

Associated Types with Class

Manuel M. T. Chakravarty Gabriele Keller
University of New South Wales & National ICT Australia
{chak,keller}@cse.unsw.edu.au

Simon Peyton Jones Simon Marlow
Microsoft Research Ltd., Cambridge, UK
{simonpj,simonmar}@microsoft.com

Abstract

In this paper, we explore an extension to Haskell type classes that allows a type class declaration to define *data types* as well as *values* (or methods). Similarly, an instance declaration gives a witness for such data types, as well as a witness for each method. It turns out that this extension directly supports the idea of a type-indexed type, and is useful in many applications, especially for self-optimising libraries that adapt their data representations and algorithms in a type-directed manner.

Crucially, just as Haskell’s existing type-class mechanism can be explained by translation into System F, so we can explain our extension by translation into System F, and we do so in full detail. This is a valuable property since it ensures that the addition of associated data types to an existing Haskell compiler leads to changes in the front end only.

1 Introduction

The efficient implementation of a parametrised data type sometimes requires a concrete representation that depends on its type parameter. For example, we may want to represent arrays of fixed-width integers differently to arrays of trees. Inventing some syntax, we might want to have a parameterized data type *Array* with several instantiations:

```
data Array e

data Array Int   = IntArray UIntArray
data Array Bool = BoolArray BitVector
data Array (a,b) = PairArray (Array a) (Array b)
```

Here, we represent an array of integers as an unboxed array, an array of booleans as a bit vector, and an array of pairs as a flattened pair of arrays. Consequently, an array of pairs of integers and booleans, *Array (Int, Bool)*, is represented as the pair of two unboxed arrays; i.e., as *PairArray UIntArray BitVector*. This specialised representation is more efficient, in terms of both space and runtime of typical operations, than a parametric representation as a boxed array of

pairs. Data types whose concrete representation depends on one or more type parameters are called *type-indexed* [17].

In this paper, we shall demonstrate that type-indexed types can be understood as class-local data type declarations, and that in fact this is a natural extension of Haskell’s type class overloading system. For example, the *Array* type above would be expressed as a local data type in a type class of array elements, *ArrayElem*:

```
class ArrayElem e where
data Array e
  index :: Array e → Int → e
```

The keyword **data** in a class introduces an *associated data type* definition – the type *Array* is associated with the class *ArrayElem*. (We are loosely following the terminology of Garcia et al. [10].) The representation of *Array* can vary on an instance-by-instance basis, but apart from that it behaves much like a top-level data type declaration.

We can now define instances of the *ArrayElem* class that give instantiations for the *Array* type:

```
instance ArrayElem Int where
data Array Int = IntArray UIntArray
  index (IntArray ar) i = indexUIntArray ar i
instance (ArrayElem a, ArrayElem b) ⇒
  ArrayElem (a, b) where
data Array (a, b) = PairArray (Array a) (Array b)
  index (PairArray ar br) i = (index ar i, index br i)
```

Together with the local data type *Array*, we have included a method *index* for indexing arrays. The type of *index* is

$$\text{index} :: \text{ArrayElem } e \Rightarrow \text{Array } e \rightarrow \text{Int} \rightarrow e$$

This signature makes both its dependence on the class *ArrayElem* as well as on the associated type *Array* explicit. In other words, for varying instantiations of the element type *e*, the concrete array representation on which *index* operates varies in dependence on the equations defining *Array*. This variation is more substantial than that of standard Haskell type classes, as the representation type of *Array* may change in a non-parametric way for different instantiations of the element type *e*. In other words, *type-indexed types permit an ad hoc overloading of types in the same way that standard type classes provide ad hoc overloading of values*.

To summarise, we make the following contributions:

- We introduce associated data type and type synonym declarations as a mechanism to implement type-indexed types, and demonstrate their usefulness with a number of motivating examples, notably self-optimising libraries (Sections 2 and 3).

- We show that associated data types are a natural extension of Haskell’s overloading system. We give typing rules for the new type system, and the evidence translation from source terms into System F (Sections 4 and 5). As most of the novel aspects of the system are confined to the System F translation, this enables a straightforward integration into existing Haskell compilers such as the Glasgow Haskell Compiler.

There is a great deal of related work on the subject of type-indexed types, which we review in detail in Section 7. Associated data type declarations narrow the gap between type classes in Haskell and modules in Standard ML; a class declaration, like an ML signature, gives describes an agglomerations of both types and values, while an instance declaration, like a structure or functor, specifies the concrete realisation of the abstraction. Moreover, associated types are obviously related to both intensional type analysis [14] and functional dependencies [21]. The latter deserve some additional comments, as they are already available in some Haskell systems. In Section 6, we discuss in detail why functional dependencies, in the form proposed by Jones [21], are inadequate for our motivating examples. Nevertheless, there are very close links: associated *data type* declarations are in many ways like bidirectional functional dependencies, while to get the effect of unidirectional functional dependencies, we also need associated *type synonyms* which we only sketch in this paper.

2 Motivation

The previous array example is representative for a whole class of applications of associated types, namely *self-optimising libraries*. These are libraries that, depending on their use, optimise their implementation—i.e., data representation and choice of algorithms—along lines determined by the library author. The optimisation process is guided by type instantiation, as in the *ArrayElem* class where the element type determines a suitable array representation. We shall discuss another instance of a representation optimisation by considering generic finite maps in Subsection 2.1. Then, in Subsection 2.2, we turn to the more sophisticated example of a generic graph library, where both data representation and algorithms vary in dependence on type parameters. The key feature of self-optimising libraries is that they do not merely rely on general compiler optimisations, but instead the library code itself contains precise instructions on how the library code is to be specialised for particular applications. Since the introduction of templates, this style of libraries has been highly successful in C++ with example such as the Standard Template Library [34], the Boost Graph Library [32], and the Matrix Template Library [33]. Work on generic programming in Haskell, also illustrates the need for type-dependent data representations [17, 16, 2].

In addition to implementing self-optimising libraries, associated types are also useful for abstract interfaces and other applications of functional dependencies. We shall discuss these in Subsection 2.3 and 2.4.

2.1 From Generalised Tries to Finite Maps

A nice example of a data structure whose representation changes in dependence of a type parameter, which was first discussed by Hinze [15] and subsequently used as an example for type-indexed types by Hinze, Jeuring, and Löh [17] in the context of Generic Haskell, are *generalised tries* or *generic finite maps*. Such maps change their representation in dependence on the structure of the key type used to index the map. In addition to the key type, finite maps are also parametrised by a value type that forms the co-domain of the map. While the representation of generic finite maps

depends on the type of keys, it is parametric polymorphic in the co-domain, which we call the value type v in the following definition:

```
class MapKey k where
  data Map k v
  empty :: Map k v
```

We express the different status of the key type k and value type v by only making k a class parameter; although the associated representation type $Map\ k\ v$ depends on both types. Assuming a suitable library implementing finite maps with integer trees, such as *Patricia trees* [30], we may provide an instance of *MapKey* for integer keys as follows:

```
instance MapKey Int where
  data Map Int v = MapInt (Patricia.Dict v)
  empty          = MapInt Patricia.emptyDict
  ...
```

In this instance, the different treatment of the key and value types is obvious in that we fix the key type for an instance, while still leaving the value type open. In other words, we can regard *Map k* as a type-indexed type constructor of kind $\star \rightarrow \star$.

As described in detail by Hinze [15], we can define generic finite maps on arbitrary algebraic data types by simply giving instances for *MapKey* for unit, product, and sum types. We do so as follows—leaving out the definitions of the class methods, which can be found in Hinze’s work:

```
instance MapKey () where
  data Map () v = MapUnit (Maybe v)

instance (MapKey a, MapKey b) =>
  MapKey (a, b) where
  data Map (a, b) v = MapPair (Map a (Map b v))

instance (MapKey a, MapKey b) =>
  MapKey (Either a b) where
  data Map (Either a b) v =
    MapEither (Map a v) (Map b v)
```

To use the class *MapKey* on any specific algebraic data type, we need a map it to its product/sum representation by means of an embedding-projection pair [18, 2].

2.2 Generic Graphs

The concept of traits has been introduced to C++ with the aim of reducing the number of parameters to templates [26]. Since then, generic programming based on templates with traits has been found to be useful for self-optimising libraries [32] where the choice of data representation as well as algorithms is guided by way of type instantiation.

This has led to an investigation of the support for this style of generic programming in a range of different languages by Garcia et al. [10]. The evaluation of Garcia et al., based on a comparative implementation of a graph library, concluded that Haskell has excellent support for generic programming with the exception of satisfactory support for associated types. The extension proposed in this paper tackles this shortcoming head on. Inspired by Garcia et al., we also use a class of graphs as an example.

```
class Graph g where
  data Edge g
  data Vertex g
  src    :: Edge g → g → Vertex g
  tgt    :: Edge g → g → Vertex g
  outEdges :: Vertex g → g → [Edge g]
```

In contrast to the *Array* and *MapKey* examples, in which the container type was associated to the element type, here the vertex and edge type are associated to the container type. This allows us to define several distinct instances of graphs which all have the same edge and vertex types, but differ in the representation and algorithms working on the data structure. Two possible instances which both model vertices as integers, and edges as pairs of source and target vertex:

```
-- adjacency matrix
newtype G1 = G1 [[Vertex G1]]
instance Graph G1 where
  newtype Vertex G1 = GV1 Int
  data Edge G1      = GE1 (Vertex G1) (Vertex G1)
-- maps vertices to neighbours
newtype G2 = G2 (FiniteMap (Vertex G2) (Vertex G2))
instance Graph G2 where
  newtype Vertex G2 = GV2 Int
  data Edge G2      = GE2 (Vertex G2) (Vertex G2)
```

In addition to the flexibility which we gain by defining the class in this way, this method also offers two more distinct advantages: first, as with traits, we reduce the number of parameters to the class; second, in contrast to class parameters, we refer to the associated types by their names, not just the position in the argument list, which further improves the readability of the program and reduces the potential for confusing the order of arguments.

2.3 Interface Abstraction

All previous examples used associated types for self-optimising libraries specialising data representations and algorithms in a type-directed manner. An entirely different application is that of defining *abstract interfaces*, not unlike abstract base classes in C++, interfaces in Java, or signatures in Standard ML.

A well-known example of such an interface, which has been discussed in the context of Haskell’s hierarchical libraries, is that of a monad based on a state transformer that supports mutable references. We can collect them in a parametrised family of types as follows:

```
class Monad m => RefM m where
  data Ref m v
  newRef  :: v -> m (Ref m v)
  readRef :: Ref m v -> m v
  writeRef :: Ref m v -> v -> m ()
instance RefM IO where
  data Ref IO = RefIO Data.IORef.IORef
  newRef      = Data.IORef.newIORef
...
instance RefM (ST s) where
  data Ref (ST s) = RefST (Data.STRef.STRef s)
  newRef          = Data.STRef.newSTRef
...
```

Note how here both the type parameter m of the associated type *Ref* as well the representation types *Ref m v* are of higher kind—that is, they are of kind $\star \rightarrow \star$. The complete signature of *newRef* is

```
newRef :: RefM m => v -> m (Ref m v)
```

A subtlety of the above code is that the type *Ref IO* is not compatible with *Data.IORef.IORef*, as Haskell data type constructors are subject to nominal type equality. This may not be desired in some interfaces, where we prefer *type synonyms* that preserve structural type equality. We introduce local type synonyms in the next subsection.

2.4 Associated Type Synonyms

Associated type *synonyms* can be used to define overloaded arithmetic operations in a manner that is much more flexible than the standard classes of Haskell 98. In particular, using an example of Duck et al. [9], we might like a multiplication operation that performs automatic coercions when its two arguments are of different numeric types. We omit the obvious method definitions:

```
class Mul a b where
  type Prod a b :: *
  (*)          :: a -> b -> Prod a b
instance Mul Int Int where
  type Prod Int Int = Int
instance Mul Int Float where
  type Prod Int Float = Float
```

In contrast to associated *data* types, the result of the $(*)$ is an existing type (*Int* or *Float* here) rather than the associated data type defined in the class. In contrast to the standard numeric classes of Haskell, the class *Mul* generalises nicely to richer types such as vectors.

```
type Vec e = [e]
instance Mul a b => Mul a (Vec b) where
  type Prod a (Vec b) = Vec (Prod a b)
```

This and similar examples that have been discussed in the context of multi-parameter type classes with functional dependencies [21] can be reformulated to use associated types. Associated type synonyms add useful new power, but also new complexity. We discuss them further in the Appendix, but otherwise restrict ourselves to associated *data* types for the rest of the paper.

3 Associated Data Types in Detail

In this section, we describe the proposed language extension in enough detail for a user of the language. Technical details of the type system are deferred until Section 5.

We propose that a type class may define, in addition to a set of methods, a set of *associated* data types. In the class declaration, the data types are declared without any definitions; the definitions will be given by the instance declarations. The associated data type must be parameterised 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. Rationale for this restriction is given in Section 5.4.

A new type constructor is introduced for each associated data type in the same way as a normal top-level data type. The kind of the type constructor is inferred in the obvious way; we also allow explicit kind signatures on the type parameters:

```
class C a where
  data T a (b :: * -> *)
```

Instance declarations must give a single definition for each associated data type of the class; such a definition must repeat the class parameters of the instance, and any additional parameters of the data type must be left as type variables. For example, the following is a legal instance of the *C* class above:

```
instance C a => C [a] where
  data T [a] b = D [T a] b
```

An instance declaration with associated data types introduces new data constructors with top-level scope. In the above example, the data constructor *D* is introduced with the following type:

$$D :: C a \Rightarrow [T a] \rightarrow b \rightarrow T [a]$$

The instance of an associated data type may use a **newtype** declaration instead of a **data** declaration if there is only a single constructor with a single field. This enables the compiler to represent the datatype without the intervening constructor at runtime.

3.1 Default definitions

In Haskell, a class method can be given a default definition in the declaration of the class, and any instance that omits a specific definition for that method will inherit the default. Unfortunately, we cannot provide a similar facility for associated data types. To see why, consider the *ArrayElem* example from the introduction, and let's add a hypothetical default definition for the *Array* associated type:

```
class ArrayElem e where
  data Array e = DefaultArray (BoxedArray e)
  index :: Array e → Int → e
```

Now, what type should the *DefaultArray* constructor have? Presumably, it should be given the type

$$\text{DefaultArray} :: \text{ArrayElem } e \Rightarrow \text{BoxedArray } e \rightarrow \text{Array } e$$

But it cannot have this type, because this constructor is not valid for certain instances of *ArrayElem*, namely those which give their own specific definitions of the *Array* type. There is no correct type that we can give to a constructor of a default definition.

However, when the system is extended to include associated type synonyms (Section A), we shall see that default definitions make sense for an associated type synonym.

3.2 Types involving associated data types

The type constructor introduced by an associated data type declaration can be thought of as a type-indexed type. Its representation is dependent on the instantiation of its parameters, and we use Haskell's existing overloading machinery to resolve these types. There is a duality between methods of a class and associated data types: methods introduce overloaded, or type-indexed, variables, and associated data type declarations introduce type-indexed types.

This means that just as an expression that refers to overloaded identifiers requires instances to be available or a context to be supplied, the same is now true of types. Going back to the *Array* example from the introduction, consider the following signature:

$$f :: \text{Array Bool} \rightarrow \text{Bool}$$

Our system declares this to be a valid type signature only if there is an instance for *ArrayElem Bool*. Similarly,

$$f :: \text{Array } e \rightarrow e$$

is invalid, because the representation for *Array e* is unknown. To make the type valid, we have to supply a context:

$$f :: \text{ArrayElem } e \Rightarrow \text{Array } e \rightarrow e$$

There is one further restriction on the use of an associated type constructor: wherever the type constructor appears, it must be applied to at least as many type arguments as there are class parameters. This is not so surprising when stated in a different way: a type-indexed type must always be applied to all of its index parameters.

3.3 Associated types in data declarations

For consistency, the system should support using associated types everywhere, including in the definition of another data type. However, doing this has some interesting consequences. Consider again the *Array* example, and suppose we wish to define a new data type *T*:

```
data T e = C (Array e)
```

The type *T e* is not valid type for all *e*, so we must add a context:

```
data ArrayElem e ⇒ T e = C (Array e)
```

This context works in a similar way to contexts on data types in Haskell currently: it adds a context to the type of the constructor.

$$C :: \text{ArrayElem } e \Rightarrow \text{Array } e \rightarrow T e$$

Now, the type constructor *T* is no ordinary type constructor: it behaves in a similar way to an associated type, in that whenever *T e* appears in a type there must be an appropriate context or instances in order to deduce *ArrayElem e*. Furthermore, *T* must always be applied to all of its type-indexed arguments.

4 Translation example

Here we illustrate the translation from Haskell with associated types into System F, by walking through the translation for the *MapKey* example in Section 2. We will present the translated terms using Haskell syntax for convenience.

To recap, here is the class declaration for the *MapKey* class:

```
class MapKey k where
  data Map k v
  empty :: Map k v
  lookup :: k → Map k v → Maybe v
```

Its translation is a new data type, *CMapKey*, which is the type of dictionaries of the *MapKey* class:

```
data CMapKey k mk
  = CMapKey {
    empty :: forall v. mk v,
    lookup :: forall v. k → mk v → Maybe v
  }
```

The *CMapKey* type has a type parameter for each class type variable as usual, in this case the single type variable *k*. However, it now also has an extra type parameter *mk* representing the associated type *Map k*. This is because each instance of the class will give a different instantiation for the type *Map k*, so the dictionary must abstract over this type.

Note that *mk* has kind $\star \rightarrow \star$; the type variable *v* is still polymorphic by virtue of not being one of the class type variables. Indeed, the class methods *empty* and *lookup* are now explicitly polymorphic in this type variable.

Our first instance is the unit instance:

```
instance MapKey () where
  data Map () v = MapUnit (Maybe v)
  empty = MapUnit Nothing
  lookup () (MapUnit m) = m
```

Its translation is a new datatype for the associated type, and a dictionary value:

```
data MapUnit v = MapUnit (Maybe v)
```

```

dMapUnit :: CMapKey () MapUnit
dMapUnit = CMapKey {
  empty      = MapUnit Nothing,
  lookup () (MapUnit r) = r
}

```

Next we consider the instance for pairs:

```

instance (MapKey a, MapKey b) =>
  MapKey (a,b) where
  data Map (a,b) v      = MapPair (Map a (Map b v))
  empty                = MapPair empty
  lookup (a,b) (MapPair m) = lookup b (lookup a m)

```

Its translation is a new datatype, as before, and a dictionary function:

```

data MapPair ma mb v = MapPair (ma (mb v))

dMapPair :: forall a b. CMapKey a ma -> CMapKey b mb
          -> CMapKey (a,b) (MapPair ma mb)
dMapPair = \da.db.CMapKey {
  empty = MapPair (empty da),
  lookup (a,b) (MapPair m) =
    (lookup db) b ((lookup da) a m)
}

```

The new datatype *MapPair* takes two additional type arguments, *ma* and *mb*, representing the types *Map a* and *Map b* respectively. Because this instance has a context, the translation is a dictionary function, taking dictionaries for *MapKey a* and *MapKey b* as arguments before delivering a dictionary value.

The translation for the *Either* instance doesn't illustrate anything new, so it is omitted. Instead, we give a translation for an example function making use of the overloaded *lookup* function:

```

f :: MapKey (a,Int) => Map (a,Int) v -> a -> v
f m x = lookup (x,42) m

```

translation:

```

f :: forall a v mk. CMapKey (a,Int) mk
  -> mk v -> a -> v
f = \da.lm.lxi. lookup da (ix,42) m

```

In this example the programmer has specified the same type as the inferred type in the signature. The translation becomes slightly trickier when the type signature specifies a different context, albeit one that is entailed by the inferred context *MapKey(a,Int)*:

```

f :: MapKey a => Map (a,Int) v -> a -> v
f m x = lookup (x,42) m

```

translation:

```

f :: forall a v mk. CMapKey a mk
  -> MapPair mk MapInt v -> a -> v
f = \da.lm.lxi.
  lookup (dMapPair da dMapInt) (ix,42) m

```

Note that in translating the type *Map (a,Int) v*, the instances for *MapKey (a,b)* and *MapKey Int* must be consulted, just as they must be consulted to check that *MapKey (a,Int)* entails *MapKey a* and to construct the dictionary for *MapKey(a,Int)* in the value translation.

5 Type System and Evidence Translation

In this section, we formalise a type system for a lambda calculus including type classes with associated data types. We then extend the typing rules to include a translation of source programs into

Symbol Classes

$\alpha, \beta, \gamma \rightarrow$ ⟨type variable⟩
 $T \rightarrow$ ⟨type constructor⟩
 $D \rightarrow$ ⟨type class⟩
 $S \rightarrow$ ⟨associated type⟩
 $C \rightarrow$ ⟨data constructor⟩
 $x, f, d \rightarrow$ ⟨term variable⟩

Source declarations

$pgm \rightarrow \overline{data}; \overline{cls}; \overline{inst}; \overline{val}$ (whole program)
 $data \rightarrow \mathbf{data} T \overline{\alpha} = C \overline{\tau}$ (data type declaration)
 $cls \rightarrow \mathbf{class} D \alpha \mathbf{where}$ (class declaration)
 $dsig; vsig$
 $inst \rightarrow \mathbf{instance} \theta \mathbf{where}$ (instance declaration)
 $adata; val$
 $val \rightarrow x = e$ (value binding)
 $dsig \rightarrow \mathbf{data} S \alpha \overline{\beta}$ (associated type signature)
 $vsig \rightarrow x :: \sigma$ (class method signature)
 $adata \rightarrow \mathbf{data} S \tau \overline{\beta} = C \overline{\xi}$ (associated data type)

Source terms

$e \rightarrow a \mid e_1 e_2 \mid \lambda x. e \mid \mathbf{let} x = e_1 \mathbf{in} e_2 \mid e :: \sigma$ (term)
 $v \rightarrow x \mid C$ (identifier)

Source types

$\tau, \xi \rightarrow T \mid \alpha \mid \tau_1 \tau_2 \mid \eta$ (monotypes)
 $\rho \rightarrow \tau \mid \pi \Rightarrow \rho$ (qualified type)
 $\sigma \rightarrow \rho \mid \forall \alpha. \sigma$ (type scheme)
 $\eta \rightarrow S \tau$ (associated-type application)

Constraints

$\pi \rightarrow D \tau$ (constraint)
 $\phi \rightarrow \pi \mid \pi \Rightarrow \phi$ (qualified constraint)
 $\theta \rightarrow \phi \mid \forall \alpha. \theta$ (constraint scheme)

Environments

$\Gamma \rightarrow \overline{v} : \overline{\sigma}$ (type environment)
 $\Theta \rightarrow \overline{\theta}$ (instance environment)

Figure 1: Syntax of expressions and types

the predicative fragment of System F. The type system is based on Jones' *Overloaded ML (OML)* [19, 20]. In fact, associated data types do not change the typing rules in any fundamental way; however, they require a substantial extension to the dictionary translation of type classes.

5.1 Syntax

The syntax of the source language is given in Figure 1. We denote type variables by α, β , and γ ; term variables by x, f , and d ; type constructors by T ; data constructors by C ; type classes by D ; and associated types by S . A program is a sequence of data type declarations, class declarations, instance declarations, and value bindings, where expressions are simple lambda terms including non-recursive local bindings.

We use overbar notation extensively. The notation $\overline{\alpha}^n$ means the sequence $\alpha_1 \dots \alpha_n$; the “ n ” may be omitted when it is unimportant. Moreover, we use comma to mean sequence extension as follows: $\overline{\tau}^n, a_{n+1} \triangleq \overline{\tau}^{n+1}$. Although we give the syntax of qualified and quantified types in a curried way, we also sometimes use equiv-

alent overbar notation, thus:

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

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 $(\rightarrow) \tau_1 \tau_2$.

There are two unconventional feature of the source language. First, **class** declarations may contain **data** type signatures in addition to method signatures and correspondingly **instance** declarations may contain **data** type declarations in addition to method implementations. These data types are the *associated types* of the class.

Second, the syntax of types τ includes η , the saturated application of an associated type; that is, an application in which the number of arguments matches the arity of the parent class. (There can be further type arguments by way of the $\tau_1 \tau_2$ production.) The arity of an associated type is the arity the class it is defined in; hence the notation **data** $S \alpha \beta$. Any extra arguments do not count towards the arity; i.e., saturated applications of associated types may be of higher kind.

We use θ to stand for *constraint schemes*—i.e., closed predicate implications—which we collect in instance environments Θ throughout the typing rules. In addition, the typing rules maintain a *type environment* Γ .

We make the following simplifying assumptions to reduce the notational burden:

- Each class has exactly one type parameter. Hence classes and associated types have arity 1.
- Each class has exactly one method, and one associated type.
- Each data type has a single constructor. Furthermore, rather than treat **case** expressions we assume that each constructor C comes with a projection function prj_i^C that selects the i th argument of the constructor C .
- We do not treat superclasses, nor default declarations in classes.
- Throughout we assume that all types are well kinded, without stating these assumptions formally.

Loosening the first three simplifications is largely a matter of adding (a great many) overbars to the typing rules, although introducing superclasses is slightly less trivial, as Section 5.5 discusses.

5.2 Type Checking

A key feature of our system is that the typing rules for expressions are very close to those of Haskell. We present them in Figure 2. The judgement $\Theta \mid \Gamma \vdash e : \sigma$ means that in type environment Γ and instance environment Θ the expression e has type σ . All the rules are absolutely standard for a Damas-Milner type system except $(\Rightarrow I)$ and $(\Rightarrow E)$. The former allows us to abstract over a constraint, while the latter allows us to discharge a constraint provided it is entailed by the environment. The latter judgement, $\Theta \Vdash \pi$ is also given in Figure 2, and is also entirely standard [20].

The only unusual feature is the judgement $\Theta \vdash \sigma$, which is used in rules $(\rightarrow I)$ and $(\forall E)$ to check that the type τ is well formed with respect to the available instances. It is this side condition that rejects, for example, the typing

$$\Theta \mid \Gamma \vdash \lambda x.x : \forall \alpha. \text{Map } \alpha \rightarrow \text{Int}$$

Target declarations	
$td \rightarrow$	$(x:\upsilon) = w \mid \text{data } T \overline{\alpha} = C \overline{\upsilon}$
Target terms	
$w \rightarrow$	$v \mid w_1 w_2 \mid \lambda(x:\upsilon).w \mid \Lambda \alpha.w \mid w \upsilon$ $\mid \text{let } x:\upsilon = w_1 \text{ in } w_2$
Target types	
$\upsilon \rightarrow$	$T \mid \alpha \mid \upsilon_1 \upsilon_2 \mid \forall \alpha.\upsilon$
Environments	
$\Delta \rightarrow$	$\overline{d}:\overline{\theta}$ (dictionary environment)
$\Omega \rightarrow$	$\overline{\omega}$ (associated-type environment)
$\omega \rightarrow$	$\forall \overline{\alpha}.\eta \rightsquigarrow T \overline{\tau}$ $\mid \eta \rightsquigarrow \alpha$

Figure 3: Syntax for target terms and types

This typing is invalid because the associated type $\text{Map } \alpha$ is meaningless without a corresponding $\text{MapKey } \alpha$ constraint.

The rules for class and instance declarations are not quite so standard, because of the possibility of one or more type declarations in the class. We omit the details because they form part of the more elaborate rules we give next. However, the reason that the type well-formedness judgement $\Theta \vdash \sigma$ is specified to work for type schemes (rather than just monotypes) is because it is needed to check the validity of the types of class methods.

5.3 Evidence Translation

A second crucial feature of our system is that, like Haskell 98, it can be translated into System F (augmented with data types) without adding any associated-type extensions to the target language. We gave an example of this translation in Section 4. In this section we formalise the translation.

5.3.1 Evidence translation for terms

The main judgement

$$\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \sigma$$

means that in environment $\Omega \mid \Delta \mid \Gamma$ the source term e has type σ , and translates to the target term w (Figure 5). The rules for this judgement are given in Figure 5; for the most part they are a well-known elaboration of the rules in Figure 2 [11].

The *target term* w is explicitly-typed in the style of System F, and its syntax is given in Figure 3. The main typing judgement derives a *source type* σ , whereas the target term is decorated with *target types*. The programmer only sees source types σ , which include qualified types and applications of associated types. In contrast, a target type υ mentions only data types: no qualified types, and no associated types appear.

The instance environment Θ from the plain type-checking rules has split into two components, Ω and Δ (see Figure 3). The *dictionary environment* Δ associates a dictionary (or dictionary-producing function) d with each constraint scheme θ , but it otherwise contains the same information as the old Θ . The *well-formedness* judgement $\Theta \vdash \sigma$ from Figure 2, used in rules $(\rightarrow I)$ and $(\forall E)$, becomes a *type-translation* judgement $\Omega \vdash \sigma \rightsquigarrow \upsilon$, that translates source types to target types. This type translation is driven by the associated-type environment Ω . We discuss type translation further in Section 5.3.2.

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

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

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

$$\frac{\Theta \Vdash D \tau \quad S \text{ is an associated type of } D}{\Theta \vdash S \tau} \quad \frac{\Theta \vdash \sigma \quad \alpha \notin \text{Fv}(\Theta)}{\Theta \vdash \forall \alpha. \sigma} \quad \frac{\Theta, \pi \vdash \rho}{\Theta \vdash \pi \Rightarrow \rho} \quad \frac{\Theta \vdash \tau_1 \quad \Theta \vdash \tau_2}{\Theta \vdash \tau_1 \tau_2} \quad \frac{}{\Theta \vdash \alpha} \quad \frac{}{\Theta \vdash T}$$

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

$$\frac{(v : \sigma) \in \Gamma}{\Theta \mid \Gamma \vdash v : \sigma} (var) \quad \frac{\Theta \mid \Gamma \vdash e_1 : \sigma_1 \quad \Theta \mid \Gamma[x : \sigma_1] \vdash e_2 : \sigma_2}{\Theta \mid \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \sigma_2} (let)$$

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

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

$$\frac{\Theta \mid \Gamma \vdash e : \sigma \quad \alpha \notin \text{Fv}(\Theta) \cup \text{Fv}(\Gamma)}{\Theta \mid \Gamma \vdash e : \forall \alpha. \sigma} (\forall I) \quad \frac{\Theta \mid \Gamma \vdash e : \forall \alpha. \sigma \quad \Theta \vdash \tau}{\Theta \mid \Gamma \vdash e : [\tau/\alpha]\sigma} (\forall E) \quad \frac{\Theta \mid \Gamma \vdash e : \sigma}{\Theta \mid \Gamma \vdash (e :: \sigma) : \sigma} (sig)$$

Figure 2: Standard type checking rules for expressions

$$\boxed{\Omega \vdash \sigma \rightsquigarrow v}$$

$$\frac{(\forall \bar{\alpha}. \eta \rightsquigarrow T \bar{\tau}) \in \Omega \quad \Omega \vdash \overline{[\bar{\tau}'/\bar{\alpha}]\tau} \rightsquigarrow v}{\Omega \vdash \overline{[\bar{\tau}'/\bar{\alpha}]\eta} \rightsquigarrow T \bar{v}} (tr\Omega 1) \quad \frac{(\eta \rightsquigarrow \alpha) \in \Omega}{\Omega \vdash \eta \rightsquigarrow \alpha} (tr\Omega 2) \quad \frac{}{\Omega \vdash \alpha \rightsquigarrow \alpha} \quad \frac{}{\Omega \vdash T \rightsquigarrow T}$$

$$\frac{\Omega \vdash \tau_1 \rightsquigarrow v_1 \quad \Omega \vdash \tau_2 \rightsquigarrow v_2}{\Omega \vdash \tau_1 \tau_2 \rightsquigarrow v_1 v_2} \quad \frac{\Omega \vdash \sigma \rightsquigarrow v \quad \alpha \notin \text{Fv}(\Omega)}{\Omega \vdash \forall \alpha. \sigma \rightsquigarrow \forall \alpha. v} \quad \frac{\Omega \vdash \pi \rightsquigarrow \alpha, \eta, v_d \quad \Omega[\eta \rightsquigarrow \alpha] \vdash \rho \rightsquigarrow v}{\Omega \vdash \pi \Rightarrow \rho \rightsquigarrow \forall \alpha. v_d \rightarrow v} (tr\pi)$$

$$\boxed{\Omega \vdash \pi \rightsquigarrow v} \quad \boxed{\Omega \vdash \pi \rightsquigarrow \alpha, \eta, v}$$

$$\frac{S = \text{the associated type of } D \quad \Omega \vdash S \tau \rightsquigarrow v}{\Omega \vdash D \tau \rightsquigarrow v} (\pi E) \quad \frac{S = \text{the associated type of } D \quad \alpha \text{ fresh} \quad \Omega \vdash \tau \rightsquigarrow v}{\Omega \vdash D \tau \rightsquigarrow \alpha, S \tau, (D v \alpha)} (\pi I)$$

$$\boxed{\Omega \mid \Delta \Vdash \theta \rightsquigarrow w}$$

$$\frac{(d : \theta) \in \Delta}{\Omega \mid \Delta \Vdash \theta \rightsquigarrow d} (mono) \quad \frac{\Omega \mid \Delta \Vdash \forall \alpha. \theta \rightsquigarrow w}{\Omega \mid \Delta \Vdash [\tau/\alpha]\theta \rightsquigarrow w \tau} (spec) \quad \frac{\Omega \mid \Delta \Vdash \pi \Rightarrow \phi \rightsquigarrow w \quad \Omega \mid \Delta \Vdash \pi \rightsquigarrow w' \quad \Omega \vdash \pi \rightsquigarrow v}{\Omega \mid \Delta \Vdash \phi \rightsquigarrow w v w'} (mp)$$

Figure 4: Definition of the entailment relation \Vdash

$$\boxed{
\begin{array}{c}
\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w' : \sigma \\
\\
\frac{(v : \sigma) \in \Gamma}{\Omega \mid \Delta \mid \Gamma \vdash v \rightsquigarrow v : \sigma} (var) \quad \frac{\Omega \mid \Delta \mid \Gamma \vdash e_1 \rightsquigarrow w_1 : \sigma_1 \quad \Omega \mid \Delta \mid \Gamma[x : \sigma_1] \vdash e_2 \rightsquigarrow w_2 : \sigma_2 \quad \Omega \vdash \sigma_1 \rightsquigarrow v}{\Omega \mid \Delta \mid \Gamma \vdash (\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2) \rightsquigarrow (\mathbf{let} \ x = v = w_1 \ \mathbf{in} \ w_2) : \sigma_2} (let) \\
\\
\frac{\Omega \mid \Delta \mid \Gamma[x : \tau_1] \vdash e \rightsquigarrow w : \tau_2 \quad \Omega \vdash \tau_1 \rightsquigarrow v_1}{\Omega \mid \Delta \mid \Gamma \vdash (\lambda x. e) \rightsquigarrow (\lambda x : v_1. w) : \tau_1 \rightarrow \tau_2} (\rightarrow I) \quad \frac{\Omega \mid \Delta \mid \Gamma \vdash e_1 \rightsquigarrow w_1 : \tau_2 \rightarrow \tau_1 \quad \Omega \mid \Delta \mid \Gamma \vdash e_2 \rightsquigarrow w_2 : \tau_2}{\Omega \mid \Delta \mid \Gamma \vdash (e_1 \ e_2) \rightsquigarrow (w_1 \ w_2) : \tau_1} (\rightarrow E) \\
\\
\frac{\Omega \vdash \pi \rightsquigarrow \alpha, \eta, v \quad \Omega[\eta \rightsquigarrow \alpha] \mid \Delta[d : \pi] \mid \Gamma \vdash e \rightsquigarrow w : \rho}{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow (\Lambda \alpha. \lambda(d : v). w) : \pi \Rightarrow \rho} (\Rightarrow I) \quad \frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \pi \Rightarrow \rho \quad \Omega \mid \Delta \vdash \pi \rightsquigarrow w' \quad \Omega \vdash \pi \rightsquigarrow v}{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w \ v \ w' : \rho} (\Rightarrow E) \\
\\
\frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \sigma \quad \alpha \notin \text{Fv}(\Delta) \cup \text{Fv}(\Gamma)}{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow (\Lambda \alpha. w) : \forall \alpha. \sigma} (\forall I) \quad \frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \forall \alpha. \sigma \quad \Omega \vdash \tau \rightsquigarrow v}{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w \ v : [\tau/\alpha]\sigma} (\forall E) \quad \frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \sigma}{\Omega \mid \Delta \mid \Gamma \vdash (e :: \sigma) \rightsquigarrow w : \sigma} (sig)
\end{array}
}$$

Figure 5: Typing rules with translation

Returning to the rules for terms, the interesting cases are the rules $(\Rightarrow I)$ and $(\Rightarrow E)$, which must deal with the associated types. In rule $(\Rightarrow I)$ we must abstract over the type variables that stand for D 's associated types. As well as augmenting the dictionary environment Δ to reflect the constraint that is now satisfied by the environment, and its witness d , we augment the type-translation environment Ω to explain how D 's associated types may be rewritten.

Dually, rule $(\Rightarrow E)$ applies the target term w to the witness *type* v as well as the witness *term* w' . The witness types are derived by the judgement $\Omega \vdash \pi \rightsquigarrow v$, while the witness terms are derived by $\Omega \mid \Delta \vdash \pi \rightsquigarrow w$, both given in Figure 4.

5.3.2 Translating types

The translation of source types to target types is formalised by the judgement $\Omega \vdash \sigma \rightsquigarrow v$ of Figure 4, which eliminates applications of associated types by consulting the *associated-type environment* Ω . This judgement relates to the well-formedness judgement $\Theta \vdash \sigma$ of Figure 2 in just the same way that the typing judgement $\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \sigma$ relates to $\Theta \mid \Gamma \vdash e : \sigma$.

To motivate the rules, here are some type translations copied from Section 4:

$$\begin{array}{l}
\Omega \vdash \text{Map Int} \rightsquigarrow \text{MapInt} \\
\Omega \vdash \forall \alpha \gamma. (\text{MapKey } \alpha) \Rightarrow \text{Map } (\alpha, ()) \gamma \rightarrow \alpha \rightarrow \gamma \\
\rightsquigarrow \forall \alpha \beta. \text{MapKey } \alpha \beta \rightarrow \text{MapPair } \beta \ \text{MapInt } \gamma \rightarrow \alpha \rightarrow \gamma
\end{array}$$

The first example is straightforward, because it arises directly from the **instance** declaration for `MapKey ()`. There is more going on in the second example. The class constraint is translated to an ordinary function, with argument type `MapKey α β` . The (target) data type `MapKey` is the type of dictionaries for class `MapKey`, and is generated by translating the class declaration. The crucial point is that this data type takes an extra type parameter β for each associated type of the class, here just one. Correspondingly, we must quantify over the new type β as well. The type `Map $(\alpha, ())$` is first translated to `MapPair (Map α) (Map ())`, by applying the instance declaration for pairs. Then the `(Map ())` is translated to `MapInt` as before, while the `Map α` is precisely the associated type for the class `MapKey α` , and so is translated to β .

The associated-type environment therefore contains two kinds of assumptions, ω (Figure 3). First, from an instance declaration we get an assumption of the form $\forall \bar{\alpha}. S \xi \rightsquigarrow T \bar{\tau}$, where S is an associated data type and T is the corresponding target data type. For

example, consider the instances of class `MapKey` in Section 4. The instances for `Int` and pairs augment Ω with the assumptions:

$$\begin{array}{l}
\text{Map Int} \rightsquigarrow \text{MapMapInt} \\
\forall \alpha_1 \alpha_2. \text{Map } (\alpha_1, \alpha_2) \rightsquigarrow \text{MapPair } (\text{Map } \alpha_1) \ (\text{Map } \alpha_2)
\end{array}$$

We will see the details of how Ω is extended in this way when we discuss the rule for instance declarations in the next section. Second, when in the midst of translating a type, we extend Ω with local assumptions of form $S \xi \rightsquigarrow \beta$ — which is denoted as $\eta \rightsquigarrow \beta$ in rule $(tr\pi)$ of Figure 4, and also $(\Rightarrow I)$ of Figure 5. For example, when moving inside the “`MapKey $\alpha \Rightarrow$` ” qualifier in the example above, we add the assumption `Map $\alpha \rightsquigarrow \beta$` to Ω .

Whenever we need to extend Ω with local assumptions of form $S \xi \rightsquigarrow \alpha$, we use a judgment of the form $\Omega \vdash \pi \rightsquigarrow \alpha, \eta, v$ (from Figure 4). Such a judgment abstracts over the associated type of π by introducing a new type variable α that represents the associated type. It also provides the application of the associated type at the class instance π , as η , and the corresponding dictionary type, as v .

5.3.3 Data type and value declarations

The rules for type-directed translation of declarations are given in Figure 6. They are somewhat intimidating, but that is largely because of the notational overheads, and much of the complexity is also present in vanilla Haskell 98. There is real work to be done, however, and that is the whole point. The programmer sees Haskell's type system more or less unchanged, but the implementation has to do a good deal of paddling under the water to implement the associated types.

The translation of data type declarations is easy: all we need do is translate the constructor argument types, using our auxiliary type-translation judgement $\Omega \vdash \tau \rightsquigarrow v$. Value declarations are also straightforward, because all of the work is done in Figures 4 and 5.

5.3.4 Class and instance declarations

The interesting cases are class and instance declarations. It may help to refer back to the example of Section 4 when reading these rules. As noted there, a class declaration for class D is translated to a data type declaration, also named D , whose data constructor is called C_D . This data type will be used to represent the dictionary for class D , so the constructor has the class methods signature(s) σ as its argument types, suitably translated of course. The translation uses an associated-type environment Ω' that maps each associated type to a fresh type variable β . The data type must be

$$\boxed{\Omega \vdash data \rightsquigarrow td : \Gamma}$$

$$\frac{\Omega \vdash \tau \rightsquigarrow v^c}{\Omega \vdash data T \bar{\alpha} = C \bar{\tau} \rightsquigarrow data T \bar{\alpha} = C \bar{v}^c : [C : \forall \bar{\alpha}. \bar{\tau} \rightarrow T \bar{\alpha}, prj^C : \forall \bar{\alpha}. T \bar{\alpha} \rightarrow \tau]} (data)$$

$$\boxed{\Omega \vdash cls \rightsquigarrow \bar{td} : \Gamma}$$

$$\frac{\beta \text{ fresh} \quad \Omega' = \Omega[S \alpha \rightsquigarrow \beta] \quad \Omega' \vdash \sigma \rightsquigarrow v}{\Omega \vdash \begin{array}{l} \text{class } D \alpha \text{ where} \\ \text{data } S \alpha \bar{\gamma} \\ f :: \sigma \end{array} \rightsquigarrow \begin{array}{l} \text{data } D \alpha \beta = C_D v \\ f : \forall \alpha \beta. D \alpha \beta \rightarrow v = prj^{C_D} : [f : \forall \alpha. D \alpha \Rightarrow \sigma] \end{array}} (cls)$$

$$\boxed{\Omega \mid \Delta \mid \Gamma \vdash inst \rightsquigarrow \bar{td} : \Omega, \Delta, \Gamma}$$

$$\frac{\Omega \vdash \pi \rightsquigarrow \beta, \eta, v_d^r \quad \Omega' = \Omega[\eta \rightsquigarrow \beta^r] \quad \Delta' = \Delta[\bar{d} : \pi^r] \quad \Omega' \vdash \bar{\tau}_c^c \rightsquigarrow v_c^c \quad (f : \forall \delta. D \delta \Rightarrow \sigma) \in \Gamma \quad \Omega' \mid \Delta' \mid \Gamma \vdash e \rightsquigarrow w : [\tau/\delta]\sigma \quad \bar{d}^r \text{ fresh} \quad d_f \text{ fresh} \quad T \text{ fresh}}{\Omega \mid \Delta \mid \Gamma \vdash \begin{array}{l} \text{instance } \forall \bar{\alpha}^s. \bar{\pi}^r \Rightarrow D \tau \text{ where} \\ \text{data } S \tau \bar{\gamma} = C \bar{\tau}_c^c \\ f = e \end{array} \rightsquigarrow \begin{array}{l} \text{data } T \bar{\alpha}^s \bar{\beta}^r \bar{\gamma} = C \bar{v}_c^c \\ d_f = \Lambda \bar{\alpha}^s \bar{\beta}^r. \lambda \bar{d}^r. v_d^r . C_D w \end{array} : \begin{array}{l} [\forall \bar{\alpha}^s. (S \tau \rightsquigarrow T \bar{\alpha}^s \bar{\eta}^r)], \\ [d_f : \forall \bar{\alpha}^s. \bar{\pi}^r \Rightarrow D \tau], \\ [C : \forall \bar{\alpha}^s \bar{\gamma}. \bar{\pi}^r \Rightarrow \bar{\tau}_c^c \rightarrow S \tau \bar{\gamma}] \end{array}} (inst)$$

$$\boxed{\Omega \mid \Delta \mid \Gamma \vdash val \rightsquigarrow td : \Gamma}$$

$$\frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \sigma \quad \Omega \vdash \sigma \rightsquigarrow v}{\Omega \mid \Delta \mid \Gamma \vdash (x = e) \rightsquigarrow (x : v = w) : [x : \sigma]} (val)$$

$$\boxed{\vdash pgm \rightsquigarrow \bar{td}}$$

$$\frac{\overline{\Omega \vdash data \rightsquigarrow d_d : \Gamma_d} \quad \overline{\Omega \vdash cls \rightsquigarrow \bar{d}_c : \Gamma_c} \quad \overline{\Omega \mid \Delta \mid \Gamma \vdash inst \rightsquigarrow \bar{d}_i : \Omega, \Delta, \Gamma_i} \quad \overline{\Omega \mid \Delta \mid \Gamma \vdash val \rightsquigarrow \bar{d}_v : \Gamma_v} \quad \Gamma = \Gamma_d, \Gamma_c, \Gamma_i, \Gamma_v}{\vdash data; cls; inst; val \rightsquigarrow \bar{d}_d; \bar{d}_c; \bar{d}_i; \bar{d}_v}$$

Figure 6: Declaration typing rules with translation

parameterised over these fresh β , because they will presumably be free in the translated method types v . Finally, we must generate a binding for each method selector function for each class method f ; in the rule, they are implemented by the corresponding projection functions prj^{C_D} . In addition to the target declarations defining the data type for the dictionary and the method selector functions, the Rule (*cls*) produces an environment Γ giving the source types of the class methods.

Instance declarations are more awkward. For each associated type S of the class, we must generate a fresh data type declaration T that implements the associated type at the instance type(s). This data type must be parameterised over (a) the quantified type variables of the instance declaration itself, $\bar{\alpha}$, (b) a type variable for each associated type of each constraint in the instance declaration, $\bar{\beta}$, and (c) the type variables in which type is parametric in all instances, $\bar{\gamma}$. For example, here is an artificial example to demonstrate the possibilities:

```

class MapKey k where
  data Map k v

```

```

instance (MapKey a, MapKey b) => MapKey (a, b) where
  data Map (a, b) v = MapPair v a (Map b) b

```

The data type that arises from the instance declaration is this:

```

data Map' a b ma mb v = MapPair v a mb b

```

The arguments ma, mb were the $\bar{\beta}$ in (b) above. They may not all be needed, as we see in this example. As an optimisation, if any are unused in the (translated) right hand side of the declaration, they can be omitted from the type-parameter list. To produce the right-hand sides v_c from the τ_c of an instance's associated type declarations, we need to replace applications of other associated types by the newly introduced type parameters. This is achieved by the associated-type environment Ω' in the hypothesis.

In addition to promoting the associated data type S to become a fresh top-level data type declaration T , rule (*inst*) also returns in its conclusion (a) a tiny associated-type environment and dictionary environment that embody the information about the instance declaration for use in the rest of the program, and (b) a tiny type

environment that embodies the types of the new data constructor(s), C .

That concludes the hard part of instance declarations. The generation of the dictionary function, d_f , and the extension of the dictionary environment Δ , is exactly as in vanilla Haskell.

5.3.5 Tying the knot

The final judgement in Figure 6 glues together the judgements for types, classes, instances, and value declarations. This rule is highly recursive: the associated-type environment Ω that is produced by type checking instance declarations, is consumed by that same judgement and the other three judgements too. Similarly, all four judgements produce a fragment of the environment Γ , which is consumed by the judgements for instance and value declarations. There is a good reason for this recursion. For example, consider the data type G_1 from Section 2.2. Its constructor mentions the type *Vertex* G_1 , and the translation for that type comes from the instance declaration!

In practice, the implementation must unravel the recursion somewhat, and our new extension makes this somewhat harder than before. For example, in Haskell 98 one can type-check the instance declaration *heads* (the part before the **where**), to generate the top level Π , then check the value declarations to generate Γ , and then take a second run at the instance declarations, this time checking the method bodies. But now the instance declarations for one class may be needed to type-check the class declaration for another class, if the associated types for the former appear in the method type signatures for the latter. None of this is rocket science, but it is an unwelcome complication.

5.4 Associated type parameters

In Section 3 we specified that the type parameters of the associated type should be identical to those of its parent class, plus some optional extra parameters $\bar{\gamma}$. Now we can see why. The class parameters must occur *first* so that we can insist that associated-type applications are saturated (wrt the class parameters). That in turn ensures that the type translation described by Ω can proceed without concern for partial applications, and without clutter arising from the extra $\bar{\gamma}$.

We could in principle permit an associated type to *permute* its parent class parameters (where there is more than one), at the cost of extra notational bureaucracy in the (*inst*) rule, but there seems to be no benefit in doing to. We could also in principle allow an associated type to mention only a *subset* of its parent class parameters; but then we would need to make extra tests to ensure that the instance declarations did not overlap taking into account only the selected class parameters, to ensure that the type translation described by Ω is confluent. (A similar test must be made when functional dependencies are employed.) Again, the benefit does not seem to justify the cost.

5.5 Superclasses

Our formalisation of the type system and evidence translation does not take superclasses into account; i.e., there is no context in the head of a class declaration. We made this simplification in the interest of the clarity of the formal rules. However, it needs to be mentioned that there is a subtlety with respect to the translation of associated types in classes that have one or more superclasses. In rule (*cls*) of Figure 6, we see that the generated dictionary data type D has a set of type arguments β , each of which corresponds to one associated type of the class. If the class D has superclasses which themselves contain associated types, each of these associated

types needs to appear as an argument to the dictionary D , too. In other words, similar to how the dictionaries of superclasses must be embedded in a class's own dictionary, the associated types of superclasses need to also be embedded.

6 Comparison to Functional Dependencies

Functional dependencies [21] are an experimental addition to multi-parameter type classes that introduce a functional relationship between different parameters of a type class, which is similar to that between class parameters and associated types. Functional dependencies rely on Jones' [19] notion of *improvement* during type checking, but do not impact the evidence translation for multi-parameter type classes. In fact, Hinze et al. [17, Section 3.4] briefly mention the possibility of using functional dependencies to implement type-indexed types, but reject it without giving more than a vague rationale. In the following, we give hard technical reasons that prevent us from defining type-indexed types, such as the *Array* type from Section 1, using functional dependencies.

We illustrate the problems by encoding *Array* in relational form as two-parameter type class, thus:

```
class ArrayRep e arr | e → arr where
  index :: arr → Int → e
```

The functional dependency $e \rightarrow arr$ restricts the binary relation *ArrayRep* to a function from element types e to representation types arr . The instance declarations populate the relation represented by *ArrayRep*. In other words, the associated type is provided as an extra argument to the class instead of being local. Consequently, the corresponding instance declarations are as before, but with the local type definition replaced by instantiation of the second parameter to the class *ArrayElem* (with methods omitted):

```
instance ArrayRep Int UIntArr
instance (ArrayRep a arr, ArrayRep b brr) ⇒
  ArrayRep (a, b) (arr, brr)
```

This class-based definition of polytypic arrays suffers from three serious shortcomings, which we shall discuss now.

Unsound type constraints. As Duck et al. [9] point out, the instance for pairs *ArrayRep* (a, b) (arr, brr) is unsound, as the type variables arr and brr do not occur in the first argument (a, b) to the type constraint. If such instances are accepted, type inference in the presence of functional dependencies becomes undecidable. More precisely, it diverges for certain terms that should be rejected as being type incorrect. Jones' [21] original proposal of functional dependencies does not allow such instances.

Clutter. In the comparative study of Garcia et al. [10], mentioned in Section 2.2, Haskell receives full marks in all categories except the treatment of associated types in type classes with functional dependencies. In essence, the requirement to make all associated types into extra parameters of type classes results in more complicated and less readable code. This is illustrated by the parameter arr in the type class *ArrayRep*. These extra parameters appear in all signatures involving associated types and can be quite large terms in more involved examples, such as the graph library discussed by Garcia et al.

Lack of abstraction. We would expect that we can define type-indexed arrays in a module of their own and hide the concrete array representation from the user of such a module. However, an encoding based on functional dependencies does not allow for this level of abstraction. To see why this is the case, consider the full type of the *index* function

$$\text{index} :: \text{ArrayRep } e \text{ arr} \Rightarrow \text{arr} \rightarrow \text{Int} \rightarrow e$$

Avoiding the use of any knowledge of how arrays of integers are represented, we would expect to be able to define

$$\begin{aligned} \text{indexInt} &:: \text{ArrayRep Int arr} \Rightarrow \text{arr} \rightarrow \text{Int} \rightarrow e \\ \text{indexInt} &= \text{index} \end{aligned}$$

However, such a definition is not admissible, as the type signature is not considered to be an instance of the type inferred for the function body in the presence of the functional dependency $e \rightarrow \text{arr}$ (cf. the class declaration of *ArrayRep*). In fact, we are forced to use the following definition instead:

$$\begin{aligned} \text{indexInt}' &:: \text{UArrInt} \rightarrow \text{Int} \rightarrow e \\ \text{indexInt}' &= \text{index} \end{aligned}$$

This clearly breaks the intended abstraction barrier!

The root of the problem lies deep. A consequence of the evidence translation for type classes is that we would expect there to be a System F term that coerces *indexInt* into *indexInt'*. However, no such coercion exists. It would require a non-parametric operation, which is not present in System F [12].

Variations on functional dependencies. Duck et al. [9] propose a more liberal form of functional dependencies in which recursive instances, such as that of *ArrayRep*, do not lead to non-termination. However, they also require a radically different form of type checker based on the HM(X) [29] framework with constraint handling rules. Stuckey & Sulzmann [35] introduce an implementation of multi-parameter type class with functional dependencies that does not depend on a dictionary translation. As a result, they can avoid some of the problems of the original form of functional dependencies.

7 Related Work

Type classes. There is a significant amount of previous work that studies the relationship between type classes and type-indexed functions [36, 18, 1, 22, 23], mostly with the purpose of expressing generic functions using standard type classes alone.

Chen et al. [3] proposed *parametric type classes*—i.e., type classes with type parameters—to represent container classes with overloaded constructors and selectors. They provide a type system and type inference algorithm, but do not present an evidence translation. Parametric type classes are not unlike a type class with a single associated type synonym.

Functional dependencies. We have discussed the original form of functional dependencies already in Section 6 including the extensions by Duck et al. [9] and Stuckey & Sulzmann [35].

Neubauer et al. [27] introduce a functional notation for type classes with a single functional dependency that is very much like that of parametric type classes [3]. However, their proposal is just syntactic sugar for functional dependencies, as they translate the new form of classes into multi-parameter classes with a functional dependency before passing them on to the type checker. The same authors are more ambitious in a second proposal [28], where they add a full-blown functional logic language to the type system, based on the HM(X) [29] framework. Our treatment of local type synonyms during type inference can be seen as a special case of their generic approach, where we carefully preserve the termination of type inference. Neubauer et al. do not address the issue of a suitable evidence translation, which means that they can infer types, but not compile their programs.

Generic Haskell. Hinze et al [15, 17] propose a translation of type-indexed data types based on a type specialisation procedure. Their efforts have culminated in *Generic Haskell*, a pre-processor that translates code including type-indexed types and functions into Haskell including type system extensions such as rank-n types. They pay special attention to type-indexed data types that are structurally defined, such as the *Map* type from Section 2.1, and automatically perform the mapping from standard Haskell data type definitions to a representation based on binary sums and products. In recent work, Löh et al. [24] elaborated the original design and introduced *Dependency-style Generic Haskell*.

Functors in Standard ML. The relationship between Standard ML's higher-order modules, in the form of signatures, structures and functors, and Haskell's type classes has been the subject of many discussions. So far, type classes lacked local type declarations, which signatures always had. Essentially, an ML signature encapsulates a set of parametrised type and value declarations, which is what a type class with associated types does, too. Moreover, a structure or functor is a concrete instance of a signature, not unlike a type class instance. However, despite this conceptual similarity, the two concepts differ significantly. Structures are a term-level entity and hence a notion of a *phase distinction* [13] is required to separate static from dynamic semantics. In contrast, type classes are a purely static concept. In part, due to the involvement of the term level, ML's higher-order modules give rise to a very rich design space [8] and it is far from clear how the different concepts relate to type classes. Nevertheless, the introduction of local types to type classes shows that the previous lack of encapsulation of types is not a fundamental property of type classes.

Intensional type analysis. *Intensional type analysis* [14] realises type-indexed types by a type-level *Typerec* construct and has originally been proposed to facilitate the type-preserving optimisation of polymorphism. Subsequent work [31, 6, 5, 37] has elaborated on the seminal work by Harper and Morrisett who already outlined the relationship to type classes. A conceptual difference between intensional type analysis and type classes is that the former is based on an explicit runtime representation of type arguments, whereas our evidence translation targets System F, which has a type-erasure semantics. Crary et al. [6] proposed an alternative view on intensional type analysis based on type erasure. In any case, we need to pass method dictionaries at runtime, which can be regarded as an implicit type representation.

Constrained data types. Xi et al. [38] introduce type-indexed data types by annotating each constructor of a data type declarations with a type pattern, which they call a *guard*, present a type system, and establish its soundness. In their internal language type-indexing is explicit, which is in contrast to our approach, where all type-indexing is removed during the evidence translation (a phase that they call elaboration). Cheney & Hinze's [4] present a slightly generalised version of guarded data types by permitting equational type constraints at the various alternatives in a data type declaration. Both of these approaches differ from our class-based approach in that our type-indexed data types are open—a new class instance can always be added—whereas theirs is closed, as data type declarations cannot be extended.

Object-oriented languages. As we discussed in Section 2.2, associated types have a long standing tradition in C++ and are often collected in *traits classes* [26]. Garcia et al. [10] compared the support for generic programming in C++, Standard ML, Haskell, Eiffel, Generic Java, and Generic C#. There exists a plethora of work on generic programming in object-oriented programming languages, but is beyond the scope of this paper to review all of it.

8 Conclusions

We proposed to include type declarations along side value declarations in Haskell type classes. Such associated types of a type class are especially useful for implementing self-optimising libraries, but also serve to implement abstract interfaces and other concepts for which functional dependencies have been used in the past. For the case of associated data types, we demonstrated that dictionary-based evidence translation, which is standard for implementing type classes can be elegantly extended to handle associated types and, in particular, that the target language is not affected by the extension of the source language.

In future work we will elaborate on type checking in the presence of associated type synonyms and we plan to investigate the feasibility of generic default methods for classes involving associated types.

Acknowledgements. We thank Amr Sabry for discussions on a previous version of this approach and for pointing us to the work of Garcia et al. We thank Roman Lechtchinsky for his detailed feedback on our work and for sharing his insights into generic programming in C++. We thank Martin Sulzmann for a number of interesting discussions about type classes and functional dependencies. Last but not least, we thank Dave Abrahams, Brian McNamara, and Jeremy Siek for an interesting email exchange comparing Haskell type classes with C++ classes.

The first two authors have been partly funded by the Australian Research Council under grant number DP0211203. National ICT Australia is funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

9 References

- [1] Artem Alimarine and Rinus Plasmeijer. A generic programming extension for clean. In *International Workshop on the Implementation of Functional Languages*, number 2312 in Lecture Notes in Computer Science, pages 168–185. Springer-Verlag, 2001.
- [2] Manuel M. T. Chakravarty and Gabriele Keller. An approach to fast arrays in Haskell. In Johan Jeuring and Simon Peyton Jones, editors, *Lecture notes for The Summer School and Workshop on Advanced Functional Programming 2002*, number 2638 in Lecture Notes in Computer Science, 2003.
- [3] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *ACM Conference on Lisp and Functional Programming*. ACM Press, 1992.
- [4] James Cheney and Ralf Hinze. First-class phantom types. CUCIS TR2003-1901, Cornell University, 2003.
- [5] Karl Cray and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, 1999.
- [6] Karl Cray, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *ACM SIGPLAN International Conference on Functional Programming*, pages 301–312. ACM Press, 1998.
- [7] Nachum Dershowitz and Jean-Pierre Jouannaud. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, chapter Rewrite Systems, pages 243–320. Elsevier and MIT Press, 1990.
- [8] Derek Dreyer, Karl Cray, 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.
- [9] 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.
- [10] 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 Programming, Systems, Languages, and Applications*, pages 115–134. ACM Press, 2003.
- [11] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In *European Symposium On Programming*, number 788 in LNCS, pages 241–256. Springer-Verlag, 1994.
- [12] Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70(1):1–5, 1999.
- [13] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 341–354. ACM Press, 1989.
- [14] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 1995.
- [15] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.
- [16] Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. In Roland Backhouse and Jeremy Gibbons, editors, *Lecture notes for The Summer School and Workshop on Generic Programming 2002*, number 2793 in Lecture Notes in Computer Science, 2003.
- [17] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In Eerke Boiten and Bernhard Möller, editors, *Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002)*, number 2386 in Lecture Notes in Computer Science, pages 148–174. Springer-Verlag, 2002.
- [18] 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.
- [19] Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 1995.
- [20] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1), 1995.
- [21] 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.
- [22] Ralf Lämmel. The sketch of a polymorphic symphony. In *2nd International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *ENTCS*. Elsevier Science, 2002.

- [23] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical approach to generic programming. In *Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37, 2003.
- [24] Andres Löh, Dave Clarke, and Johan Jeuring. Dependency-style generic haskell. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 141–152. ACM Press, 2003.
- [25] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.
- [26] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
- [27] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A functional notation for functional dependencies. In *2001 ACM SIGPLAN Haskell Workshop*, 2001.
- [28] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 2002.
- [29] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999.
- [30] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.
- [31] Zhong Shao. Flexible representation analysis. In *Proceedings ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, 1997.
- [32] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library User Guide and Reference Manual*. Addison-Wesley, 2001.
- [33] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: Generic components for high-performance scientific computing. *Computing in Science and Engineering*, 1(6):70–78, 1999.
- [34] A. A. Stepanov and M. Lee. The standard template library. Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, 1994.
- [35] Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Transaction on Programming Languages and Systems*, 2004. To appear.
- [36] Stephanie Weirich. Type-safe cast: Functional pearl. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM Press, 2000.
- [37] Stephanie Weirich. Higher-order intensional type analysis. In *European Symposium on Programming (ESOP02)*, 2002.
- [38] Hongwei Xi, Chiyen 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.

A Associated Type Synonyms

As we have seen in the previous section, we can add local data type declarations to Haskell type classes without any significant changes to the typing rules, or indeed without any serious changes to type inference. Only when translating source types, expressions, and declarations into an explicitly typed intermediate language do we need to pay special attention to associated data types. In this translation, in the same way as class methods introduce dictionary parameters, class-local data types introduce representation type parameters.

The situation changes as soon as we allow local type synonyms in type classes. As we shall discuss in this section, local type synonyms lead to a significant change to the type system and, in particular, seriously impact the structure of type inference and especially type term unification. However, once the machinery for handling local data types during the dictionary translation is in place, local type synonyms do not have a major impact on the structure of that translation. In other words, both local data types and type synonyms require similar changes to the evidence translation, and additionally local type synonyms require further changes to the basic type system.

Space constraints prevent us from a thorough formal treatment of local type synonyms and their handling during type inference. However, we will outline the essential points in this section and discuss the details in a companion paper. The next subsection motivates the need for local type synonyms by discussing their special properties when compared to local data types. The following subsection sketches type inference in the presence of local type synonyms as well as the necessary changes to the evidence translation procedure.

A.1 Local Type Synonyms Versus Data Types

The utility of local type synonyms becomes apparent in the example of the *Mul* class of Section 2.4, part of whose definition we repeat here:

```
class Mul a b where
  type Prod a b :: *
  (*)          :: a -> b -> Prod a b

instance Mul Int Int where
  type Prod Int Int = Int
instance Mul Int Float where
  type Prod Int Float = Float
```

In this example, we want any application of the associated type *Prod a b* to be *equal* to the corresponding instance type; i.e., we want *Prod Int Int* to be equal to *Int*, so that we can use values of one for the other. If instead *Prod Int Int* were lifted and nominally distinct from *Int*, we could not use the conventional arithmetic operations on the result of the multiplication operation of the class *Mul*.

For similar reasons, we might want to use associated synonyms, instead of data types, in the example of the class *RefM* from Section 2.3. With data types, we cannot use the original operations on reference, such as *Data.IORef.newIORef*, interchangeable with the methods of the interface class *RefM*, as values of type *Ref m v* are wrapped into an additional data constructor, namely *RefIO* or *RefST*.

Finally, the idea of traits, discussed in Section 2.2 relies on encapsulating types in a class in a form that permits them to be passed as arguments to type constructors and classes in addition to enforcing the equality of two traits in different classes when those classes are used in conjunction.

In contrast to data types, associated synonyms can be of higher kind without explicitly naming the additional arguments; hence, we need to add a kind signature, as in the *Mul* example above. Moreover, associated type synonyms can have default declarations much like the default declarations of class methods. That is, if an instance does omit the definition of an associated synonym having a definition declarations, the default type is used for that instance. Such default declarations are, for example, useful for type-indexed container types, such as arrays, where we want to fall back on a vanilla boxed representation for all instances where we do not provide an optimised data representation. This is not possible for associated data types, as discussed in Section 3.1. The use of associated type synonyms also underlies some additional restrictions, which we cannot discuss here in detail—for example, we may not have recursive occurrences for the same reasons as recursion is not allowed for toplevel type synonyms.

A.2 Equational Type Equality

Just like toplevel type synonyms, associated type synonyms introduce a richer than the purely syntactic equality on type terms. However, in contrast to toplevel synonyms, we cannot expand associated synonyms out before type checking. As a result, the typing rules and implementation in the form of a type inference procedure need to explicitly deal with associated synonyms and the resulting richer notion of equality on type terms.

In other words, purely syntactic type equality is now extended to be modulo an equational theory defined by the type equations in class instances. In fact, the type equations of all class instances form a term rewriting system (TRS) on type terms. To preserve decidability of type checking and inference as well as a principal types property, we restrict the TRS to be confluent and terminating [7] by imposing the following three well-studied restrictions:

1. Instance heads must be left linear (i.e., no repeated occurrences of a type variable).
2. Instance heads must be constructor-based (i.e., no non-reducible synonym applications).
3. Right-hand sides of local synonyms must be decreasing (i.e., each application of a local synonyms must have strictly less constructors in its arguments than there are constructors on the left-hand side).

A confluent and terminating TRS guarantees unique normal forms for type terms, which in turn are sufficient for a tractable notion of equality. Given this equality, the typing rules from Figure 2 are easily extended to allow to replace type terms with equals.

A.3 Type Inference and Unification

Unfortunately, matters get more involved once we move from typing rules to a type inference algorithm. The basic, and well known, problem is that during unification, we may come across applications of associated type synonyms that we cannot reduce under the current set of substitutions. For example, referring again to the class *Mul*, we may encounter a situation where we need to unify *Prod Int a* with *Int*. We cannot decide whether this unification should succeed unless we fix a type for *a*—i.e., unification succeeds or not depending on whether *a* is instantiated to *Int* or *Float*, and hence, *Prod Int a* reduces to *Int* and *Float* respectively.

We call *Prod Int a ~ Int* a *unification constraint* and extend the standard type inference algorithm for a Damas-Milner with type classes to maintain a set of *pending unification constraints* in addition to the type substitution that arises from solved unification steps. In addition, we need to take care to avoid generalising over

variables that are mentioned in unification constraints in the rule for **let** bindings. We solve unification constraints with a set of rewrite rules based on the nondeterministic algorithm by Martelli & Montanari [25], where we add additional rules to rewrite type terms according to the rules of the TRS induced by the program’s instance declarations.

To add the treatment of associated synonyms to the evidence translation of Section 5, we need to add the mapping of source types to their normalised version to the type translation environment Ω .