Associated Types with Class

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

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

Abstract

Haskell's type classes allow ad-hoc overloading, or type-indexing, of *functions*. A natural generalisation is to allow type-indexing of *data types* as well. It turns out that this idea directly supports a powerful form of abstraction called *associated types*, which are available in C++ using traits classes. Associated types are useful in many applications, especially for self-optimising libraries that adapt their data representations and algorithms in a type-directed manner.

In this paper, we introduce and motivate associated types as a rather natural generalisation of Haskell's existing type classes. Formally, we present a type system that includes a type-directed translation into System F; the existence of this translation ensures that the addition of associated data types to an existing Haskell compiler only requires changes to the front end.

1 Introduction

In a recent OOPSLA paper, Garcia *et al.* compare the support for generic programming offered by Haskell, ML, C++, C#, and Java, using a graph-manipulation library as a motivating example [11]. They offer a table of qualitative conclusions, in which Haskell is rated favourably in all respects except one: access to so-called *associated types*. For example, we may want to represent arrays in a manner that depends on its element type. So, given an element type *e*, there is an associated type *Array e* of arrays of those elements. Inventing some syntax, we might define *Array* like this:

[*]The first two authors have been partly funded by the Australian Research Council under grant number DP0211203.

Submitted to POPL'05; July 19, 2004.

```
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 pair of arrays. (We assume that *UIntArr* and *BitVector* are built-in types representing unboxed integer arrays and bit vectors respectively.) These specialised representations are more efficient, in terms of both space and runtime of typical operations, than a type-invariant parametric representation. Data types whose concrete representation depends on one or more type parameters are called *type analysing* [15] or *type indexed* [18].

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 can now define instances of the *ArrayElem* class that give instantiations for the *Array* type, assuming *indexUIntArr* is a pre-defined function indexing unboxed integer arrays:

```
instance ArrayElem Int where
  data Array Int = IntArray UIntArr
  index (IntArray ar) i = indexUIntArr 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

```
index :: ArrayElem e ⇒ Array e → Int → 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 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 the new type system, and the evidence translation from source terms into System F (Sections 3.3 and 4). 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 Section 6. Our approach has a particularly close relationship to *functional dependencies* [22], which we review in Section 5.

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 [35], the Boost Graph Library [33], and the Matrix Template Library [34]. Work on generic programming in Haskell, also illustrates the need for type-dependent data representations [18, 17, 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 abstract interfaces in Subsection 2.3.

2.1 Self-optimising finite maps

A nice example of a data structure whose representation changes in dependence of a type parameter, which was first discussed by Hinze [16] and subsequently used as an example for type-indexed types by Hinze, Jeuring, and Löh [18] 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 *k* used to index the map. We express this idea by defining a type class *MapKey*, parameterised by the key type, with an associated type *Map* as one of its components:

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

(We give only two class operations, *empty* and *lookup*, but in reality there would be many.) In addition to the key type, finite maps are parametrised by a value type that forms the co-domain of the map. While the representation of generic finite maps depends on the type *k* of keys, it is parametrically polymorphic in the value type *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* [31], 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
  lookup k (MapInt d)  = Patricia.lookupDict k d
```

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 $\star \to \star$.

As described in detail by Hinze [16], 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—for a detailed motivation of this definition, please see Hinze's work:

```
instance MapKey () where
  data Map () v            = MapUnit (Maybe v)
  empty                    = MapUnit Nothing
  lookup Unit Nothing      = error "unknown key"
  lookup Unit (Just v)     = v
instance (MapKey a, MapKey b) ⇒
    MapKey (a, b) where
  data Map (a, b) v = MapPair (Map a (Map b v))
  empty             = Nothing
  lookup (a, b) fm  = lookup a (lookup b fm)
instance (MapKey a, MapKey b) ⇒
   MapKey (Either a b) where
  data Map (Either a b) v      =
   MapEither (Map a v) (Map b v)
  empty                        = (Nothing, Nothing)
  lookup (Left a) (fm1, fm2)   = lookup a fm1
  lookup (Right b) (fm1, fm2)  = lookup b fm2
```

To use the class *MapKey* on any specific algebraic data type, we need a map to its product/sum representation by means of an embedding-projection pair [19, 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 [27]. Since then, generic programming based on templates with traits has been found to be useful for self-optimising libraries [33] 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. [11]. 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 *ArrayElem* and *MapKey* examples, in which the container type depended on the element type, here the vertex and edge type depend on 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. Here are two possible instances which both model vertexes 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 vertexes 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)
```

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, motivated by Haskell's hierarchical standard library, is that of a monad based on a state transformer that supports mutable references. We can base this interface on 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 v = RefIO (IORef v)
  newRef        = newIORef
  ...
instance RefM (ST s) where
  data Ref (ST s) v = RefST (STRef s v)
  newRef            = newSTRef
  ...
```

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

```
newRef :: RefM m ⇒ v → m (Ref m v)
```

A subtlety of the above code is that the definition of *Ref IO v* introduces a new type that by Haskell's type equality is not compatible with *IORef v*. However, sometimes we might like to use an existing type as an associated type instead of introducing a new type. This requires *associated type synonyms,* which we plan to discuss in a future 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 4.

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 4.4.

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 a)
```

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 ⇒ [T a] → (b a) → 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 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 close analogy between methods of a class and associated data types: methods introduce overloaded, or type-indexed, variables, while associated data type declarations introduce type-indexed types.

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 :: Array Bool → Bool
```

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

```
f :: Array e → e
```

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

```
f :: ArrayElem e ⇒ Array e → e
```

This validity check for programmer-supplied type annotations is conveniently performed as part of the kind-checking of these annotations. 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.

3.2 Associated types in data declarations

For consistency, the system must support using associated types everywhere, including within 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)
```

As just discussed, the type *Array e* is not a valid type for all *e*, so we must add a context to the declaration of *T*:

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

Haskell 98 already supports contexts on *data* declarations, whose effect is to add a context to the type of the data constructor, which makes it satisfy our validity principle:

```
C :: ArrayElem e ⇒ Array e → 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. Just as a top-level function that calls overloaded functions itself becomes overloaded, so a data type that mentions type-indexed types itself becomes type indexed. We call such type-indexed data types *associated top-level types.*

3.3 Translation example

An important feature of our system is that we can explain it by translation into System F. To give the idea, we now walk through the translation for the *MapKey* example in Section 2. Recall 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 $\star \to \star$; 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 instance for integer keys:

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

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

```
data MapInt v = MapInt (Patricia.Dict v)

dMapUnit :: CMapKey () MapInt
dMapUnit = CMapKey {
   empty                 = MapInt Patricia.emptyDict,
   lookup k (MapInt d) = Patricia.lookupDict k d
}
```

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 = λda.λdb.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 ⇒ Map (a,Int) v → a → v
f m x = lookup (x,42) m
```

The translation looks like this:

```
f :: forall a v mk. CMapKey a mk
      → MapPair mk MapInt v → a → v
f = λda.λm.λix.
      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.

3.4 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

```
DefaultArray :: ArrayElem e ⇒ BoxedArray e → 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.

4 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 the predicative fragment of System F. The type

Symbol Classes

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

Source declarations

$$\begin{array}{llll}
pgm & \rightarrow & \overline{data};\ \overline{cls};\ \overline{inst};\ \overline{val} & (\text{whole program}) \\
data & \rightarrow & \textbf{data}\ T\ \overline{\alpha}\ =\ C\ \overline{\tau} & (\text{data type decl}) \\
 & \mid & \textbf{data}\ \overline{D\ \alpha} \Rightarrow S\ \alpha\ \overline{\beta}\ =\ C\ \overline{\tau} & (\text{assoc. type decl}) \\
cls & \rightarrow & \textbf{class}\ D\ \alpha\ \textbf{where} & (\text{class decl}) \\
 & & \quad dsig;\ vsig & \\
inst & \rightarrow & \textbf{instance}\ \theta\ \textbf{where} & (\text{instance decl}) \\
 & & \quad adata;\ val & \\
val & \rightarrow & x\ =\ e & (\text{value binding}) \\
dsig & \rightarrow & \textbf{data}\ S\ \alpha\ \overline{\beta} & (\text{assoc. type sig}) \\
vsig & \rightarrow & x :: \sigma & (\text{class method sig}) \\
adata & \rightarrow & \textbf{data}\ S\ \tau\ \overline{\beta}\ =\ C\ \overline{\xi} & (\text{assoc. data type})
\end{array}$$

Source terms

$$\begin{array}{llll}
e & \rightarrow & v \mid e_1\ e_2 \mid \lambda x.e & (\text{term}) \\
 & \mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid e :: \sigma & \\
v & \rightarrow & x \mid C & (\text{identifier})
\end{array}$$

Source types

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

Constraints

$$\begin{array}{llll}
\pi & \rightarrow & D\ \alpha \mid D\ (\alpha\ \tau_1 \cdots \tau_n) & (\text{simple constraint}) \\
\phi & \rightarrow & D\ \tau \mid \pi \Rightarrow \phi & (\text{qualified constraint}) \\
\theta & \rightarrow & \phi \mid \forall\alpha.\theta & (\text{constraint scheme})
\end{array}$$

Environments

$$\begin{array}{llll}
\Gamma & \rightarrow & \overline{v{:}\sigma} & (\text{type environment}) \\
\Theta & \rightarrow & \overline{\theta} & (\text{instance environment})
\end{array}$$

Figure 1: Syntax of expressions and types

system is based on Jones' *Overloaded ML (OML)* [20, 21]. 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.

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 in a curried way, we also sometimes use equivalent overbar 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 $(\rightarrow)\ \tau_1\ \tau_2$.

There are three unusual features 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, and are syntactically separated with type constructors ranged over by S rather than T. Second, we syntactically distinguish two forms of top-level data type declaration: ordinary ones T; and top-level associated types S, that mention associated types in their right-hand side (see Section 3.2). In the declaration of associated types, whether in a **class** declaration or a top-level **data** declaration, the type indexes must come first. Third, the syntax of types τ includes η, the saturated application of an associated type to all its type indexes. (There can be further type arguments by way of the $\tau_1\ \tau_2$ production.)

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

- Each class has exactly one type parameter, one method, and one associated type.
- Each top-level associated type has exactly one type-index parameter.
- 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.

Loosening these restrictions is largely a matter of adding (a great many) overbars to the typing rules. Introducing superclasses is slightly less trivial, as Section 4.5 discusses.

4.2 Type checking

A key feature of our system is that the typing rules for expressions are very close to those of Haskell 98. 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 [21].

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

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

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

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

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

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

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

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

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

Figure 2: Standard type checking rules for expressions

The auxiliary judgement $\Theta \vdash \sigma$, which is used in rules $(\to I)$ and $(\forall E)$, is the kind-checking judgement, used when the system "guesses" a type to ensure that the type is well-kinded. In the interests of brevity, however, the rules of Figure 2 elide all mention of kinds, leaving only the well-formedness check that is distinctive to our system. Specifically, in a well-formed type, every associated type $S\,\tau$ must be in a context that satisfies the classes D to which S is associated—there can be more than one in the case of associated top-level data types. It is this side condition that rejects, for example, the typing

$$\Theta \mid \Gamma \vdash \lambda x.x : \forall\alpha.\mathit{Array}\;\alpha \to \mathit{Int}$$

This typing is invalid because the associated type *Array* α is meaningless without a corresponding *ArrayElem* α constraint. This is as *Array* is simply not defined on all types of kind $\star$, but only on the subset for which there is a *ArrayElem* instance. This is akin to a simple form of *refinement kinds* [9].

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 mono-types) is because it is needed to check the validity of the types of class methods.

Target declarations

$td \quad \to \quad (x{:}\upsilon) \;=\; w \;\mid\; \mathbf{data}\; T\;\overline{\alpha} \;=\; C\;\overline{\upsilon}$

Target terms

$w \quad \to \quad v \mid w_1\; w_2 \mid \lambda(x{:}\upsilon).w \mid \Lambda\alpha.w \mid w\;\upsilon$

$\qquad\;\; \mid \quad \mathbf{let}\; x{:}\upsilon = w_1 \;\mathbf{in}\; w_2$

Target types

$\upsilon \quad \to \quad T \mid \alpha \mid \upsilon_1\;\upsilon_2 \mid \forall\alpha.\upsilon$

Environments

Δ	$\to$	$\overline{d{:}\theta}$	(dictionary environment)
Ω	$\to$	$\overline{\omega}$	(associated-type environment)
ω	$\to$	$\forall\overline{\alpha}.(\eta \rightsquigarrow T\;\overline{\tau})$	
	$\mid$	$\eta \rightsquigarrow \alpha$	

Figure 3: Syntax for target terms and types

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

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

$$\frac{\Omega \vdash \tau_1 \rightsquigarrow \upsilon_1 \qquad \Omega \vdash \tau_2 \rightsquigarrow \upsilon_2}{\Omega \vdash \tau_1\ \tau_2 \rightsquigarrow \upsilon_1\ \upsilon_2} \qquad \frac{\Omega \vdash \sigma \rightsquigarrow \upsilon \qquad \alpha \notin \mathrm{Fv}(\Omega)}{\Omega \vdash \forall\alpha.\sigma \rightsquigarrow \forall\alpha.\upsilon} \qquad \frac{\Omega \vdash \pi \rightsquigarrow (\eta \rightsquigarrow \alpha),\ \upsilon_d \qquad \Omega[\eta \rightsquigarrow \alpha] \vdash \rho \rightsquigarrow \upsilon}{\Omega \vdash \pi \Rightarrow \rho \rightsquigarrow \forall\alpha.\upsilon_d \to \upsilon}(tr\pi)$$

$$\boxed{\Omega \vdash \pi \rightsquigarrow \upsilon} \qquad \boxed{\Omega \vdash \pi \rightsquigarrow (\eta \rightsquigarrow \alpha),\ \upsilon}$$

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

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

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

Figure 4: Translating types

$$\boxed{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w' : \sigma}$$

$$\frac{(v : \sigma) \in \Gamma}{\Omega \mid \Delta \mid \Gamma \vdash v \rightsquigarrow v : \sigma}(var) \qquad \frac{\Omega \mid \Delta \mid \Gamma \vdash e_1 \rightsquigarrow w_1 : \sigma_1 \qquad \Omega \mid \Delta \mid \Gamma[x : \sigma_1] \vdash e_2 \rightsquigarrow w_2 : \sigma_2 \qquad \Omega \vdash \sigma_1 \rightsquigarrow \upsilon}{\Omega \mid \Delta \mid \Gamma \vdash (\mathbf{let}\, x = e_1\ \mathbf{in}\, e_2) \rightsquigarrow (\mathbf{let}\, x{:}\upsilon = w_1\ \mathbf{in}\, w_2) : \sigma_2}(let)$$

$$\frac{\Omega \mid \Delta \mid \Gamma[x{:}\tau_1] \vdash e \rightsquigarrow w : \tau_2 \qquad \Omega \vdash \tau_1 \rightsquigarrow \upsilon_1}{\Omega \mid \Delta \mid \Gamma \vdash (\lambda x.e) \rightsquigarrow (\lambda x{:}\upsilon_1.w) : \tau_1 \to \tau_2}(\to I) \qquad \frac{\Omega \mid \Delta \mid \Gamma \vdash e_1 \rightsquigarrow w_1 : \tau_2 \to \tau_1 \qquad \Omega \mid \Delta \mid \Gamma \vdash e_2 \rightsquigarrow w_2 : \tau_2}{\Omega \mid \Delta \mid \Gamma \vdash (e_1\ e_2) \rightsquigarrow (w_1\ w_2) : \tau_1}(\to E)$$

$$\frac{\Omega \vdash \pi \rightsquigarrow (\eta \rightsquigarrow \alpha),\ \upsilon \qquad \Omega[\eta \rightsquigarrow \alpha] \mid \Delta[d{:}\pi] \mid \Gamma \vdash e \rightsquigarrow w : \rho}{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow (\Lambda\alpha.\lambda(d{:}\upsilon).w) : \pi \Rightarrow \rho}(\Rightarrow I) \qquad \frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \pi \Rightarrow \rho \quad \Omega \mid \Delta \Vdash \pi \rightsquigarrow w' \quad \Omega \vdash \pi \rightsquigarrow \upsilon}{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w\ \upsilon\ w' : \rho}(\Rightarrow E)$$

$$\frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \sigma \quad \alpha \notin \mathrm{Fv}(\Delta) \cup \mathrm{Fv}(\Gamma)}{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow (\Lambda\alpha.w) : \forall\alpha.\sigma}(\forall I) \qquad \frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \forall\alpha.\sigma \quad \Omega \vdash \tau \rightsquigarrow \upsilon}{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w\ \upsilon : [\tau/\alpha]\sigma}(\forall E) \qquad \frac{\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \sigma}{\Omega \mid \Delta \mid \Gamma \vdash (e :: \sigma) \rightsquigarrow w : \sigma}(sig)$$

Figure 5: Typing rules with translation

4.3 Evidence translation

A second crucial feature of our system is that, like Haskell 98 [10], 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 3.3. In this section we formalise the translation.

4.3.1 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 [12].

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 4.3.2.

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 variable that stands for π's associated type (i.e., only the one directly associated with the class mentioned in π; associated top-level types need not be abstracted). As well as augmenting the dictionary environment Δ to reflect the constraint that is now satisfied by the environment including its witness d, we augment the type-translation environment Ω to explain how π's associated type (called η) may be rewritten.

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

4.3.2 Translating types

The translation of source types to target types is formalised by the judgement $\Omega \vdash \sigma \rightsquigarrow \upsilon$ 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 3.3:

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

The first example is straightforward, because it arises directly from the **instance** declaration for *MapKey Int*. There is more going on in the second example. The class constraint is translated to an ordinary function, with argument type *CMapKey* α β, where the data type *CMapKey* 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* (α, Int) is first translated to *MapPair* $(\mathit{Map}\ \alpha)$ $(\mathit{Map\ Int})$, by applying the translation scheme added to Ω when translating the instance declaration for pairs. Then $(\mathit{Map\ Int})$ 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 $\forall\overline{\alpha}.S\ \xi \rightsquigarrow T\ \overline{\tau}$, 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 3.3. The instances for *Int* and pairs augment Ω with the assumptions:

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

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 $S\ \tau \rightsquigarrow \beta$ — 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 $S\ \tau \rightsquigarrow \alpha$, we use a judgement of the form $\Omega \vdash \pi \rightsquigarrow (\eta \rightsquigarrow \alpha),\ \upsilon$ (from Figure 4). Such a judgement 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 υ.

4.3.3 Data type and value declarations

The rules for type-directed translation of declarations are given in Figure 6. They are somewhat complex, 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 vanilla data type declarations is easy: all we need to do is translate the constructor argument types, using our auxiliary type-translation judgement $\Omega \vdash \tau \rightsquigarrow \upsilon$. The handling of associated top-level data types is more involved, but their treatment closely mirrors that of associated types in instance declarations, which we discuss below. Value declarations are also straightforward, because all of the work is done in Figures 4 and 5.

4.3.4 Class declarations

The interesting cases are class and instance declarations. It may help to refer back to the example of Section 3.3 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 σ as its argument type, suitably translated of course. The translation uses an

$$\boxed{\Omega \vdash \mathit{data} \leadsto \mathit{td} : \Omega, \Gamma}$$

$$\dfrac{\Omega \vdash \overline{\tau \leadsto \upsilon}^{c}}{\Omega \vdash \textbf{data}\ T\ \overline{\alpha} = C\ \overline{\tau}^{c} \quad \leadsto \quad \textbf{data}\ T\ \overline{\alpha} = C\ \overline{\upsilon}^{c} : [], [C : \forall\overline{\alpha}.\overline{\tau}^{c} \to T\ \overline{\alpha},\ \overline{\mathit{prj}^{C} : \forall\overline{\alpha}.T\ \overline{\alpha} \to \tau}^{c}]}\,(\mathit{data})$$

$$\dfrac{\Omega \vdash \overline{\pi \leadsto (\eta \leadsto \beta),\ \upsilon_d}^{r} \qquad \Omega' = \Omega[\overline{\eta \leadsto \beta}^{r}] \qquad \Omega' \vdash \overline{\tau_c \leadsto \upsilon_c}^{c}}{\Omega \vdash \textbf{data}\ \overline{\pi}^{r} \Rightarrow S\ \gamma\ \overline{\alpha} = C\ \overline{\tau_c}^{c} \quad \leadsto \quad \textbf{data}\ T\ \gamma\ \overline{\beta}^{r}\ \overline{\alpha} = C\ \overline{\upsilon_c}^{c} : \begin{array}{l} [\forall\gamma.\ (S\ \gamma \leadsto T\ \gamma\ \overline{\eta}^{r})], \\ [C : \forall\gamma\overline{\alpha}.\overline{\pi}^{r} \Rightarrow \overline{\tau_c}^{c} \to S\ \gamma\ \overline{\alpha},\ \overline{\mathit{prj}^{C} : \forall\gamma\overline{\alpha}.\overline{\pi}^{r} \Rightarrow S\ \gamma\ \overline{\alpha} \to \tau_c}^{c}] \end{array}}\,(\mathit{adata})$$

$$\boxed{\Omega \vdash \mathit{cls} \leadsto \overline{\mathit{td}} : \Gamma}$$

$$\dfrac{\beta \text{ fresh} \qquad \Omega' = \Omega[S\ \alpha \leadsto \beta] \qquad \Omega' \vdash \sigma \leadsto \upsilon \qquad \sigma = \forall\overline{\delta}.\overline{\pi} \Rightarrow \tau \text{ where } \alpha \notin \mathrm{Fv}(\overline{\pi})}{\Omega \vdash \begin{array}{l} \quad\textbf{class}\ D\ \alpha\ \textbf{where} \\ \qquad\textbf{data}\ S\ \alpha\ \overline{\gamma} \\ \qquad f :: \sigma \end{array} \quad \leadsto \quad \begin{array}{l} \textbf{data}\ D\ \alpha\ \beta = C_D\ \upsilon \\ (f : \forall\alpha\beta.D\ \alpha\ \beta \to \upsilon) = \mathrm{prj}^{C_D} \end{array} : [f : \forall\alpha.D\ \alpha \Rightarrow \sigma]}\,(\mathit{cls})$$

$$\boxed{\Omega \mid \Delta \mid \Gamma \vdash \mathit{inst} \leadsto \overline{\mathit{td}} : \Omega, \Delta, \Gamma}$$

$$\dfrac{\begin{array}{c} \Omega \vdash \overline{\pi \leadsto (\eta \leadsto \beta),\ \upsilon_d}^{r} \qquad \Omega' = \Omega[\overline{\eta \leadsto \beta}^{r}] \qquad \Delta' = \Delta[\overline{d : \pi}^{r}] \qquad \Omega' \vdash \overline{\tau_c}^{c} \leadsto \overline{\upsilon_c}^{c} \\ (f : \forall\delta.D\ \delta \Rightarrow \sigma) \in \Gamma \qquad \Omega' \mid \Delta' \mid \Gamma \vdash e \leadsto w : [\tau/\delta]\sigma \\ \overline{d}^{r} \text{ fresh} \qquad d_f \text{ fresh} \qquad T \text{ fresh} \end{array}}{\Omega \mid \Delta \mid \Gamma \vdash \begin{array}{l} \textbf{instance}\ \forall\overline{\alpha}^{s}.\overline{\pi}^{r} \Rightarrow D\ \tau\ \textbf{where} \\ \quad \textbf{data}\ S\ \tau\ \overline{\gamma} = C\ \overline{\tau_c}^{c} \\ \quad f = e \end{array} \quad \leadsto \quad \begin{array}{l} \textbf{data}\ T\ \overline{\alpha}^{s}\ \overline{\beta}^{r}\ \overline{\gamma} = C\ \overline{\upsilon_c}^{c} \\ d_f = \Lambda\overline{\alpha}^{s}\overline{\beta}^{r}.\lambda\overline{d : \upsilon_d}^{r}.C_D\ w \end{array} \quad : \quad \begin{array}{l} [\forall\overline{\alpha}^{s}.\ (S\ \tau \leadsto T\ \overline{\alpha}^{s}\ \overline{\eta}^{r})], \\ [d_f : \forall\overline{\alpha}^{s}.\ \overline{\pi}^{r} \Rightarrow D\ \tau]), \\ [C : \forall\overline{\alpha}^{s}\overline{\gamma}.\ \overline{\pi}^{r} \Rightarrow \overline{\tau_c}^{c} \to S\ \tau\ \overline{\gamma}, \\ \overline{\mathit{prj}^{C} : \forall\overline{\alpha}^{s}\overline{\gamma}.\overline{\pi}^{r} \Rightarrow S\ \tau\ \overline{\gamma} \to \tau_c}^{c}] \end{array}}\,(\mathit{inst})$$

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

$$\dfrac{\Omega \mid \Delta \mid \Gamma \vdash e \leadsto w : \sigma \qquad \Omega \vdash \sigma \leadsto \upsilon}{\Omega \mid \Delta \mid \Gamma \vdash (x = e) \leadsto (x{:}\upsilon = w) : [x : \sigma]}\,(\mathit{val})$$

$$\boxed{\vdash \mathit{pgm} \leadsto \overline{\mathit{td}}}$$

$$\dfrac{\begin{array}{c} \Omega = \Omega_d, \Omega_i \qquad \Gamma = \Gamma_d, \Gamma_c, \Gamma_i, \Gamma_v \\ \overline{\Omega \vdash \mathit{data} \leadsto d_d : \Omega_d, \Gamma_d} \qquad \overline{\Omega \vdash \mathit{cls} \leadsto \overline{d_c} : \Gamma_c} \qquad \overline{\Omega \mid \Delta \mid \Gamma \vdash \mathit{inst} \leadsto \overline{d_i} : \Omega_i, \Delta, \Gamma_i} \qquad \overline{\Omega \mid \Delta \mid \Gamma \vdash \mathit{val} \leadsto \overline{d_v} : \Gamma_v} \end{array}}{\vdash \overline{\mathit{data}};\ \overline{\mathit{cls}};\ \overline{\mathit{inst}};\ \overline{\mathit{val}} \leadsto \overline{d_d};\ \overline{d_c};\ \overline{d_i};\ \overline{d_v}}$$

Figure 6: Declaration typing rules with translation

associated-type environment Ω' that maps each associated type to a fresh type variable β. The data type must be parameterised over these fresh β, because they will presumably be free in the translated method types υ. Finally, we must generate a binding for the method selector function for the class method f; in the rule, this is 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.

We impose the same restriction on method types that Haskell 98 does, namely that constraints in the method type σ must not constrain the class parameter α. Lifting this restriction would permit classes like this one:

class $D\ a$ **where**
  $op :: C\ a \Rightarrow a \rightarrow T\ a$

where the constraint $C\ a$ constrains only the class variable a. In the functional-dependency setting, classes like these are known to be tricky, and the situation is the same for us. In this paper we simply exclude the possibility.

4.3.5 Instance declarations

Instance declarations are more involved. 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. This data type must be parameterised over (a) the quantified type variables of the instance declaration itself, $\overline{\alpha}$, (b) a type variable for each associated type of each constraint in the instance declaration, $\overline{\beta}$, and (c) the type variables in which the associated type is parametric in all instances, $\overline{\gamma}$. Here is an artificial example to demonstrate the possibilities:

class $MK\ k$ **where**
  data $M\ k\ v$
instance $(MK\ a,\ MK\ b) \Rightarrow MK\ (a,b)$ **where**
  data $M\ (a,b)\ v =\ MP\ v\ a\ (M\ b)\ b$

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

data $M'\ a\ b\ ma\ mb\ v\ =\ MP\ v\ a\ mb\ b$

The arguments ma, mb were the $\overline{\beta}$ 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 υ_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, 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.

4.3.6 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 slightly 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 Δ, 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.

4.4 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 $\overline{\gamma}$. Now we can see why. The class parameters must occur *first* so that we can insist that associated-type applications are saturated (w.r.t. 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 $\overline{\gamma}$.

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.

4.5 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 type argument β, which corresponds to the associated type of the class (in general, there can be multiple associated types, and hence, multiple such type arguments). 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.

4.6 Soundness

As the evidence translation maps programs from a typed source into a typed intermediate language, we expect it to generate only well typed programs. That this expectation is met is asserted by the formal results sketched in the following (full details are in a companion technical report). Type checking of target declarations $\overline{td}$ and target terms w is denoted by $\vdash_F \overline{td}$ and $\Gamma \vdash_F w : \upsilon$, respectively, where Γ is a target type environment and υ a target type. The type checking rules are standard for a type passing lambda calculus and omitted for space reasons. Moreover, we lift the type translation judgement $\Omega \vdash \sigma \rightsquigarrow \upsilon$ pointwise to translate source to target environments.

THEOREM 1. *Given a source program pgm, if we can translate it as* $\vdash pgm \rightsquigarrow \overline{td}$*, the resulting target program is well typed; i.e.,* $\vdash_F \overline{td}$.

PROOF. The proof considers the target declaration producing rules from Figure 6 in turn and demonstrates that the type environment produced for the source program corresponds to that produced for the target program by the type translation judgement $\Omega \vdash \sigma \rightsquigarrow \upsilon$. $\square$

THEOREM 2. *Given a type translation environment* Ω*, dictionary environment* Δ*, and type environment* Γ*, if a source term* e *of type* σ *translates as* $\Omega \mid \Delta \mid \Gamma \vdash e \rightsquigarrow w : \sigma$*, its type as* $\Omega \vdash \sigma \rightsquigarrow \upsilon$*, and the environment as* $\Omega \vdash \Gamma \rightsquigarrow \Gamma_F$*, then we have* $\Gamma_F \vdash_F w : \upsilon$.

PROOF. The proof proceeds by rule induction over the target term producing translation rules. The tricky cases are those for rules $(\Rightarrow I)$ and $(\Rightarrow E)$, where we abstract over types associated with the class of a context and supply corresponding representation types, respectively. Moreover, we need to make use of some auxiliary properties of the judgments of Figure 4. $\square$

5 Comparison to Functional Dependencies

Functional dependencies [22] 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. Indeed, the extra type parameters introduced implicitly by our System-F translation appear explicitly when the program is expressed using functional dependencies. For example, using FDs one might express the *Array* example like this:

```
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 use of functional dependencies to describe type-indexed data types suffers from three serious shortcomings, which we shall discuss now.

Undecidable type constraints. As Duck et al. [8] point out, the instance for pairs *ArrayRep* (a, b) (arr, brr) is problematic, 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' [22] original proposal of functional dependencies does not allow such instances.

Clutter. In the comparative study of Garcia et al. [11], 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 associates 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

```
index :: ArrayRep e arr ⇒ arr → Int → e
```

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

```
indexInt :: ArrayRep Int arr ⇒ arr → Int → Int
indexInt = index
```

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 $e \to arr$ (cf. the class declaration of *ArrayRep*). In fact, we are forced to use the following definition instead:

$$\begin{aligned} &indexInt' :: UArrInt \to Int \to Int \\ &indexInt' = index \end{aligned}$$

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 [13].

Variations on functional dependencies. Duck et al. [8] 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) [30] framework with constraint handling rules. Stuckey & Sulzmann [36] 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.

Neubauer et al. [28] 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 [29], where they add a full-blown functional logic language to the type system, based on the HM(X) [30] framework. 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.

6 Related Work

Type classes. There is a significant amount of previous work that studies the relationship between type classes and type-indexed functions [37, 19, 1, 23, 24], 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.

Generic Haskell. Hinze et al [16, 18] 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. [26] elaborated on the original design and introduced *Dependency-style Generic Haskell.* In fact, our translation of associated types to additional type parameters is akin to the translation of type-indexed types in Generic Haskell. Associated types are a more lightweight extension to Haskell, but they miss the automatic generation of embedding projection pairs.

ML modules. It has been repeatedly observed that there is a significant overlap in functionality between Haskell type classes and Standard ML modules. The introduction of associated types has increased this overlap; ML modules have always been an agglomeration of both values and types. Nevertheless, there are interesting differences between type classes and ML modules. In particular, ML structures are a term-level entity and hence a notion of *phase distinction* [14] is required to separate static from dynamic semantics, with Leroy [25] proposing a variant. 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 [7] and it is far from clear how the different concepts relate to type classes. Despite these differences, the introduction of associated types shows that the commonality between type classes and ML modules may be more significant than previously assumed. Hence, it would be worthwhile to investigate this relationship in more detail.

Intensional type analysis. *Intensional type analysis* [15] realises type-indexed types by a type-level Typerec construct and has been proposed to facilitate the type-preserving optimisation of polymorphism. Subsequent work [32, 6, 5, 38] elaborated on Harper and Morrisett's seminal work that 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 types, whereas the target language of our evidence translation has a standard type-erasure semantics. Nevertheless, Crary et al. [6] proposed an alternative view on intensional type analysis based on type erasure and we need to pass method dictionaries at runtime, which can be regarded as an implicit type representation.

Constrained data types. Xi et al. [39] 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.

Refinement kinds. In Section 4.2, we said that we understand a type constraint of the form $D\ \alpha$ as a restriction of the range of the variable α; i.e., α only ranges over a subset of the types characterised by kind $\star$. In particular, it restricts α such that the term $S\ \alpha$ is well-defined on α's entire range if S is an associated type of D. This subkinding relationship is related to Duggan's notion of refinement kinds [9]. However, Duggan only considers type-indexed functions, but not type-indexed types.

Object-oriented languages. As we mentioned in Section 2.2, associated types have a long standing tradition in C++ and are often collected in *traits classes* [27]. Garcia et al. [11] 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 it is beyond the scope of this paper to review all of it.

7 Conclusions

We propose to include type declarations alongside 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. In particular, the target language is not affected by the extension of the source language.

In future work we hope to extend the mechanism from associated *data types* to associated *type synonyms*, a generalisation that has substantial implications. 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 particularly thank Martin Sulzmann for his detailed and constructive comments on earlier drafts of the paper. We also thank Dave Abrahams, Brian McNamara, and Jeremy Siek for an interesting email exchange comparing Haskell type classes with C++ classes. We thank Robert Harper, Greg Morrisett, André Pang and the anonymous ICFP referees for feedback on a previous version of this paper.

8 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 Crary and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, 1999.

[6] Karl Crary, 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] 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.

[8] 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.

[9] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *Transactions on Programming Languages and Systems*, 21(1):11–45, 1999.

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

[11] 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.

[12] 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.

[13] Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70(1):1–5, 1999.

[14] 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.

[15] 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.

[16] Ralf Hinze. Generalizing generalized tries. *Journal of Functional Programming*, 10(4):327–351, 2000.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st Symposium Principles of Programming Languages*, pages 109–122. ACM Press, 1994.

[26] 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.

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

[28] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A functional notation for functional dependencies. In *2001 ACM SIGPLAN Haskell Workshop*, 2001.

[29] 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.

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

[31] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *Workshop on ML*, pages 77–86, 1998.

[32] Zhong Shao. Flexible representation analysis. In *Proceedings ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, 1997.

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

[34] 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.

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

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

[37] Stephanie Weirich. Type-safe cast: Functional pearl. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. ACM Press, 2000.

[38] Stephanie Weirich. Higher-order intensional type analysis. In *European Symposium on Programming (ESOP02)*, 2002.

[39] 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.