Associated Type Synonyms

Manuel M. T. Chakravarty   Gabriele Keller
University of New South Wales
Programming Languages and Systems
{chak,keller}@cse.unsw.edu.au

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

Abstract

Haskell programmers often use a multi-parameter type class in which one or more type parameters are functionally dependent on the first. Although such functional dependencies have proved quite popular in practice, they express the programmer's intent somewhat indirectly. Developing earlier work on *associated data types*, we propose to add functionally-dependent types as type synonyms to type-class bodies. These *associated type synonyms* constitute an interesting new alternative to explicit functional dependencies.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages, Theory

Keywords Type classes; Type functions; Associated types; Type inference; Generic programming

1. Introduction

Suppose you want to define a family of containers, where the representation type of the container defines (or constrains) the type of its elements. For example, suppose we want containers supporting at least insertion, union, and a membership test. Then a list can contain elements of any type supporting equality; a balanced tree can only contain elements that have an ordering; and a bit-set might represent a collection of characters. Here is a rather natural type for the insertion function over such collections:

$$insert :: Collects\ c \Rightarrow Elem\ c \rightarrow c \rightarrow c$$

The type class $Collects$ says that $insert$ is overloaded: it will work on a variety of collection types c, namely those types for which the programmer writes an instance declaration for $Collects$. But what is $Elem$? The intent is obviously that $Elem\ c$ is the element type for collection type c; you can think of $Elem$ as a type-level function that transforms the collection type to the element type. However, just as $insert$ is non-parametric (its implementation varies depending on c), so is $Elem$. For example, $Elem\ [e]$ is e, but $Elem\ BitSet$ is $Char$.

The core idea of this paper is to extend traditional Haskell type classes with the ability to define *associated type synonyms*. In our example, we might define $Collects$ like this:

```
class Collects c where
  type Elem c                 -- Associated type synonym
  empty  :: c
  insert :: Elem c → c → c
  toList :: c → [Elem c]
```

The *type* definition says that c has an associated type $Elem\ c$, without saying what that type is. This associated type may then be used freely in the types of the class methods. An instance declaration gives an implementation for $Elem$, just as it gives an implementation for each method. For example:

```
instance Eq e ⇒ Collects [e] where
    {type Elem [e] = e; ...}
instance Collects BitSet where
    {type Elem BitSet = Char; ...}
instance (Collects c, Hashable (Elem c))
       ⇒ Collects (Array Int c) where
    {type Elem (Array Int c) = Elem c; ...}
```

Haskell aficionados will recognise that associated type synonyms attack exactly the same problem as *functional dependencies*, introduced to Haskell by Mark Jones five years ago [15], and widely used since then in surprisingly varied ways, many involving type-level computation. We discuss the relative strengths of the two approaches in detail in Section 6. It is too early to say which is "better"; our goal here is only to describe and characterise a new point in the design space of type classes.

Specifically, our contributions are these:

- We explore the utility and semantics of type synonym declarations in type classes (Section 2).
- We discuss the syntactic constraints necessary to keep type inference in the presence of associated type synonyms decidable (Section 3).
- We give a type system that supports associated type synonyms and allows an evidence translation to an explicitly typed core language in the style of System F (Section 4).
- We present a type inference algorithm that can handle the non-syntactic equalities arising from associated type synonyms; the algorithm conservatively extends Jones' algorithm for qualified types [12] (Section 5).

This paper is a natural development of, and is complementary to, our earlier work on *associated data types* [1], in which we allow a **class** declaration to define new algebraic data types. We discuss other related type systems—in particular, functional dependencies, HM(X), and ML modules—in detail in Sections 6 and 7. We developed a prototype implementation of the type checker, which we make available online.[1]

ICFP'05 September 26–28, 2005, Tallinn, Estonia.

---

[1] http://www.cse.unsw.edu.au/~chak/papers/CKP05.html

2. Applications of Associated Type Synonyms

We begin informally, by giving several examples that motivate associated type synonyms, and show what can be done with them.

2.1 Formatting: type functions compute function types

The implementation of a string formatting function whose type depends on a format specifier seems a natural application for dependent types and meta programming [26]. Although Danvy [4] demonstrated that Standard ML's type system is powerful enough to solve this problem, type functions enable a more direct solution [10], using an inductive definition instead of explicit continuation passing style. The following implementation with associated synonyms is based on [22]. Format specifiers are realised as singleton types:[2]

```
data I f = I f            -- Integer value
data C f = C f            -- Character value
data S f = S String f     -- Literal string


formatSpec :: S (I (S (C String)))
formatSpec = S "Int: " $ I $ S ", Char: " $ C $ "."
                 -- Example format: "Int: %d, Char: %c."
```

The singleton type declarations reflect the structure of a format specifier *value* in their *type*. Consequently, we can use a specifier's type to calculate an appropriate type for a *sprintf* function applied to that specifier. We implement this type level calculation by defining an associated synonym *Sprintf* in a class *Format* in the following way:

```
 class Format fmt where
   type Sprintf fmt
   sprintf′ :: String → fmt → Sprintf fmt
instance Format String where
   type Sprintf String = String
   sprintf′ prefix str    = prefix ++ str
instance Format a ⇒ Format (I a) where
   type Sprintf (I a)  = Int → Sprintf a
   sprintf′ prefix (I a) = λi. sprintf′ (prefix ++ show i) a
instance Format a ⇒ Format (C a) where
   type Sprintf (C a)  = Char → Sprintf a
   sprintf′ prefix (C a) = λc. sprintf′ (prefix ++ [c]) a
instance Format a ⇒ Sprintf (S a) where
   type Sprintf (S a)       = Sprintf a
   sprintf′ prefix (S str a) = sprintf′ (prefix ++ str) a
sprintf :: Format fmt ⇒ fmt → Sprintf fmt
sprintf = sprintf′ ""
```

New format-specifier types (such as I and S above) can be introduced by the programmer at any time, simply by defining the type, and giving a matching instance declaration; that is, the definition of *sprintf* is open, or extensible.

Notice how important it is that the associated type is a *synonym*: it is essential that *Sprintf fmt* is a function type, not a data type.

2.2 Generic data structures

The collections abstraction *Collects* from Section 1 is an example of a *generic data structure*—others include sequences, graphs, and so on. Several very successful C++ libraries, such as the Standard Template Library [29] and the Boost Graph Library [28], provide highly-parameterised interfaces to these generic data structures, along with a wide range of implementations of these interfaces with different performance characteristics. Recently, Garcia et al. [8] published a qualitative comparison of six programming languages when used for this style of programming. In their comparison Haskell, including multi-parameter type classes and functional dependencies, was rated very favourably, *except for its lack of support for associated types*.

Here is part of the interface to a graph library, inspired by their paper; although, we have simplified it considerably:

```
type Edge g = (Node g, Node g)
   -- We simplify by fixing the edge representation
class Graph g where
   type Node g
   outEdges :: Node g → g → [Edge g]
class Graph g ⇒ BiGraph g where
   inEdges :: Node g → g → [Edge g]
```

Using an associated type synonym, we can make the type of nodes, *Node g*, a function of the graph type g. Basic graphs only support traversals along outgoing edges, whereas bi-graphs also support going backwards by following incoming edges. A graph representation based on adjacency lists would only implement the basic interface, whereas one based on an adjacency matrix can easily support the bi-graph interface, as the following instances illustrate:

```
data AdjList v = AdjList [[v]]
instance Enum v ⇒ Graph (AdjList v) where
   type Node (AdjList v) = v
   outEdges v g = [(v, w) | w ← g!!fromEnum v]
type AdjMat = Array.Array (Int, Int) Bool
instance Graph AdjMat where
   type Node AdjMat = Int
   outEdges v g = let ((from, _), (to, _)) = bounds g
                  in [w | w ← [from..to], g!(v, w)]
instance BiGraph AdjMat where
   inEdges v g = let ((from, _), (to, _)) = bounds g
                 in [w | w ← [from..to], g!(w, v)]
```

By making *Edge*, as well as *Node*, an associated type synonym of *Graph* and by parameterising over traversals and the data structures used to maintain state during traversals, the above class can be made even more flexible, much as the Boost Graph Library, or the skeleton used as a running example by Garcia et al. [8].

3. The programmer's-eye view

In this section, we give a programmer's-eye view of the proposed language extension. Formal details follow later, in Section 4.

We propose that a type class may declare, in addition to a set of methods, a set of associated type synonyms. The declaration head alone is sufficient, but optionally a *default definition*—much like those for methods—may be provided. If no default definition is given, an optional kind signature may be used; otherwise, the result kind of a synonym application is assumed to be $\star$. An associated type synonym must be parametrised over all the type variables of the class, and these type variables must come first, and be in the same order as the class type variables.

Each associated type synonym introduces a new top-level type constructor. The kind of the type constructor is inferred as usual in Haskell; we also allow explicit kind signatures on type parameters:

```
class C a where
   type S a (k :: ⋆ → ⋆) :: ⋆
```

Instance declarations must give a definition for each associated type synonym of the class, unless the synonym has been given a default definition in the class declaration. The definition in an instance declaration looks like this:

```
instance C [a] where
   type S [a] k = (a, k a)
```

---

[2] The infix operator $f \,\$\, x$ in Haskell is function application $f\ x$ at a lesser precedence.

The part to the left of the " =" is called the *definition head*. The head must repeat the type parameters of the instance declaration exactly (here $[a]$); and any additional parameters of the synonym must be simply type variables (k, in our example). The overall number of parameters, called the synonym's *arity*, must be the same as in the class declaration. All applications of associated type synonyms must be saturated; i.e., supplied with as many type arguments as prescribed by their arity.

We omit here the discussion of toplevel data type declarations involving associated types, as we covered these in detail previously [1]. In all syntactic restrictions in this section, we assume that any *toplevel* type synonyms have already been replaced by their right-hand sides.

3.1 Equality constraints

Suppose we want to write a function $\mathit{sumColl}$ that adds up the elements of a collection with integer elements. It cannot have type

$$\begin{array}{ll} \mathit{sumColl} :: (\mathit{Collects}\ c) \Rightarrow c \rightarrow \mathit{Int} & \text{-- Wrong!} \\ \mathit{sumColl}\ c = \mathit{sum}\ (\mathit{toList}\ c) & \end{array}$$

because not all collections have Int elements. We need to constrain c to range only over Int-element collections. The way to achieve this is to use an *equality constraint*:

$$\begin{array}{l} \mathit{sumColl} :: (\mathit{Collects}\ c, \mathit{Elem}\ c = \mathit{Int}) \Rightarrow c \rightarrow \mathit{Int} \\ \mathit{sumColl}\ c = \mathit{sum}\ (\mathit{toList}\ c) \end{array}$$

As another example, suppose we wanted to merge two collections, perhaps with different representations, but with the same element type. Then again, we need an equality constraint:

$$\begin{array}{l} \mathit{merge} :: (\mathit{Collects}\ c1, \mathit{Collects}\ c2, \mathit{Elem}\ c1 = \mathit{Elem}\ c2) \\ \qquad \Rightarrow c1 \rightarrow c2 \rightarrow c2 \\ \mathit{merge}\ c1\ c2 = \mathit{foldr}\ \mathit{insert}\ c2\ (\mathit{toList}\ c1) \end{array}$$

Without loss of generality, we define an equality constraint to have the form $(S\ \overline{\alpha}\ \overline{\tau} = \upsilon)$, where S is an associated type synonym, $\overline{\alpha}$ are as many type variables as the associated class has parameters, and the $\overline{\tau}$ and υ are arbitrary monotypes. There is no need for greater generality than this; for example, the constraint $([S\ a] = [\mathit{Int}])$ is equivalent to $(S\ a = \mathit{Int})$; the constraint $([S\ a] = \mathit{Bool})$ is unsatisfiable; and $(a = \mathit{Int})$ can be eliminated by replacing a by Int. These restrictions are stronger than they would have to be. However, they allow us later on to characterise well-formed programs on a purely syntactical level.

3.2 Constraints for associated type synonyms

Does this type signature make sense?

$$\mathit{funnyFst} :: (\mathit{Elem}\ c, c) \rightarrow \mathit{Elem}\ c$$

Recall that Elem is a *partial* function at the type level, whose domain is determined by the set of instances of $\mathit{Collects}$. So it only makes sense to apply $\mathit{funnyFst}$ at a type that is an instance of $\mathit{Collects}$. Hence, we reject the signature, requiring you to write

$$\mathit{funnyFst} :: \mathit{Collects}\ c \Rightarrow (\mathit{Elem}\ c, c) \rightarrow \mathit{Elem}\ c$$

to constrain the types at which $\mathit{funnyFst}$ can be called. More precisely, each use of an associated type synonym in a programmer-written type signature gives rise to a class constraint for its associated class; and that constraint must be satisfied by the context of the type signature, or by an instance declaration, or a combination of the two. This validity check for programmer-supplied type annotations is conveniently performed as part of the kind checking of these annotations, as we will see in Section 4. Kind checking is only required for programmer-supplied type annotations, because inferred types will be well-kinded by construction.

3.3 Instance declarations

Given that associated type synonyms amount to functions on types, we need to restrict their definitions so that type checking remains tractable. In particular, they must be *confluent;* i.e., if a type expression can be reduced in two different ways, there must be further reduction steps that join the two different reducts again. Moreover, type functions must be *terminating;* i.e., applications must reach an irreducible normal form after a finite number of reduction steps. The first condition, confluence, is already standard on the level of values, but the second, termination, is a consequence of the desire to keep type checking decidable.

Similar requirements arise already for vanilla type classes as part of a process known as *context reduction.* In a declaration

$$\mathbf{instance}\ (\pi_1, \ldots, \pi_n) \Rightarrow C\ \tau_1 \cdots \tau_m$$

we call $C\ \tau_1 \cdots \tau_m$ the *instance head* and $(\pi_1, \ldots, \pi_n)$ the *instance context,* where each π_i is itself a class constraint. Such an instance declaration implies a *context reduction rule* that replaces the instance head by the instance context. The critical point is that the constraints π_i can directly or indirectly trigger other context reduction rules that produce constraints involving C again. Hence, we have recursive reduction rules and the same issues of confluence and termination as for associated type synonyms arise. Haskell 98 carefully restricts the formation rules for instance declarations such that the implied context reduction rules are confluent and terminating. It turns out, that we can use the same restrictions to ensure these properties for associated type synonyms. In the following, we discuss these restrictions, but go beyond Haskell 98 by allowing multi-parameter type classes. We will also see how the standard formation rules for instances affect the type functions induced by associated synonym definitions.

Restrictions on instance heads. Haskell 98 imposes the following three restrictions. Restriction (1): Heads must be *constructor-based;* i.e., the type patterns in the head may only contain variables and data type constructors, synonyms are not permitted. Restriction (2): Heads must be *specific;* i.e., at least one type parameter must be a non-variable term. Restriction (3): Heads must be *non-overlapping;* i.e., there may be no two declarations whose heads are unifiable.

Given that the heads of associated synonyms must repeat the type parameters of the instance head exactly, the above three restrictions directly translate to associated synonyms. Restriction (1) is familiar from the value level, and we will discuss Restriction (2) a little later. The value level avoids Restriction (3) by defining that the selection of equations proceeds in textual order (i.e., if two equations overlap, the textually earlier takes precedence). However, there is no clear notion of textual order for instance declarations, which may be spread over multiple modules.

Restrictions on instance contexts. Haskell 98 imposes one more restriction. Restriction (4): Instance contexts must be *decreasing*. More specifically, Haskell 98 requires that the parameters of the constraints π_i occurring in an instance context are variables. If we have multi-parameter type classes, we need to further require that these variable parameters of a single constraint are distinct. Restriction (4) and (2) work together to guarantee that each context reduction rule simplifies at least one type parameter. As type terms are finite, this guarantees termination of context reduction.

In the presence of associated types, we generalise Restriction (4) slightly. Assuming $\epsilon_1, \ldots, \epsilon_n$ are each either a type variable or an associated type applied to type variables, a context constraint π_i can either be a class constraint of the form $D\ \epsilon_1 \cdots \epsilon_n$ or be an equality constraint of the form $S\ \alpha_1 \cdots \alpha_m = \tau$.

The right-hand sides of the associated type synonyms of an instance are indirectly constrained by Restriction (4), as they may only contain applications of synonyms whose associated class appears in the instance context. So, if we have

$$\begin{array}{l} \mathbf{instance}\ (\pi_1, \ldots, \pi_n) \Rightarrow C\ \tau\ \mathbf{where} \\ \quad \mathbf{type}\ S_C\ \tau = [S_D\ \alpha] \end{array}$$

and S_D is associated with class D, then one of the π_i must be $D\ \alpha$. In other words, as a consequence of the instance context restriction, associated synonym applications must have parameters that are either distinct variables or other synonyms applied to variables. Hence, the reduction of associated synonym applications terminates for the same reason that context reduction terminates.

This might seem a little restrictive, but is in fact sufficient for most applications. Strictly speaking we, and Haskell 98, could be a bit more permissive and allow that if there are n occurrences of data type constructors in the type parameters of an instance head, each constraint in the instance context may have up to $n-1$ occurrences of data type constructors in its arguments. Moreover, we may permit repeated variable occurrences if the type checker terminates once it sees the same constraint twice in one reduction chain. Work on *term rewriting system (TRS)* [19] has identified many possible characterisations of systems that are guaranteed to be confluent and terminating, but the restrictions stated above seem to be a particularly good match for a functional language.

3.4 Ambiguity

This celebrated function has an ambiguous type:

$$\begin{array}{l} \mathit{echo} :: (\mathit{Read}\ a,\ \mathit{Show}\ a) \Rightarrow \mathit{String} \rightarrow \mathit{String} \\ \mathit{echo}\ s = \mathit{show}\ (\mathit{read}\ s) \end{array}$$

The trouble is that neither argument nor result type mention "a", so any call to *echo* will give rise to the constraints $(\mathit{Read}\ a,\ \mathit{Show}\ a)$, with no way to resolve a. Since the meaning of the program depends on this resolution, Haskell 98 requires that the definition is rejected as ambiguous.

The situation is much fuzzier when functional dependencies [15] are involved. Consider

$$\begin{array}{l} \mathbf{class}\ C\ a\ b \mid a \rightarrow b\ \mathbf{where} \\ \quad \ldots \\ \mathit{poss} :: (C\ a\ b,\ \mathit{Eq}\ b) \Rightarrow a \rightarrow a \end{array}$$

Is the type of poss ambiguous? It looks like it, because b is not mentioned in the type after the "$\Rightarrow$". However, because of the functional dependency, fixing a will fix b, so all is well. But the dependency may not be so obvious. Suppose class D has no functional dependency, but it has an instance declaration like this:

$$\begin{array}{l} \mathbf{class}\ D\ p\ q\ \mathbf{where} \\ \quad \ldots \\ \mathbf{instance}\ C\ a\ b \Rightarrow D\ [a]\ b\ \mathbf{where} \\ \quad \ldots \\ \mathit{poss}2 :: (D\ a\ b,\ \mathit{Eq}\ b) \Rightarrow a \rightarrow a \end{array}$$

Now, suppose $\mathit{poss}2$ is applied to a list of integers. The call will give rise to a constraint $(D\ [\mathit{Int}]\ t)$, which can be simplified by the instance declaration to a constraint $(C\ \mathit{Int}\ t)$. Now the functional dependency for C will fix t, and no ambiguity arises. In short, some calls to $\mathit{poss}2$ may be ambiguous, but some may not.

It does no harm to delay reporting ambiguity. No unsoundness arises from allowing even *echo* to be defined; but, in the case of *echo*, every single call will result in an ambiguity error, so it is better to reject *echo* at its definition. When the situation is less clear-cut, it does no harm to accept the type signature, and report errors at the call site. Indeed, the only reason to check types for ambiguity at all is to emit error messages for unconditionally-ambiguous functions at their definition rather than at their use.

Associated type synonyms are easier, because functionally-dependent types are not named with a separate type variable. Here is how class C and poss would be written using an associated synonym S instead of a functionally-dependent type parameter:

$$\begin{array}{l} \mathbf{class}\ C\ a\ \mathbf{where} \\ \quad \mathbf{type}\ S\ a \\ \mathit{poss} :: (C\ a,\ \mathit{Eq}\ (S\ a)) \Rightarrow a \rightarrow a \end{array}$$

So, just as in Haskell 98, a type is unconditionally ambiguous if one of its constraints mentions no type variable that is free in the value part of the type.

There is a wrinkle, however. Consider this function:

$$\mathit{poss}3 :: (C\ a) \Rightarrow S\ a \rightarrow S\ a$$

It looks unambiguous, since a is mentioned in the value part of the type, but it is actually unconditionally ambiguous. Suppose we apply $\mathit{poss}3$ to an argument of type Int. Can we deduce what a is? By no means! There might be many instance declarations for C that all implement S as Int:

$$\begin{array}{l} \mathbf{instance}\ \ldots \Rightarrow C\ \tau\ \mathbf{where} \\ \quad \mathbf{type}\ S\ \tau = \mathit{Int} \end{array}$$

(In fact, this situation is not new. Even in Haskell 98 we can have degenerate type synonyms such as

$$\mathbf{type}\ S\ a = \mathit{Bool}$$

which would render $\mathit{poss}3$ ambiguous.)

The conclusion is this: When computing unconditional ambiguity—to emit earlier and more-informative error messages—*we should ignore type variables that occur under an associated type synonym*. For $\mathit{poss}3$, this means that we ignore the a in $S\ a$, and hence, there is no occurrence of a left to the right of the double arrow, which renders the signature unconditionally ambiguous. An important special case is that of class method signatures: Each method must mention the class variable somewhere that is not under an associated synonym. For example, this declaration defines an unconditionally-ambiguous method *op*, and is rejected:

$$\begin{array}{l} \mathbf{class}\ C\ a\ \mathbf{where} \\ \quad \mathbf{type}\ S\ a \\ \quad \mathit{op} :: S\ a \rightarrow \mathit{Int} \end{array}$$

4. The Type System

In this section, we formalise a type system for a lambda calculus including type classes with associated type synonyms. This type system is based on Jones' *Overloaded ML (OML)* [13, 14] and is related to our earlier system for associated *data types* [1]. Like Haskell 98 [9, 7], our typing rules can be extended to give a type-directed translation of source programs into an explicitly-typed lambda calculus akin to the predicative fragment of System F. We omit these extended rules here, as the extension closely follows our earlier work on associated data types [1].

The key difference between type checking in the presence of associated data types compared to associated type synonyms is the treatment of type equality. In the conventional Damas-Milner system as well as in its extension by type classes with associated data types, type equality is purely syntactic—i.e, types are equal iff they are represented by the same term. When we add associated type synonyms, type equality becomes more subtle. More precisely, the equations defining associated type synonyms in class instances refine type equality by introducing non-free functions over types. The treatment of this richer notion of equality in a Damas-Milner system with type classes during type checking and type inference constitutes the main technical contribution of this paper.

4.1 Syntax

The syntax of the source language is given in Figure 1. 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. Moreover, we use comma to mean sequence extension as follows: $\overline{a}^n, a_{n+1} \triangleq \overline{a}^{n+1}$. Although we give the syntax of qualified and quantified types and constraints in an uncurried way, we also some-

Symbol Classes

$$
\begin{array}{lll}
\alpha, \beta, \gamma & \rightarrow & \langle\text{type variable}\rangle \\
T & \rightarrow & \langle\text{type constructor}\rangle \\
S^k & \rightarrow & \langle\text{associated type synonym, arity } k\rangle \\
D & \rightarrow & \langle\text{type class}\rangle \\
x, f, d & \rightarrow & \langle\text{term variable}\rangle
\end{array}
$$

Source declarations

$$
\begin{array}{llll}
pgm & \rightarrow & \overline{cls};\ \overline{inst};\ \overline{val} & \text{(whole program)} \\
cls & \rightarrow & \mathbf{class}\ \forall\alpha.\overline{D'\ \alpha} \Rightarrow D\ \alpha\ \mathbf{where} & \text{(class decl)} \\
 & & \quad tsig;\ vsig & \\
inst & \rightarrow & \mathbf{instance}\ \forall\overline{\alpha}.\phi\ \mathbf{where} & \text{(instance declaration)} \\
 & & \quad atype;\ val & \\
val & \rightarrow & x\ =\ e & \text{(value binding)} \\
tsig & \rightarrow & \mathbf{type}\ S^k\ \overline{\alpha}^k & \text{(assoc. type signature)} \\
vsig & \rightarrow & x :: \sigma & \text{(method signature)} \\
atype & \rightarrow & \mathbf{type}\ S^k\ \tau\ \overline{\beta}^{(k-1)}\ =\ \xi & \text{(assoc. type synonym)}
\end{array}
$$

Source terms

$$
e \quad \rightarrow \quad x \mid e_1\ e_2 \mid \lambda x.e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid e :: \sigma
$$

Source types

$$
\begin{array}{llll}
\tau, \xi & \rightarrow & T \mid \alpha \mid \tau_1\ \tau_2 \mid \eta & \text{(monotype)} \\
\eta & \rightarrow & S^k\ \overline{\tau}^k & \text{(associated type)} \\
\rho & \rightarrow & \overline{\pi} \Rightarrow \tau & \text{(qualified type)} \\
\sigma & \rightarrow & \forall\overline{\alpha}.\rho & \text{(type scheme)}
\end{array}
$$

Constraints

$$
\begin{array}{llll}
\pi^c & \rightarrow & D\ \tau & \text{(class constraint)} \\
\pi^= & \rightarrow & \eta = \tau & \text{(equality constraint)} \\
\pi & \rightarrow & \pi^c \mid \pi^= & \text{(simple constraint)} \\
\phi & \rightarrow & \overline{\pi} \Rightarrow \pi^c & \text{(qualified constraint)} \\
\theta & \rightarrow & \forall\overline{\alpha}.\phi \mid \forall\overline{\alpha}.\pi^= & \text{(constraint scheme)}
\end{array}
$$

Environments

$$
\begin{array}{llll}
\Gamma & \rightarrow & \overline{x{:}\sigma} & \text{(type environment)} \\
\Theta & \rightarrow & \overline{\theta} & \text{(instance environment)} \\
U & \rightarrow & \overline{\pi^=} & \text{(set of equality constraints)}
\end{array}
$$

Figure 1: Syntax of expressions and types

times use equivalent curried notation, thus:

$$
\begin{array}{rcl}
\overline{\pi}^n \Rightarrow \tau & \equiv & \pi_1 \Rightarrow \cdots \Rightarrow \pi_n \Rightarrow \tau \\
\overline{\tau}^n \rightarrow \xi & \equiv & \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \xi \\
\forall\overline{\alpha}^n.\rho & \equiv & \forall\alpha_1 \cdots \forall\alpha_n.\rho
\end{array}
$$

We accommodate function types $\tau_1 \rightarrow \tau_2$ by regarding them as the curried application of the function type constructor to two arguments, thus $(\rightarrow)\ \tau_1\ \tau_2$. We use $(\mathrm{Fv}\ x)$ to denote the free variables of a structure x, which maybe an expression, type term, or environment.

The unusual features of the source language all concern associated type synonyms. A **class** declaration may contain **type** declarations as well as method signatures, and correspondingly an **instance** declaration may contain **type** definitions as well as method implementations. These type synonyms are the *associated type synonyms* of the class, and are syntactically distinguished: S is an associated type synonym constructor, while T is a regular type constructor (such as lists or pairs). In the declaration of an associated type synonym, the type indexes come first. The *arity* of a type synonym is the number of arguments given in its defining $tsig$. The arity is given by a superscript to the constructor name, but we drop it when it is clear from the context. The syntax of types τ includes η, the saturated application of an associated type. Note that such a saturated application can be of higher kind, if the result kind κ in the defining $tsig$ is not $\star$.

In the syntax of Figure 1, and in the following typing rules, we make two simplifying assumptions to reduce the notational burden:

1. Each class has exactly one type parameter, one method, and one associated type synonym.
2. There are no default definitions, neither for methods nor synonyms. A program with default definitions can be rewritten into one without, by duplicating the defaults at instances not providing their own versions.
3. We elide all mention of kinds, as we exactly follow Jones' system of constructor classes [14].

Lifting the first restriction is largely a matter of adding (a great many) overbars to the typing rules.

4.2 Type checking

Figures 2 and 3 present the typing rules for our type system. Our formalisation is similar to [9] in that we maintain the context reduction rules as part of the instance environment. The main judgement has the conventional form $\Theta \mid \Gamma \vdash e : \sigma$ meaning "in type environment Γ, and instance environment Θ, the term e has type σ". Declarations are typed by Figure 3, where all the rules are standard for Haskell, except for Rule $(inst)$.

The instance environment Θ is a set of *constraint schemes* θ that hold in e. A constraint scheme θ takes one of two forms (Figure 1): it is either a *class constraint scheme* $\forall\overline{\alpha}.\phi$, or an *equality scheme* $\forall\overline{\alpha}.\pi^=$. The instance environment is populated firstly by class and instance declarations, which generate constraint schemes using the rules of Figure 3; and secondly by moving underneath a qualified type (rule $(\Rightarrow I)$ of Figure 2). The latter adds only a *simple constraint* π, which can be a *class constraint* π^c or an *equality constraint* $\pi^=$; these simple constraints are special cases of the two forms described earlier[3].

The typing rules are almost as for vanilla Haskell 98, with two major differences. The first is in the side conditions $\Theta \vdash \sigma$ that check the well-formedness of types, in rules $(\rightarrow I)$ and $(\forall E)$, for reasons we discussed in Section 3.2. The rules for this judgement are also in Figure 2. The judgement needs the instance environment Θ so that it can check the well-formedness of applications of associated-type applications (Rule (wf_{syn})), using the entailment judgement $\Theta \Vdash \theta$ described below. In the interest of brevity, the presented rules elide all mention of kinds, leaving only the well-formedness check that is distinctive to a system including associated types. More details concerning the implications of the judgement $\Theta \vdash \sigma$ are in our previous work [1]. It is worthwhile to note that Rule (sig) does not contain well-formedness judgement, although it mentions a user-supplied type. This type is also produced by the typing judgement in the hypothesis, and the judgement always produces well-formed types. So, the rule will never apply to a malformed type.

The second major difference is Rule $(conv)$, which permits type conversion between types τ_1 and τ_2 if we have $\Theta \Vdash \tau_1 = \tau_2$. The auxiliary judgement $\Theta \Vdash \theta$ defines *constraint entailment,* which in Haskell 98 only handles type classes, but which we extend here with additional rules for type equality constraints. These rules are also given in Figure 2 and include the four standard equality axioms (eq_{refl}), (eq_{symm}), (eq_{trans}), and (eq_{subst}). The last of these allows equal types to be wrapped in an arbitrary context: for example, if $\tau_2 = \tau_3$, then $\mathit{Tree}\ (\mathit{List}\ \tau_2)\ =\ \mathit{Tree}\ (\mathit{List}\ \tau_3)$.

---

[3] Rule $(\Rightarrow I)$, and the syntax of types, does not allow one to quantify over constraint *schemes*, an orthogonal and interesting possible extension[11].

$$\boxed{\Theta \mid \Gamma \vdash e : \sigma}$$

$$\frac{(x:\sigma) \in \Gamma}{\Theta \mid \Gamma \vdash x : \sigma}(\mathit{var}) \qquad \frac{\Theta_1 \mid \Gamma \vdash e_1 : \sigma_1 \qquad \Theta_2 \mid \Gamma[x:\sigma_1] \vdash e_2 : \sigma_2}{\Theta_1, \Theta_2 \mid \Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \sigma_2}(\mathit{let}) \qquad \frac{\Theta \mid \Gamma \vdash e : \tau_1 \qquad \Theta \Vdash \tau_1 = \tau_2}{\Theta \mid \Gamma \vdash e : \tau_2}(\mathit{conv})$$

$$\frac{\Theta \mid \Gamma[x{:}\tau_1] \vdash e_2 : \tau_2 \qquad \Theta \vdash \tau_1}{\Theta \mid \Gamma \vdash \lambda x.e_2 : \tau_1 \to \tau_2}(\to I) \qquad \frac{\Theta \mid \Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Theta \mid \Gamma \vdash e_2 : \tau_2}{\Theta \mid \Gamma \vdash e_1\ e_2 : \tau_1}(\to E)$$

$$\frac{\Theta, \pi \mid \Gamma \vdash e : \rho}{\Theta \mid \Gamma \vdash e : \pi \Rightarrow \rho}(\Rightarrow I) \qquad \frac{\Theta \mid \Gamma \vdash e : \pi \Rightarrow \rho \qquad \Theta \Vdash \pi}{\Theta \mid \Gamma \vdash e : \rho}(\Rightarrow E)$$

$$\frac{\Theta \mid \Gamma \vdash e : \sigma \qquad \alpha \notin \mathrm{Fv}\,\Theta \cup \mathrm{Fv}\,\Gamma}{\Theta \mid \Gamma \vdash e : \forall \alpha.\sigma}(\forall I) \qquad \frac{\Theta \mid \Gamma \vdash e : \forall \alpha.\sigma \qquad \Theta \vdash \tau}{\Theta \mid \Gamma \vdash e : [\tau/\alpha]\sigma}(\forall E) \qquad \frac{\Theta \mid \Gamma \vdash e : \sigma \qquad \mathrm{Fv}\,\sigma = \emptyset}{\Theta \mid \Gamma \vdash (e :: \sigma) : \sigma}(\mathit{sig})$$

$$\boxed{\Theta \Vdash \theta}$$

$$\frac{\theta \in \Theta}{\Theta \Vdash \theta}(\mathit{mono}) \qquad \frac{\Theta \Vdash \forall \alpha.\theta}{\Theta \Vdash [\tau/\alpha]\theta}(\mathit{spec}) \qquad \frac{\Theta \Vdash \pi \Rightarrow \phi \qquad \Theta \Vdash \pi}{\Theta \Vdash \phi}(\mathit{mp})$$

$$\frac{}{\Theta \Vdash \tau = \tau}(\mathit{eq}_{\mathit{refl}}) \qquad \frac{\Theta \Vdash \tau_2 = \tau_1}{\Theta \Vdash \tau_1 = \tau_2}(\mathit{eq}_{\mathit{symm}})$$

$$\frac{\Theta \Vdash \tau_1 = \tau_2 \qquad \Theta \Vdash \tau_2 = \tau_3}{\Theta \Vdash \tau_1 = \tau_3}(\mathit{eq}_{\mathit{trans}}) \qquad \frac{\Theta \Vdash [\tau_1/\alpha]\pi \qquad \Theta \Vdash \tau_1 = \tau_2}{\Theta \Vdash [\tau_2/\alpha]\pi}(\mathit{eq}_{\mathit{subst}})$$

$$\boxed{\Theta \vdash \sigma}$$

$$\frac{\Theta \vdash \tau_i \qquad \overline{\Theta \Vdash D\ \tau_0} \quad S \text{ is an associated type of } D}{\Theta \vdash S^k\ \tau_0\ \overline{\tau}^{(k-1)}}(\mathit{wf}_{\mathit{syn}}) \qquad \frac{\Theta \vdash \sigma \qquad \alpha \notin \mathrm{Fv}\,\Theta}{\Theta \vdash \forall \alpha.\sigma}(\mathit{wf}_{\mathit{spec}}) \qquad \frac{\Theta, \pi \vdash \rho}{\Theta \vdash \pi \Rightarrow \rho}(\mathit{wf}_{\mathit{mp}})$$

$$\frac{\Theta \vdash \tau_1 \qquad \Theta \vdash \tau_2}{\Theta \vdash \tau_1\ \tau_2}(\mathit{wf}_{\mathit{app}}) \qquad \frac{}{\Theta \vdash \alpha}(\mathit{wf}_{\mathit{var}}) \qquad \frac{}{\Theta \vdash T}(\mathit{wf}_{\mathit{con}})$$

Figure 2: Typing rules for expressions

$$\boxed{\Theta \mid \Gamma \vdash \mathit{cls} : \Theta, \Gamma}$$

$$\frac{\Theta, D\ \alpha \vdash \sigma}{\Theta \mid \Gamma \vdash \begin{array}{l} \mathbf{class}\ \forall \alpha.\overline{D'\ \alpha}^n \Rightarrow D\ \alpha\ \mathbf{where} \\ \quad \mathbf{type}\ S\ \alpha\ \overline{\beta} :: \kappa \\ \quad f :: \sigma \end{array} : [\overline{\forall \alpha.D\ \alpha \Rightarrow D'\ \alpha}^n], [f : \forall \alpha.D\ \alpha \Rightarrow \sigma]}(\mathit{cls})$$

$$\boxed{\Theta_c, \Theta_i \mid \Gamma \vdash \mathit{inst} : \Theta}$$

for each D' where $(\forall \delta.D\ \delta \Rightarrow D'\ \delta) \in \Theta_c$, we have $\Theta_c, \Theta_i \Vdash \forall \overline{\alpha}.\overline{\pi} \Rightarrow D'\ \tau$

$$\frac{\Theta_c, \overline{\pi} \vdash \xi \qquad (f : \forall \gamma.D\ \gamma \Rightarrow \sigma) \in \Gamma \qquad \Theta_c, \Theta_i, \overline{\pi} \mid \Gamma \vdash e : [\tau/\gamma]\sigma}{\Theta_c, \Theta_i \mid \Gamma \vdash \begin{array}{l} \mathbf{instance}\ \forall \overline{\alpha}.\overline{\pi} \Rightarrow D\ \tau\ \mathbf{where} \\ \quad \mathbf{type}\ S\ \tau\ \overline{\beta} = \xi \\ \quad f = e \end{array} : [\forall \overline{\alpha}.\overline{\pi} \Rightarrow D\ \tau, \forall \overline{\alpha}.S\ \tau\ \overline{\beta} = \xi]}(\mathit{inst})$$

$$\boxed{\Theta \mid \Gamma \vdash \mathit{val} : \Gamma} \qquad \boxed{\vdash \mathit{pgm}}$$

$$\frac{\Theta \mid \Gamma \vdash e : \sigma}{\Theta \mid \Gamma \vdash (x = e) : [x : \sigma]}(\mathit{val}) \qquad \frac{\begin{array}{c} \Theta = \Theta_c, \Theta_i \qquad \Gamma = \Gamma_c, \Gamma_v \\ \overline{\Theta \mid \Gamma \vdash \mathit{cls} : \Theta_c, \Gamma_c} \qquad \overline{\Theta \mid \Gamma \vdash \mathit{inst} : \Theta_i} \qquad \overline{\Theta \mid \Gamma \vdash \mathit{val} : \Gamma_v} \end{array}}{\vdash \overline{\mathit{cls}}; \overline{\mathit{inst}}; \overline{\mathit{val}}}$$

Figure 3: Typing rules for declarations

Much of this is standard for extensions of the Hindley-Milner system with non-syntactic type equality [24, 22, 21, 23, 32, 2]. Novel in our system is the integration of entailment of type class predicates with type equalities. In particular, our system allows equality schemes, such as

$$\forall a.\ \mathit{Sprintf}\ (I\ a) = \mathit{Int} \rightarrow \mathit{Sprintf}\ a$$

in the constraint context Θ of the typing rules. Equality schemes are introduced by Rule (inst) of the declaration typing rules from Figure 3. This rule turns the definitions of associated type synonyms into equality schemes, such as the $\mathit{Sprintf}$ scheme above, which are used in the inference rules of judgement $\Theta \Vdash \theta$ from Figure 2.

An important property of the system is that the well-formedness of types is invariant under the expansion of associated type synonyms. Specifically, Rule (inst) ensures that each equality scheme $\forall\overline{\alpha}.S\ \tau\ \overline{\beta} = \xi$ is paired with a class constraint scheme $\forall\overline{\alpha}.\overline{\pi} \Rightarrow D\ \tau$, where D is the class S is associated with. Hence, all the premises $\overline{\pi}$ for the validity of ξ are fulfilled whenever the equality is applicable. Rule (inst) ensures this with the premise $\Theta_c, \overline{\pi} \vdash \xi$. Let us consider an example: The Format class of Section 2.1 has an instance

$$\begin{array}{l} \textbf{instance}\ \mathit{Format}\ a \Rightarrow \mathit{Format}\ (I\ a)\ \textbf{where} \\ \quad \textbf{type}\ \mathit{Sprintf}\ (I\ a) = \mathit{Int} \rightarrow \mathit{Sprintf}\ a \end{array}$$

Hence, Rule (inst) adds the two constraint schemes

$$\begin{array}{l} \theta_C = \forall a.\mathit{Format}\ a \Rightarrow \mathit{Format}\ (I\ a) \\ \theta_= = \forall a.\ \mathit{Sprintf}\ (I\ a) = \mathit{Int} \rightarrow \mathit{Sprinft}\ a \end{array}$$

to the context Θ. Now consider an expression e of type

$$\mathit{Format}\ (I\ \mathit{String}) \Rightarrow \mathit{Char} \rightarrow \mathit{Sprintf}\ (I\ \mathit{String})$$

Note that the context $\mathit{Format}\ (I\ \mathit{String})$ is required for the type to be well-formed according to judgement $\Theta \vdash \sigma$. In order to remove the constraint $\mathit{Format}\ (I\ \mathit{String})$ from the type of e by Rule $(\Rightarrow E)$, we need to have $\Theta \Vdash \mathit{Format}\ (I\ \mathit{String})$, which according to θ_C and Rule (mp) requires $\Theta \Vdash \mathit{Format}\ \mathit{String}$.

The result of applying $\theta_=$ to $\mathit{Char} \rightarrow \mathit{Sprintf}\ (I\ \mathit{String})$, i.e., $\mathit{Char} \rightarrow \mathit{Int} \rightarrow \mathit{Sprintf}\ \mathit{String}$, is well-formed only if $\Theta \Vdash \mathit{Format}\ \mathit{String}$ holds. That it holds is enforced by θ_C.

There are two more noteworthy points about Rule (inst). Firstly, the rule checks that the superclass constraints of the class D are fulfilled in the first premise. Secondly, when checking the validity of the right-hand side of the associated synonym, namely ξ, we assume only Θ_c (the superclass environment). If we would add Θ_i, we would violate Restriction (4) of Section 3.3 and potentially allow non-terminating synonym definitions.

5. Type Inference

The addition of Rule (conv) to type expressions and of equality axioms to constraint entailment has a significant impact on type inference. Firstly, we need to normalise type terms involving associated types according to the equations defining associated types. Secondly, type terms involving type variables often cannot be completely normalised until some type variables are further instantiated. Consequently, we need to extend the standard definition of unification in the Hindley-Milner system to return partially-solved equality constraints in addition to a substitution.

To illustrate this, reconsider the definition of the class $\mathit{Collects}$ with associated type Elem from Section 1. The unification problem $(\mathit{Int},\ a) = (\mathit{Elem}\ c,\ \mathit{Bool})$ implies the substitution $[\mathit{Bool}/a]$, but also the additional constraint $\mathit{Int} = \mathit{Elem}\ c$. We cannot decide whether the latter constraint is valid without knowing more about c, so we call such constraints *pending equality constraints.* In the presence of associated types, the type inference algorithm has to maintain pending equality constraints together with class predicates.

In this section, after fixing some constraints on source programs, we will first discuss a variant of Hindley-Milner type inference with predicates. The inference algorithm depends on a unification procedure that includes type normalisation (Section 5.3). Afterwards, we will use the unification procedure to test the subsumption relation of type schemes (Section 5.4). We conclude the section with some formal properties of our inference system for associated type synonyms.

5.1 Well-formed programs

To achieve *decidable* type inference computing principal types, we impose the following restrictions on the source programs as produced by Figure 1 (cf., Restrictions (1) to (4) in Section 3):

- Instance heads must be constructor-based, specific, and non-overlapping.
- Instance contexts must be decreasing.

To guarantee a *coherent* (i.e., unambiguous) translation of well-typed source programs to an *explicitly-typed* language in the style of System F, which implements type classes by dictionaries, we require two further constraints:

- Equality constraints (in programmer-supplied type annotations) must be of the form $S\ \alpha\ \overline{\tau} = \xi$ (i.e., the first argument must be a type variable).
- If $\sigma \equiv (\forall\overline{\alpha}.\overline{\pi} \Rightarrow \tau)$ is a method signature in a class declaration for $D\ \beta$, we require that $\beta \notin \mathrm{Fv}\,\overline{\pi}$.

(These restrictions are not necessary if the semantics of the program is given by translation to an untyped intermediate language, or perhaps one with a richer type system than System F; see for example [30].) Finally, to stay with Haskell's tradition of rejecting *unconditionally-ambiguous* type signatures, in the sense of Section 3.4, we require two more constraints:

- If $\sigma \equiv (\forall\overline{\alpha}.\overline{\pi} \Rightarrow \tau)$ is a method signature in a class declaration for $D\ \beta$, we require that $\beta \in \mathrm{Fixv}\,\sigma$.
- Similarly, we require of all signatures $e :: \forall\overline{\alpha}.\rho$ that $\overline{\alpha} \cap \mathrm{Fv}\,\rho \subseteq \mathrm{Fixv}\,\rho$.

Here $\mathrm{Fixv}\,\sigma$, the set of *fixed variables* of a signature σ, is defined as follows:

$$\begin{array}{lcl} \mathrm{Fixv}\ T & = & \{\} \\ \mathrm{Fixv}\ \alpha & = & \{\alpha\} \\ \mathrm{Fixv}\ (\tau_1\ \tau_2) & = & \mathrm{Fixv}\,\tau_1 \cup \mathrm{Fixv}\,\tau_2 \\ \mathrm{Fixv}\ \eta & = & \{\} \\ \mathrm{Fixv}\ ((\eta = \tau) \Rightarrow \rho) & = & \mathrm{Fixv}\,\tau \cup \mathrm{Fixv}\,\rho \\ \mathrm{Fixv}\ (D\ \tau \Rightarrow \rho) & = & \mathrm{Fixv}\,\rho \\ \mathrm{Fixv}\ (\forall\alpha.\sigma) & = & \mathrm{Fixv}\,\sigma \setminus \{\alpha\} \end{array}$$

Intuitively, the fixed variables of a type (or signature) are those free variables that will be constrained when we unify the type (or type component of the signature) with a ground type; provided it matches.

A program that meets all of the listed constraints is *well-formed*. The declaration typing rules of Figure 3 determine, for a given program, the validity of a *program context* Θ_P and typing environment Γ. Both are inputs to type inference for expressions and, if they are without superfluous elements, we call them well-formed. As in the typing rules, we implicitly assume all types to be well-kinded. It is straight-forward to add the corresponding judgements to the presented rules.

We call an expression well-formed if any signature annotation of the form $(e :: \sigma)$ obeys the listed constraints. For the rest of the paper, we confine ourselves to well-formed programs, program contexts, typing environments, and expressions. This in particular means that the rewrite system implied by the equality schemes in a program context Θ_P is confluent and terminating.

5.2 Type inference

We begin with type inference for type classes with associated types. Figure 4 displays the rules for the inference judgement $\Theta, U \mid T\Gamma \overset{\mathrm{W}}{\vdash} e : \tau$. Given a type environment Γ and an expression e as inputs, it computes the following outputs (1) a set of class constraints Θ, (2) a set of pending equality constraints U, (3) a substitution T, and (4) a monotype τ. The judgement implicitly also depends on the program context Θ_P, initially populated by the instance declarations and remains constant thereafter. Because **class** and **instance** declarations are, in effect, explicitly typed, their type inference rules are extremely similar to those in Figure 3; so, we do not give them separately here.

Our inference rules generalise the standard rules for Haskell in two ways: (1) the inference rules maintain a set of equality constraints U and (2) unification produces a set of pending equality constraints in addition to a substitution. Let us look at both aspects in some more detail.

Type inference for Haskell needs to maintain a set of Θ of constraints, called the *constraint context.* In Haskell 98, the constraint context is a set of type class predicates. We extend the context to also include equality constraints, just as in the typing rules. However, as we need to treat these two kinds of constraints differently during the inference process, we partition the overall context into the two subsets Θ and U denoting the class constraints and equality constraints, respectively. Hence, the phrase Θ, U to the left of the vertical bar in the inference judgement captures the whole context. In particular, Rules (var_W) and (sig_W) result in contexts $[\overline{\beta}/\overline{\alpha}]\overline{\pi}$, which are implicitly partitioned into the two components Θ and U in the hypotheses of other inference rules. During generalisation by the function $\mathrm{Gen}(\Gamma; \rho)$ in Rule (let_W) and (sig_W), both class and equality constraints are moved into the signature context of the produced signature.

In Rule $(\rightarrow E_W)$, the two sets of equality constraints U_1 and U_2 are both extracted out of the constraint context, to be fed to the unification process in conjunction with the new equality constraint $T_2\tau_1 = \tau_2 \rightarrow \alpha$. Unification simplifies the equalities as far as possible, producing, in addition to a substitution R, a set of pending equality constraints U. This set is used in the conclusion of the inference rule.

Rules $(\rightarrow E_W)$ and (sig_W) make use of two auxiliary judgements for unification and subsumption, respectively. Both of these judgements depend on the program context Θ_P and are the subject of the following two subsections.

5.3 Unification

Associated type synonyms extend Haskell 98's purely syntactic type equality; e.g, *Elem BitSet* $=$ *Char* holds under the definitions for the class *Collects* discussed before. To handle non-syntactic equalities during unification, we exploit the properties of well-formed programs. In particular, the type functions in well-formed programs are confluent and terminating; i.e., type terms have unique normal forms. Hence, we can determine whether two type terms are equal by syntactically comparing their normal forms.

The judgement $\Theta \Vdash \tau \leadsto \tau'$, whose inference rules are given by Figure 5, defines a *one-step reduction relation* on type terms under a constraint environment Θ. The reduction relation on types is used by the *one-step unification judgement* $\Theta \Vdash_U \tau_1 = \tau_2 \leadsto U; R$. This judgement holds for an equality $\tau_1 = \tau_2$ iff the corresponding unification problem can be reduced to a simpler unification problem in the form of a set of equality constraints U and a substitution R. The symbol $\bullet$ in the inference rules represents an empty set of equality constraints and the identity substitution, respectively. The repeated application of one-step unification, as performed by the reflexive and transitive closure $\Theta \Vdash_U U \leadsto^* U'; R$, reduces a set of equality constraints U to a set of pending equality constraints U' and a substitution R. If U' cannot be reduced any further, we also write $\Theta \Vdash_U U \leadsto^! U'; R$, hence turning the transitive closure of one-step unification into a deterministic function. Note that, due to the syntactic restrictions on τ, Rule (app_U) and (red_U) are not overlapping.

Unification is performed under a constraint environment Θ only because this environment is required by associate type synonym expansion. It is easy to see that, where two types τ_1 and τ_2 are free of associated synonyms, the relation $\bullet \Vdash_U \{\tau_1 = \tau_2\} \leadsto^! \bullet; R$ coincides with standard syntactic unification as employed in type inference for vanilla Haskell 98.

Just like Jones [14], we only need first-order unification despite the presence of higher-kinded variables, as we require all applications of associated type synonyms to be saturated.

Soundness of unification. The judgements of Figure 5 enjoy the following properties.

LEMMA 1 (Soundness of type reduction). *Given a well-formed constraint environment Θ and a type τ with $\Theta \vdash \tau$, we have that $\Theta \Vdash \tau \leadsto \tau'$ implies $\Theta \Vdash \tau = \tau'$.*

LEMMA 2 (Soundness of one-step unification). *Given a well-formed constraint environment Θ, the judgement $\Theta \Vdash_U \tau_1 = \tau_2 \leadsto U; R$ implies that $\Theta \Vdash R(\tau_1 = \tau_2)$ iff $\Theta \Vdash U$.*

THEOREM 1 (Soundness of unification). *Given a well-formed constraint environment Θ and a set of equalities U, then $\Theta \Vdash_U U \leadsto^* U'; R$ implies that $\Theta \Vdash RU$ iff $\Theta \Vdash U'$.*

The proofs proceed by rule induction. Moreover, Theorem 1 requires that "$\Theta \Vdash R\theta_1$ iff $\Theta \Vdash R\theta_2$" follows from "$\Theta \Vdash \theta_1$ iff $\Theta \Vdash \theta_2$" for any substitution R, which is a basic property of constraint entailment.

5.4 Subsumption

To handle type annotations, such as in expressions of the form $e :: \sigma$, during type inference, we need to have an algorithm deciding type subsumption. We say a type scheme σ_1 *subsumes* σ_2, written $\sigma_1 \leq \sigma_2$ iff any expression that can be typed as σ_1 can also be typed as σ_2.

The subsumption check can be formulated as a constraint entailment problem in the presence of equality constraints. If we want to check

$$(\forall\overline{\alpha_1}.\overline{\pi_1} \Rightarrow \tau_1) \leq (\forall\overline{\alpha_2}.\overline{\pi_2} \Rightarrow \tau_2),$$

in the context of a program context Θ_P, we need to demonstrate that, given a set of new type constructors $\overline{T}$ of the appropriate kind, there exists a substitution $R = [\overline{\xi}/\overline{\alpha_1}]$, such that

$$\Theta_P, [\overline{T}/\overline{\alpha_2}]\overline{\pi_2} \Vdash R\overline{\pi_1}, R\tau_1 = [\overline{T}/\overline{\alpha_2}]\tau_2$$

where we define the entailment of a set of predicates $\Theta \Vdash \overline{\theta}$ as the conjunction of all $\Theta \Vdash \theta_i$.

Given that we implement the entailment of equalities by unification during type inference, it is not surprising that we can implement subsumption via a combination of unification and the usual backchaining approach to entailment; a process called *context reduction* when used to simplify type class contexts. In Figure 6, we define the procedure computing entailment using the same structure as we used for the unification procedure: Firstly, we define a one-step entailment judgement $\Theta \Vdash_E \pi \leadsto P; R$ that reduces a constraint π (which may be a class constraint or equality constraint) to a simpler set of constraints P and a substitution R.

$$\boxed{\Theta, U \mid T\Gamma \overset{\mathrm{W}}{\vdash} e : \tau}$$

$$\frac{(x : \forall\overline{\alpha}.\overline{\pi} \Rightarrow \tau) \in \Gamma \qquad \overline{\beta}\text{ fresh}}{[\overline{\beta}/\overline{\alpha}]\overline{\pi} \mid \Gamma \overset{\mathrm{W}}{\vdash} x : [\overline{\beta}/\overline{\alpha}]\tau}(var_W)$$

$$\frac{\Theta_1, U_1 \mid T_1\Gamma \overset{\mathrm{W}}{\vdash} e_1 : \tau_1 \qquad \Theta_2, U_2 \mid T_2(T_1\Gamma) \overset{\mathrm{W}}{\vdash} e_2 : \tau_2 \qquad \begin{array}{c}\alpha\text{ fresh}\\ \Theta_P \Vdash_U U_1, U_2, (T_2\tau_1 = \tau_2 \to \alpha) \leadsto^{!} U; R\end{array}}{R(T_2\Theta_1, \Theta_2), U \mid (RT_2T_1)\Gamma \overset{\mathrm{W}}{\vdash} e_1\ e_2 : R\alpha}(\to E_W)$$

$$\frac{\Theta, U \mid T\Gamma[x : \alpha] \overset{\mathrm{W}}{\vdash} e : \tau \qquad \alpha\text{ fresh}}{\Theta, U \mid T\Gamma \overset{\mathrm{W}}{\vdash} \lambda x.e : T\alpha \to \tau}(\to I_W)$$

$$\frac{\Theta_1, U_1 \mid T_1\Gamma \overset{\mathrm{W}}{\vdash} e_1 : \tau_1 \qquad \Theta_2, U_2 \mid T_2(T_1\Gamma[x : \sigma]) \overset{\mathrm{W}}{\vdash} e_2 : \tau_2 \qquad \sigma = \mathrm{Gen}(T_1\Gamma; (\Theta_1, U_1) \Rightarrow \tau_1)}{\Theta_2, U_2 \mid (T_2T_1)\Gamma \overset{\mathrm{W}}{\vdash} (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) : \tau_2}(let_W)$$

$$\frac{\Theta, U \mid T\Gamma \overset{\mathrm{W}}{\vdash} e : \tau_1 \qquad \sigma = \mathrm{Gen}(T\Gamma; (\Theta, U) \Rightarrow \tau_1) \qquad \Theta_P \Vdash \sigma \le (\forall\overline{\alpha}.\overline{\pi} \Rightarrow \tau_2) \quad \begin{array}{c}\overline{\beta}\text{ fresh}\\ \mathrm{Fv}\,(\overline{\pi} \Rightarrow \tau_2) \subseteq \overline{\alpha}\end{array}}{[\overline{\beta}/\overline{\alpha}]\overline{\pi} \mid T\Gamma \overset{\mathrm{W}}{\vdash} (e :: \forall\overline{\alpha}.\overline{\pi} \Rightarrow \tau_2) : [\overline{\beta}/\overline{\alpha}]\tau_2}(sig_W)$$

$$\mathrm{Gen}(\Gamma; \rho) = \forall\overline{\alpha}.\rho \qquad \text{, where } \overline{\alpha} = \mathrm{Fv}\,\rho \setminus \mathrm{Fv}\,\Gamma$$

Figure 4: Type inference rules

$$\boxed{\Theta \Vdash \tau \leadsto \tau'}$$

$$\frac{\Theta \Vdash \tau_1 \leadsto \tau_1'}{\Theta \Vdash \tau_1\ \tau_2 \leadsto \tau_1'\ \tau_2}(lapp_R) \qquad \frac{\Theta \Vdash \tau_2 \leadsto \tau_2'}{\Theta \Vdash \tau_1\ \tau_2 \leadsto \tau_1\ \tau_2'}(rapp_R)$$

$$\frac{1 \le i \le k \qquad \Theta \Vdash \tau_i \leadsto \tau_i' \qquad \tau_j' = \tau_j, \forall j \ne i}{\Theta \Vdash S^k\ \overline{\tau}^k \leadsto S^k\ \overline{\tau'}^k}(arg_R) \qquad \frac{(\forall\overline{\alpha}.\eta = \tau) \in \Theta}{\Theta \Vdash [\overline{\xi}/\overline{\alpha}]\eta \leadsto [\overline{\xi}/\overline{\alpha}]\tau}(syn_R)$$

$$\boxed{\Theta \Vdash_U \tau_1 = \tau_2 \leadsto U; R}$$

$$\frac{}{\Theta \Vdash_U \tau = \tau \leadsto \bullet; \bullet}(refl_U) \qquad \frac{\Theta \Vdash_U \tau_2 = \tau_1 \leadsto U; R}{\Theta \Vdash_U \tau_1 = \tau_2 \leadsto U; R}(symm_U) \qquad \frac{\alpha \notin \mathrm{Fv}\,\tau}{\Theta \Vdash_U \alpha = \tau \leadsto \bullet; [\tau/\alpha]}(var_U)$$

$$\frac{}{\Theta \Vdash_U \tau_1\ \tau_2 = \tau_1'\ \tau_2' \leadsto \{\tau_1 = \tau_1', \tau_2 = \tau_2'\}; \bullet}(app_U) \qquad \frac{\Theta \Vdash \eta \leadsto \tau_2}{\Theta \Vdash_U \eta = \tau_1 \leadsto \{\tau_2 = \tau_1\}; \bullet}(red_U)$$

$$\boxed{\Theta \Vdash_U U \leadsto^* U'; R}$$

$$\frac{}{\Theta \Vdash_U U \leadsto^* U; \bullet} \qquad \frac{\Theta \Vdash_U \tau_1 = \tau_2 \leadsto U'; R' \qquad \Theta \Vdash_U R'U, U' \leadsto^* U''; R''}{\Theta \Vdash_U U, \tau_1 = \tau_2 \leadsto^* U''; R''R'}$$

Figure 5: Unification in the presence of type functions

$$\boxed{\Theta \Vdash_E \pi \leadsto P; R}$$

$$\frac{(\forall\overline{\alpha}.\overline{\pi} \Rightarrow D\ \tau) \in \Theta}{\Theta \Vdash_E [\overline{\xi}/\overline{\alpha}]D\ \tau \leadsto [\overline{\xi}/\overline{\alpha}]\overline{\pi}; \bullet}(bchain_E) \qquad \frac{\Theta \Vdash \tau \leadsto \tau'}{\Theta \Vdash_E D\ \tau \leadsto \{D\ \tau'\}; \bullet}(red_E) \qquad \frac{\Theta \Vdash_U \tau_1 = \tau_2 \leadsto U; R}{\Theta \Vdash_E \tau_1 = \tau_2 \leadsto U; R}(eq_E)$$

$$\boxed{\Theta \Vdash_E P \leadsto^* P'; R}$$

$$\frac{}{\Theta \Vdash_U P \leadsto^* P; \bullet} \qquad \frac{\Theta \Vdash_U \pi \leadsto P'; R' \qquad \Theta \Vdash_U R'P, P' \leadsto^* P''; R''}{\Theta \Vdash_U P, \pi \leadsto^* P''; R''R'}$$

Figure 6: Constraint entailment with equality constraints

Rule $(bchain_E)$ formalises standard backchaining, Rule (red_E) enables the reduction of associated type synonyms in class parameters, and Rule (eq_E) invokes one-step unification for equality constraints. Secondly, we define the reflexive and transitive closure $\Theta \Vdash_E P \leadsto^* P'; R$. Finally, we turn the closure into a deterministic function by requiring that for $\Theta_P \Vdash_E P \leadsto^! P'; R$, the resulting set of constraints P' cannot be reduced further.

Now, we can define subsumption as follows:

$$\frac{\begin{array}{c}\Theta_P, [\overline{T}/\overline{\alpha_2}]\overline{\pi_2} \Vdash_E \{\overline{\pi_1}, \tau_1 = [\overline{T}/\overline{\alpha_2}]\tau_2\} \leadsto^! \bullet; R \\ \overline{T} \text{ new type constructors}\end{array}}{\Theta_P \Vdash (\forall\overline{\alpha_1}.\overline{\pi_1} \Rightarrow \tau_1) \leq (\forall\overline{\alpha_2}.\overline{\pi_2} \Rightarrow \tau_2)}(\leq)$$

In other words, subsumption succeeds if context reduction can resolve the constraint set entirely.

Soundness of subsumption. The subsumption rule enjoys the following property.

LEMMA 3 (Soundness of subsumption). *Let Θ_P be a well-formed constraint environment, and $\sigma_1 = \forall\overline{\alpha_1}.\overline{\pi_1} \Rightarrow \tau_1$ and $\sigma_2 = \forall\overline{\alpha_2}.\overline{\pi_2} \Rightarrow \tau_2$ signatures with $\Theta_P \Vdash \sigma_1$ and $\Theta_P \Vdash \sigma_2$, where σ_2 is closed. Then, $\Theta_P \Vdash \sigma_1 \leq \sigma_2$ implies that, given a set of new type constructors $\overline{T}$ of the appropriate kind, there exists a substitution $R = [\overline{\xi}/\overline{\alpha_1}]$, such that $\Theta_P, [\overline{T}/\overline{\alpha_2}]\overline{\pi_2} \Vdash R\overline{\pi_1}, R\tau_1 = [\overline{T}/\overline{\alpha_2}]\tau_2$.*

The lemma is an immediate consequence of Theorem 1.

5.5 Formal properties of type inference

THEOREM 2 (Soundness of type inference). *Assume a well-formed program context Θ_P, typing environment Γ, and expression e. If $\Theta, U \mid T\Gamma \overset{W}{\vdash} e : \tau$, then $(\Theta_P, \Theta, U) \mid T\Gamma \vdash e : \tau$.*

In proving this theorem, we followed Jones [12] who introduces a *syntax-directed system* as a bridge between the declarative typing rules and the algorithmic inference rules, as first suggested by Clément at al. [3]. The tricky bit is to define the inference rule for function elimination in the syntax-directed system such that a correspondence to the inference system can be established. We define the rule as follows:

$$\frac{\begin{array}{c}\Theta \mid \Gamma \vdash e_1 : \tau_1 \qquad \Theta \mid \Gamma \vdash e_2 : \tau_2 \\ \Theta_P, \Theta \Vdash \tau_1 = \tau_2 \rightarrow \tau_1'\end{array}}{\Theta \mid \Gamma \vdash e_1\ e_2 : \tau_1'}(\rightarrow E_S)$$

In addition, we need to use Theorem 1 and Lemma 3.

We believe that our inference algorithm is complete and that we can obtain principal types by performing a further generalisation step on its outputs. However, we leave the proof to further work—which we plan to do by again following Jones' scheme for qualified types.

6. Functional Dependencies

Associated type synonyms cover much the same applications as functional dependencies. It is too early to say which of the two is "better"; hence, we will simply contrast the properties of both. The comparison is complicated by the existence of at least three variants of functional dependencies: (1) The system described by Mark Jones [15], (2) the generalised system implemented in the Glasgow Haskell Compiler (GHC) [31], and (3) an even more permissive system formalised by Stuckey & Sulzmann [30]. Like our associated type synonyms, the first two of these systems permit an implementation by a dictionary translation to an explicitly-typed language akin to System F. Stuckey and Sulzmann's system, based on an encoding into *constraint handling rules*, gains additional generality by giving up on a type-preserving translation and on separate compilation.

As pointed out by Duck et al. [6], GHC's system is not decidable, as it permits recursive instance declarations that, for some programs that should be rejected, leads to non-termination of type inference. Jones original system is decidable. The Stuckey-Sulzmann system, and the associated type synonyms we describe here, both ensure decidability by a suitable set of restrictions on the admissible constraint handling rules and associated type synonyms, respectively; both systems can handle more general rule sets if decidable inference is not required.

6.1 Type substitution property

We usually require from a type system that, if we can type an expression e with a type scheme $\forall\alpha.\sigma$, then we can also type it with an instance $[\tau/\alpha]\sigma$. Interestingly, this property does not hold for Jones' and GHC's version of functional dependencies. Here is an example:

```
 class C a b | a  →  b where
   foo :: a → b
instance C Bool Int where
   foo False =  0
   foo True  =  1
```

Under these definitions, consider the following three signatures of the function *bar*:

```
bar :: C a      b ⇒ a    → b     -- (1)... most general type
bar :: C Bool b ⇒ Bool → b     -- (2)... subst. instance
bar ::                 Bool → Int   -- (3)... apply fun-dep
bar = foo
```

With signature (1) and (3) the program type checks fine, but with (2) it is rejected by both GHC and Hugs. This is despite (2) clearly being a substitution instance of (1), and (2) and (3) being equivalent (i.e., each one subsumes the other). The Stuckey-Sulzmann system accepts all three signatures.

The corresponding declarations with associated types would be

```
class C a where
  type B a
  foo :: a → B a
instance C Bool where
  type B Bool = Int
  foo False    = 0
  foo True     = 1
```

and we get the following signatures:

```
bar :: C a ⇒ a    → B a      -- (1)... most general type
bar ::        Bool → B Bool   -- (2)... substitution inst.
bar ::        Bool → Int      -- (3)... reduce assoc syn.
bar = foo
```

Here we immediately get a ground constraint (which we omit as usual) by the substitution and all three signatures are accepted. The crucial difference is that with associated type synonyms, we do not need a type variable to represent the functionally dependent type.

6.2 Readability

As observed by Garcia et al. [8], if functional dependencies are used for the type of generic programming outlined in Section 2.2—on a full scale graph library and not the cut down version we used for illustration purposes—type classes quickly accumulate a significant number of parameters. This affects readability for reasons similar to why records with named fields are preferred over tuples of large arity. Associated types, just like records with named fields, refer to parameters by name and not by position. The same observation led to the addition of traits classes [20] in C++, which use a form of associated type synonyms in C++ classes.

As discussed in Section 3.4, if a class with functional dependencies is used in the context of an instance declaration of a class that does not have functional dependencies, the arising dependencies can be quite subtle and depend indirectly on the types of arguments a function is applied to.

6.3 Expressiveness

An immediate question is whether associated types can generally be encoded as functional dependencies and vise versa. This question does not have a simple answer. Duck et al. [6, Example 4] use the following class head:

```
class Zip a b c | a c → b, b c → a
```

It is not immediately clear how to translate this to associated types. However, as Oleg Kiselyov [17] demonstrates, the simpler class header

```
class Zip a b c | c → a b
```

is sufficient for this and similar applications. This second declaration can also be readily captured as an associated type synonym. In fact, it appears from inspecting publicly available code as if functional dependencies are normally used in a way that can be directly translated into associated types. In particular, even sophisticated uses of functional dependencies, such as HList [18] and a type safe evaluator, can be mirrored with associated types.

Conversely, a direct encoding of associated types with functional dependencies is not possible with Jones' and GHC's system due to the limitation discussed in Section 6.1. A more sophisticated encoding similar to the translation to an explicitly type passing language in the style of System F, as discussed in [1], does not require functional dependencies; plain multi-parameter type classes are sufficient. The Stuckey-Sulzmann system is, however, more powerful and seems to be able to directly handle all signatures involving associated types.

7. Other related work

Associated data types. In our previous work on associated data types [1], we allowed a **class** declaration to define a new data type (rather than a new type synonym). Is it necessary to have associated *type synonyms* as well as associated *data types*? At first, it might seem as if not—but both mechanisms are subtly different and cannot completely emulate each other.

Let's first try realising associated type synonyms as associated data types. Recall, for example, the list instance for $Collects$:

```
instance Eq e ⇒ Collects [e] where
  type Elem [e] = e
  insert x xs = x : xs
```

The elements of $[e]$ are of type e. It would be extremely inconvenient to have to wrap the elements of the list with a constructor, so that they were a fresh type:

```
instance Eq e ⇒ Collects [e] where
  data Elem [e] = ListElem e
  insert (ListElem x) xs = x : xs
```

In fact, it is not only inconvenient, as now signatures, such as

```
merge :: (Collects c1, Collects c2, Elem c1 = Elem c2)
           ⇒ c1 → c2 → c2
```

do not make any sense anymore. If $Elem$ is an associated data type, the equality constraint $Elem\ c1 = Elem\ c2$ is unsatisfiable. Moreover, the $Format$ class crucially rests on $Sprintf$ being a synonym; otherwise, the right-hand sides would not be function types anymore.

Now, how about the converse—can associated synonyms emulate associated data types? They almost can. Declarations of the form

```
class ArrayElem e where
  data Array e
  index :: Array e → Int → e
instance ArrayElem Int where
  data Array Int = ArrInt UnboxedIntArray
  index = indexUnboxedIntArray
```

can be rewritten as

```
class ArrayElem e where
  type Array e
  index :: Array e → Int → e
data ArrayInt = ArrInt UnboxedIntArray
instance ArrayElem Int where
  type Array Int = ArrayInt
  index = indexUnboxedIntArray
```

The translation becomes more involved for more complicated instances, but the principle remains the same. The difference between the two versions is the following. If we pass to the $index :: Array\ e \to Int \to e$ function a value of type $Array\ Int$, then for an associated data type, we know that $e = Int$. However, for an associated synonym, we cannot draw that conclusion. The equation $Array\ e = Array\ Int$ may have more than one solution, depending on the available $ArrayElem$ instances.

More generally, the introduction of a new type by an associated data type implies a *bijection* between collection type and element type, much like a *bi-directional* functional dependency. This is too strong a requirement for, for example, the class $Collects$. It needs a *uni-directional* dependence, and that is just what an associated synonym does. In particular, two collections represented differently may still contain elements of the same type.

Open versus closed type functions. Each associated type synonym constitutes a function over types, which is defined by the set of type synonym equations contained in the instance declarations of the class the synonym is associated with. Since Haskell permits to add new instances to an existing class at any point, associated type synonyms are *open* type functions—in the same way as class methods are open functions on the value level.

Using associated synonyms, we can define computations on the type level, and such computations, expressed using functional dependencies, have become quite popular with hard-core Haskell programmers. The standard example is addition on Peano numerals:

```
data Zero                                -- empty type
data Succ a                              -- empty type
 class Nat n where
  type Add n m :: ⋆
instance Nat Zero where
  type Add Zero m = m
instance Nat n ⇒ Nat (Succ n) where
  type Add (Succ n) m = Succ (Add n m)
```

This is, for example, useful to define bounded data types, such as fixed length vectors:

```
 class Nat n ⇒ VecBound n where
  data Vec n a
  appVec :: (VecBound m, VecBound (Add n m))
         ⇒ Vec n a → Vec m a → Vec (Add n m) a
instance VecBound Zero where
  data Vec Zero a = Nil
  appVec Nil ys = ys
instance VecBound n ⇒ VecBound (Succ n) where
  data Vec (Succ n) a = Cons a (Vec n a)
  appVec (Cons x xs) ys = Cons x (appVec xs ys)
```

The signature of *appVec* guarantees that the length of the result vector is the sum of the lengths of the argument vectors. The type *Vec* is an associated data type.

This definition of *Add* is unsatisfactory in two ways: (1) In its use of empty data types (i.e. ones with no constructors) and (2) the use of type classes to define *Add*. After all, we would not use type classes to define summation on Peano numerals on the value level! Instead, we would use an algebraic data type and a closed function. On the type level, this corresponds to an *algebraic kind definition* and a *closed type function,* which we might denote as follows:

```
kind Nat              = Zero | Succ Nat
type Add Zero     m   = m
type Add (Succ n) m   = Succ (Add n m)
```

This is not only more concise, it also captures our intentions much better: it ensures that *Add* is only applied to types of kind *Nat*, and it asserts that we will not extend the definition of *Nat* and *Add* in the future. Closed kind and type function definitions in a Haskell-like language have been proposed by Sheard for the Ωmega system [25].

Closed type functions are definitely useful—but so are open type functions! Indeed, open type functions are an extremely natural complement to the value-level open functions expressed by type classes. Although we have not addressed closed type functions in this paper, we would like to note that the type system and type inference algorithm presented in Sections 4 and 5 can handle closed type functions as well as open type functions. All we have to require is that the equations defining closed type functions are total and obey the same restrictions as those for open type functions, which we outlined in Section 3.3, so that they are confluent and terminating.

Functional logic overloading, HM(X), and constraint handling rules. Neubauer et al. [22] proposed a variant of type classes that define functions, instead of predicates, for use in type contexts. In their system, constraint environments do not contain any predicates anymore, but only equalities. Moreover, one type class defines exactly one function, whereas a type class can have multiple associated type synonyms in our system. Multiple associated synonyms are, for example, useful for a more fully fledged version of our *Graph* class. Neubauer et al. consider both open and closed functions as well as overlapping definitions, whereas we only consider open functions without overlapping definitions. An evidence translation for Neubauer et al.'s system is future work, whereas the translation of associated synonyms follows our earlier work on associated data types.

Neubauer et al. base their formal system on work by Odersky et al. [23], which introduces Hindley-Milner type inference parametrised with a wide range of constraint systems, called HM(X). The HM(X) system is based on type subsumption, of which equality is a special case.

Stuckey and Sulzmann [30] have developed the HM(X) framework further by fixing the constraint system to be based on *constraint handling rules (CHRs).* The resulting system is no longer restricted to regular equational theories and has been successfully used to encode a general version of functional dependencies. The price for this generality is the loss of the ability to translate to an explicit type-passing language, as is standard in dictionary translations of type classes. Moreover, separate compilation is problematic as whole-program information is required for the translation. Nevertheless, we expect to be able to implement type classes with associated types via HM(CHR) and indeed this seems to be an interesting item for future work.

Guarded recursive data types and generalised abstract data types. Generalised abstract data types (GADTs), also known as guarded recursive data types and first-class phantom types, constrain the construction of data types by equality constraints in data type definitions [32, 2, 16, 25]. These constraints are available in the corresponding branches of case expressions, which enables typing expressions that would otherwise have been too specific. There appears to be a connection between these equalities and equality constraints, but they are handled differently during type inference. Moreover, GADTs are closed and not associated with type classes.

ML modules. There is a significant overlap in functionality between Haskell type classes and Standard ML modules [5] . Associated types increase this overlap even more, and our equality constraints appear to bear a relation to sharing constraints. Nevertheless, there are also interesting differences, and both constructs have a large design space. We regard a more precise comparison of the two as interesting future work.

C++ traits classes. Typedefs in C++ class templates are of a similar nature as associated data types [20], and there are many C++ libraries [29, 28, 8] that demonstrate the usefulness of this mechanism. Type checking in C++ is, however, of an entirely different nature. In particular, template definitions are not separately type checked, but only after expansion at usage occurrences. Moreover, C++ does not have type inference. In exchange, C++ does not constrain the resulting type functions to be terminating or confluent, but permits arbitrary computations at compile time.

Siek and Lumsdaine [27] recently proposed an interesting combination of Haskell-style type classes with the C++ approach to generic programming. They propose a language F^{G} that adds a notion of *concepts* to System F and includes support for associated types of the same flavour as those introduced in the present paper. The main difference between Siek and Lumsdaine's work and ours is that Siek and Lumsdaine trade type inference for a more expressive type language.

8. Conclusions and further work

In this paper and its predecessor, we have explored the consequences of enabling a Haskell type class to contain **data** and **type** definitions, as well as method definitions. Doing so directly addresses Garcia et al's primary concern about large-scale programming in Haskell. It also fills out Haskell's existing ability to define open functions at the value level using type classes, with a complementary type-level facility.

There is clearly a big overlap between functional dependencies and associated type synonyms, and no language would want both. We do not yet have enough experience to know what difficulties (either of expressiveness or convenience), if any, programmers would encounter if functional dependencies were replaced by associated type synonyms—but we regard that as an attractive possibility.

Acknowledgements. We particularly thank Martin Sulzmann for his detailed and thoughtful feedback, which helped to identify and characterise several potential pitfalls. We are also grateful to Roman Leshchinskiy and Stefan Wehr who suggested significant improvements to the presentation and the formal system; moreover, Stefan Wehr throughly improved and extended the prototype type checker. We also thank the anonymous referees for their helpful comments. The first two authors have been partly funded by the Australian Research Council under grant number DP0211203.

References

[1] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In Martin Abadi, editor, *Conference Record of POPL 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–13. ACM Press, 2005.

[2] James Cheney and Ralf Hinze. First-class phantom types. Technical Report TR2003-1901, Cornell University, 2003.

[3] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: mini-ML. In *LFP '86: Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 13–27, New York, NY, USA, 1986. ACM Press.

[4] Olivier Danvy. Functional unparsing. *J. Funct. Program.*, 8(6):621–625, 1998.

[5] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 236–249, 2003.

[6] Gregory J. Duck, Simon Peyton Jones, Peter J. Stuckey, and Martin Sulzmann. Sound and decidable type inference for functional dependencies. In *ESOP;04*, LNCS. Springer-Verlag, 2004.

[7] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4+5), 2002.

[8] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 115–134. ACM Press, 2003.

[9] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996.

[10] Ralf Hinze. Formatting: A class act. *Journal of Functional Programming*, 13:935–944, 2003.

[11] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.

[12] Mark P. Jones. A theory of qualified types. In *ESOP'92: Symposium proceedings on 4th European symposium on programming*, pages 287–306, London, UK, 1992. Springer-Verlag.

[13] Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 1995.

[14] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), 1995.

[15] Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming (ESOP 2000)*, number 1782 in Lecture Notes in Computer Science. Springer-Verlag, 2000.

[16] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. `http://research.microsoft.com/Users/simonpj/papers/gadt/index.htm`, 2004.

[17] Oleg Kiselyov. Functions with the variable number (of variously typed) arguments. `http://okmij.org/ftp/Haskell/vararg-fn.lhs`, 2004.

[18] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Haskell '04: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107, New York, NY, USA, 2004. ACM Press.

[19] J. W. Klop. Term rewriting systems. In S. Abramsky D. M. Gabbay and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.

[20] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[21] Matthias Neubauer and Peter Thiemann. Type classes with more higher-order polymorphism. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 179–190, New York, NY, USA, 2002. ACM Press.

[22] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–244. ACM Press, 2002.

[23] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999.

[24] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, INRIA Rocquencourt, 1992.

[25] Tim Sheard. Languages of the future. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 116–119, New York, NY, USA, 2004. ACM Press.

[26] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. *ACM SIGPLAN Notices: PLI Workshops*, 37(12):60–75, 2002.

[27] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. *SIGPLAN Not.*, 40(6):73–84, 2005.

[28] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library User Guide and Reference Manual*. Addison-Wesley, 2001.

[29] A. A. Stepanov and M. Lee. The standard template library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, 1994.

[30] Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems*, 2004. To appear.

[31] The GHC Team. The Glasgow Haskell Compiler. `http://haskell.org/ghc/documentation.html`.

[32] 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.