 2) fully solves a constraint only if it is unambiguously solvable. As a reasonable completeness requirement we could ask for the inverse property, so that: a sound and complete solver solves a constraint iff it is unambiguously solvable. We may then modify the $\vdash^{R}$ judgement to require that:

> the principal constraint of the expression be unambiguously solvable in the current given constraints and axiom set.

As the issue of ambiguity is orthogonal to the subject of this paper, we leave the task of formally specifying these modifications as future work.

7. Related and future work

There is a very long line of work in Hindley-Milner derivatives, parameterized over various constraint domains (Jon92; Sul00; OSW99; SMZ99). Rémy and Pottier (PR05) give a comprehensive account of type inference for HM(X) (Hindley-Milner, parameterized over the constraint domain X). To our knowledge, our presentation is the first one that deals with local assumptions introduced by type signatures and data constructors, and where those local assumptions may include type equalities.

Simonet and Pottier study type inference for GADTs, where local GADT type equalities may be introduced as a result of pattern matching (SP07). They propose a solution that does generalization over local `let`-bound definitions, by abstracting over the full generated constraint. We have seen that this approach has practical disadvantages, though theoretically appealing and technically straightforward. Interestingly, since ML is call by value the constraints arising from a `let`-bound definition have to be satisfiable by *some* substitution, since the expression will be evaluated independently of whether it will be called or not. By contrast, in Haskell we may postpone the satisfiability check of the generated constraints all the way to the call sites of a definition. In the case of our previous work on type inference for GADTs (SJSV09) such a satisfiability check happens implicitly since at local `let`-bound definitions, the constraint generation procedure calls the solver to discharge the generated constraints by means of substitutions.

The pioneering work of Mark Jones on qualified types (Jon92) is closely related to our approach, except for the fact that we additionally have to deal with type equalities and local assumptions. The problem of ambiguity has been identified by many (Jon92; NP95) but, to our knowledge, there has never been a proposal for a type system that rejects programs with ambiguity. In the work of Mark Jones, a characterization of unambiguous types of the form $\forall\overline{a}\,.\,Q \Rightarrow \tau$ requires that the free type variables of Q must appear in τ. With the coming of type functions, it appears that these characterizations are no longer adequate.

Stuckey and Sulzmann (SS05) employ a more elaborate ambiguity condition on type signatures than Jones — and in fact one of the same flavour as Definition 1 in Section 6.5.

1 Definition [Unambiguous types]: A type $\forall\overline{a}\,.\,Q \Rightarrow \tau$ is unambiguous in Q_0, iff for some fresh set of variables $\overline{b}$ we have that

$$Q_0 \wedge Q \wedge (\overline{[a \mapsto b]}Q) \wedge (\tau \sim \overline{[a \mapsto b]}\tau) \Vdash \overline{a \sim b}$$

In other words, the equality between two instantiated types implies equality of instantiations.

For example, consider the type: $\forall ab.F\ a \sim b \Rightarrow \texttt{Int} \rightarrow a$ and a renaming $[a \mapsto a_1, b \mapsto b_1]$. Then we must show that

$$F\ a \sim b \wedge F\ a_1 \sim b_1 \wedge a \sim a_1 \Vdash (a \sim a_1) \wedge (b \sim b_1)$$

There exist even constraint-free types that are ambiguous. Take for example $\forall a.F\ a \rightarrow Int$. Assuming a renaming $[a \mapsto a_1]$, it does not follow that:

$$F\ a_1 \sim F\ a \Vdash a_1 \sim a$$

as type functions need not be injective. In practical terms, this means that we can never apply a function with that type to a value of type, say, F `Int`.

The precise connection between the Stuckey-Sulzmann condition and our notion of unambiguously solvable constraints (Section 6.5) is an interesting direction for future work.

Finally, ambiguity seems also related to the discussion about type functions and type classes defined under an "open" or "closed" world assumption (Sul00). Recently, there have been papers (SSS08; DPSS04; SSJC07; SPJCS08) that give the properties of constraint solvers for GADTs or type functions under open world assumptions, though we are not aware of previous high-level type system specifications that additionally deal with implications, such as the one that we present.

Future work Apart from addressing the ambiguity problems outlined in Section 5.6, we plan to provide an actual solver for all the major type system features of GHC (type classes, type functions, and GADTs) and show its soundness and completeness properties. Our goal is to have a type inference algorithm that additionally produces evidence (such as dictionary abstractions and applications, and coercions) and enjoys a high-level specification. The natural next step is to parameterize the type checker by external domain-specific solvers, towards "pluggable" type systems for Haskell.

Acknowledgements

Many thanks to Martin Sulzmann, Manuel Chakravarty, Andrew Kennedy, and the anonymous reviewers for their suggestions.

References

[CKP05] M. Chakravarty, Gabriele Keller, and S. Peyton Jones. Associated type synonyms. In *Proc. of ICFP '05*, pages 241–253, New York, NY, USA, 2005. ACM Press.

[CKPM05] M. Chakravarty, Gabriele Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *Proc. of POPL '05*, pages 1–13. ACM Press, 2005.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proc. of POPL '82*, pages 207–12, New York, 1982. ACM Press.

[DPSS04] G. J. Duck, S. Peyton Jones, P. J. Stuckey, and M. Sulzmann. Sound and decidable type inference for functional dependencies. In *Proc. of (ESOP'04)*, number 2986 in LNCS, pages 49–63. Springer-Verlag, 2004.

[HHPW96] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[HLvI03] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, August 2003. ACM Press.

[Jon92] M. P. Jones. *Qualified Types: Theory and Practice*. D.phil. thesis, Oxford University, September 1992.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In *Proc. of ESOP 2000*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[Ken96] AJ Kennedy. Type inference and equational theories. LIX RR/96/09, Ecole Polytechnique, September 1996.

[Mil78] R Milner. A theory of type polymorphism in programming. *JCSS*, 13(3), December 1978.

[NP95] Tobias Nipkow and Christian Prehofer. Type reconstruction for type classes. *Journal of Functional Programming*, 5(2):201–224, 1995.

[OSW99] M. Odersky, M. Sulzmann, and M Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[Pey03] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PR05] F. Pottier and D. Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[PVWS07] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17:1–82, January 2007.

[PVWW06] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proc. of ICFP'06*, pages 50–61. ACM Press, 2006.

[SJSV09] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proc. of ICFP'09*. ACM Press, 2009.

[SMZ99] M. Sulzmann, M. Mller, and C. Zenger. Hindley/milner style type systems in constraint form. Research Report ACRC-99-009, University of South Australia, School of Computer and Information Science, 1999.

[SP07] V. Simonet and F. Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Prog. Languages Systems*, 29(1), January 2007.

[SPJCS08] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 51–62, New York, NY, USA, 2008. ACM.

[SS05] P. J. Stuckey and M. Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1–54, 2005.

[SSJC07] T. Schrijvers, M. Sulzmann, S. Peyton Jones, and M. Chakravarty. Towards open type functions for Haskell. In O. Chitil, editor, *Proceedings of the 19th International Symposium on Implemantation and Application of Functional Languages*, pages 233–251, 2007.

[SSS08] M. Sulzmann, T. Schrijvers, and P. Stuckey. Type inference for GADTs via Herbrand constraint abduction. Report CW 507, Department of Computer Science, K.U.Leuven, Leuven, Belgium, January 2008.

[Sul00] M. Sulzmann. *A General Framework for Hindley/Milner Type Systems with Constraints*. PhD thesis, Yale University, Department of Computer Science, May 2000.

[SWJ06] M. Sulzmann, Jeremy Wazny, and P. J.Stuckey. A framework for extended algebraic data types. In *Proc. of FLOPS'06*, volume 3945 of *LNCS*, pages 47–64. Springer-Verlag, 2006.

[Tof90] M Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc 16th ACM Symposium on Principles of Programming Languages, Austin, Texas*. ACM, January 1989.

[Wri95] Andrew Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8:343–355, 1995.

[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.