Derivable Type classes

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* `ralf@informatik.uni-bonn.de`)

SIMON PEYTON JONES
Microsoft Research Ltd
St George House, 1 Guildhall St, Cambridge CB2 3NH, England
(*e-mail:* `simonpj@microsoft.com`)

Abstract

Generic programming allows you to write a function once, and use it many times at different types. A lot of good foundational work on generic programming has been done. The goal of this paper is to propose a practical way of supporting generic programming within the Haskell language, while going "with the grain" of the language.

On the way, we came across a separate issue, concerning type-class overloading where higher kinds are involved. We propose a simple type-class system extension to allow the programmer to write richer contexts than is currently possible.

1 Introduction

A *generic*, or *polytypic*, function is one that the programmer writes once, but which works over many different data types. The standard examples are parsing and printing, serialising, taking equality, ordering, hashing, and so on. There is lots of work on generic programming (Bird *et al.*, 1996; Jansson & Jeuring, 1997; Backhouse *et al.*, 1999; Hinze, 2000b).

In this paper we present the design and implementation of an extension to Haskell that supports generic programming. At first sight it might seem that Haskell's type classes are in competition with generic programming — after all, both concern functions that work over many data types. But we have found that they can be combined very gracefully, offering a smooth upward extension to Haskell.

On the way we describe an orthogonal, but complementary idea. Haskell allows one to define higher-order kinded data types for which it is impossible to define, for example, an equality instance. This seems unfortunate: one part of the language is more powerful than another. We describe a modest extension of Haskell's type-class system that removes this difficulty.

More specifically, our contributions are these:

- We present the language design for an extension to Haskell that supports generic programming (Sections 2–6). Generic functions appear solely in class declarations.
- We describe an entirely separate extension that lets one write certain instance declarations for higher-order kinded data types that are simply inexpressible in Haskell 98 (Sections 7–8).
- We discuss the implementation of both parts.

The first part of this paper is a follow-up to (Hinze, 1999); the new achievements are detailed in Section 9.

2 Setting the scene

In this section we set the context for our proposal. We begin by reviewing Haskell's type-class overloading mechanism.

2.1 A brief review of type class overloading

Haskell supports overloading, based on type classes. For example, the Prelude defines the class *Eq*:

$$\begin{array}{l} \textbf{class } Eq\ t \textbf{ where} \\ \quad (==), (/=) \quad :: \quad t \to t \to Bool \ . \end{array}$$

This defines two overloaded top-level functions, $(==)$ and $(/=)$, whose types are

$$(==), (/=) \quad :: \quad (Eq\ t) \Rightarrow t \to t \to Bool \ .$$

Before we can use $(==)$ on values of, say *Int*, we must explain how to take equality over *Int* values:

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{instance } Eq\ Int \textbf{ where}} \\ \quad (==) & = & eqInt \\ \quad (/=) & = & neInt \ . \end{array}$$

Here we suppose that $eqInt :: Int \to Int \to Bool$, and similarly *neInt* are provided from somewhere. The **instance** declarations says "the $(==)$ function at type *Int* is implemented by *eqInt*".

How can we take equality of pairs of values? Presumably by comparing their components; but that requires equality over the component types:

$$\begin{array}{lcl} \multicolumn{3}{l}{\textbf{instance } (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b) \textbf{ where}} \\ \quad (x_1, y_1) == (x_2, y_2) & = & (x_1 == x_2) \wedge (y_1 == y_2) \\ \quad (x_1, y_1) \mathrel{/=} (x_2, y_2) & = & (x_1 \mathrel{/=} x_2) \vee (y_1 \mathrel{/=} y_2) \ . \end{array}$$

It is a bit annoying to keep having to write code for the $(/=)$ method, since it is simply the negation of the code for the (==) method, so Haskell allows you to write a *default method* in the class declaration:

$$\begin{array}{lll}\multicolumn{3}{l}{\textbf{class}\ Eq\ t\ \textbf{where}}\\ \quad (==), (/=) & :: & t \to t \to Bool\\ \quad x_1 \mathrel{/=} x_2 & = & not\ (x_1 == x_2)\ .\end{array}$$

Now, if you give an instance declaration for *Eq* that lacks a definition for (/=), Haskell "fills in" the missing method definition with code copied from the class declaration. So we can write:

$$\begin{array}{lll}\multicolumn{3}{l}{\textbf{instance}\ (Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)\ \textbf{where}}\\ \quad (x_1, y_1) == (x_2, y_2) & = & (x_1 == x_2) \wedge (y_1 == y_2)\end{array}$$

and get just the same effect as before. You can even specify a default method for both methods:

$$\begin{array}{lll}\multicolumn{3}{l}{\textbf{class}\ Eq\ t\ \textbf{where}}\\ \quad (==), (/=) & :: & t \to t \to Bool\\ \quad x_1 == x_2 & = & not\ (x_1 \mathrel{/=} x_2)\\ \quad x_1 \mathrel{/=} x_2 & = & not\ (x_1 == x_2)\ .\end{array}$$

In an instance declaration, you can now *either* give a definition for (==), *or* a definition for (/=), or both. If you specify neither, then you will get an infinite loop, unfortunately!

If you give an instance declaration without specifying code for method *op*, and the class has no default method for *op*, then invoking the method will halt the program with an error message. It is not a compile-time error; sometimes a method just doesn't make sense for a particular instance type.

2.2 Overloading is not generic programming

Haskell as it stands does not support generic, or polytypic, programming. In particular, suppose you define a new data type:

$$\textbf{data}\ Wibble \quad = \quad Wag\ Int \mid Wug\ Bool\ .$$

It is "obvious" how to take equality over *Wibble*, and support for generic programming would allow us to specify this "obvious" precisely. In Haskell, however, you have to give an explicit instance declaration, containing the code for equality:

$$\begin{array}{lll}\multicolumn{3}{l}{\textbf{instance}\ Eq\ Wibble\ \textbf{where}}\\ \quad (Wag\ i_1) == (Wag\ i_2) & = & i_1 == i_2\\ \quad (Wug\ b_1) == (Wug\ b_2) & = & b_1 == b_2\\ \quad w_1 == w_2 & = & False\ .\end{array}$$

Cranking out this sort of boilerplate code is so tiresome that Haskell provides special support — the so-called "**deriving**" mechanism — for a handful of built-in classes.

In particular, for *Eq* you say

$$\textbf{data}\ \mathit{Wibble} \quad = \quad \mathit{Wag\ Int} \mid \mathit{Wug\ Bool}\ \textbf{deriving}\ (\mathit{Eq})\ \ .$$

The "**deriving** (*Eq*)" part tells Haskell to generate the "obvious" code for equality. What "obvious" means is specified informally in an Appendix of the language definition (Peyton Jones & Hughes, 1999). This is all rather *ad hoc*, and in particular *it only works for a fixed set of built-in classes* (*Eq*, *Ord*, *Enum*, *Bounded*, *Read*, *Show*, and *Ix*).

2.3 Generalising default methods

What we seek, then, is an automatic mechanism that "fills in" a suitable implementation for the methods of an instance declaration. But wait a minute! That's what a default method does! Indeed so but, as we have already remarked, default methods as they stand are too weak. If we write merely:

$$\textbf{instance}\ \mathit{Eq\ Wibble}$$

we would, as remarked earlier, just get an infinite loop. We have to provide some real code somewhere! What we want is a richer language in which to write default methods. That is what we turn our attention to now.

3 Generic definitions

From a language-design point of view, our story is this: *providing a richer language for default method definitions in a class declaration gives an elegant way to extend Haskell with the power of generic programming.* We will justify this statement more fully in Section 4.4, but first we must present our design.

3.1 Two examples

We adopt with minor changes the proposal in (Hinze, 1999). Two examples will serve to give the idea. First, here is the *Eq* class augmented with generic equality:

$$\begin{array}{lcl}
\textbf{class}\ \mathit{Eq}\ t\ \textbf{where} & & \\
\quad (==), (/=) & :: & t \to t \to \mathit{Bool} \\
\qquad \text{-- generic default method} & & \\
\quad (==)\{\,1\,\}\mathit{Unit}\ \mathit{Unit} & = & \mathit{True} \\
\quad (==)\{\,a+b\,\}(\mathit{Inl}\ x_1)\ (\mathit{Inl}\ x_2) & = & x_1 == x_2 \\
\quad (==)\{\,a+b\,\}(\mathit{Inr}\ y_1)\ (\mathit{Inr}\ y_2) & = & y_1 == y_2 \\
\quad (==)\{\,a+b\,\}\_\ \_ & = & \mathit{False} \\
\quad (==)\{\,a\times b\,\}(x_1 :\!*\!: y_1)\ (x_2 :\!*\!: y_2) & = & x_1 == x_2 \wedge y_1 == y_2 \\
\qquad \text{-- vanilla, non-generic default method} & & \\
\quad (/=)\ x_1\ x_2 & = & \mathit{not}\ (x_1 == x_2)\ \ .
\end{array}$$

This new class declaration contains an ordinary, default declaration for $(/=)$, just as before. The new feature is a *generic definition* for equality, distinguished by the curly braces on the left hand side, which enclose a *type* argument. We will study such generic definitions in more detail in Section 4.3. For now, we simply observe that a generic definition works by induction over the structure of the type (written in curly braces) at which the class is instantiated.

Here is another example. The class *Binary* has methods *showBin* and *readBin* that respectively convert a value to a list of bits and vice versa:

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Bit} & = & \mathtt{0} \mid \mathtt{1} \\
\textbf{type}\ \mathit{Bin} & = & [\mathit{Bit}] \\
\multicolumn{3}{l}{\textbf{class}\ \mathit{Binary}\ t\ \textbf{where}} \\
\quad \mathit{showBin} & :: & t \to \mathit{Bin} \\
\quad \mathit{readBin} & :: & \mathit{Bin} \to (t, \mathit{Bin})\ .
\end{array}
$$

A real implementation might have a more sophisticated representation for *Bin* but that is a separate matter. We can give generic definitions for *showBin* and *readBin* like this:

$$
\begin{array}{lll}
\mathit{showBin}\{1\}\,\mathit{Unit} & = & [\,] \\
\mathit{showBin}\{a+b\}(\mathit{Inl}\ x) & = & \mathtt{0} : \mathit{showBin}\ x \\
\mathit{showBin}\{a+b\}(\mathit{Inr}\ y) & = & \mathtt{1} : \mathit{showBin}\ y \\
\mathit{showBin}\{a\times b\}(x :\!*\!: y) & = & \mathit{showBin}\ x +\!\!+\ \mathit{showBin}\ y \\
\mathit{readBin}\{1\}\mathit{bs} & = & (\mathit{Unit}, \mathit{bs}) \\
\mathit{readBin}\{a+b\}[\,] & = & \mathit{error}\ \texttt{"readBin:␣unexpected␣end␣of␣input"} \\
\mathit{readBin}\{a+b\}(\mathtt{0} : \mathit{bs}) & = & \textbf{let}\ (x, \mathit{bs}') = \mathit{readBin}\ \mathit{bs}\ \textbf{in}\ (\mathit{Inl}\ x, \mathit{bs}') \\
\mathit{readBin}\{a+b\}(\mathtt{1} : \mathit{bs}) & = & \textbf{let}\ (y, \mathit{bs}') = \mathit{readBin}\ \mathit{bs}\ \textbf{in}\ (\mathit{Inr}\ y, \mathit{bs}') \\
\mathit{readBin}\{a\times b\}\mathit{bs} & = & \textbf{let}\ (x, \mathit{bs}_1) = \mathit{readBin}\ \mathit{bs} \\
 & & \qquad (y, \mathit{bs}_2) = \mathit{readBin}\ \mathit{bs}_1 \\
 & & \textbf{in}\ ((x :\!*\!: y), \mathit{bs}_2)\ .
\end{array}
$$

Notice that *readBin* *produces* a value of the unknown type t, whereas *showBin* and $(==)$ both *consume* such values.

3.2 Instances and **deriving**

Suppose we have given a class declaration of *Eq* with generic default methods, as we did in the previous section. Now we can give an instance declaration like this:

$$\textbf{instance}\ \mathit{Eq}\ \mathit{Wibble}$$

without giving code for either method. Both methods will be "filled in" from the class declaration. The ordinary, non-generic default method, $(/=)$, is filled in verbatim. The generic default method, $(==)$, is specialised in a way we will describe, to give essentially the code in Section 2.2. That is, the effect of the **instance** declaration is exactly as if we had written

$$\begin{array}{lcl}\multicolumn{3}{l}{\textbf{instance}\ \mathit{Eq}\ \mathit{Wibble}\ \textbf{where}}\\ \quad(\mathit{Wag}\ i_1) == (\mathit{Wag}\ i_2) & = & i_1 == i_2\\ \quad(\mathit{Wug}\ b_1) == (\mathit{Wug}\ b_2) & = & b_1 == b_2\\ \quad w_1 == w_2 & = & \mathit{False}\\ \quad w_1 \mathrel{/=} w_2 & = & \mathit{not}\ (w_1 == w_2)\ .\end{array}$$

And that is all. Though this sounds simple enough, it has interesting and important consequences:

- Though the instance declaration is now rather brief, it must still be given to make *Wibble* an instance of *Eq*. It is *not* the case that all types become instances of *Eq* when one gives a generic default method.
- It is not necessary for the instance declaration to appear in the same module as the data type declaration, or the class declaration. By contrast, in Haskell 98 a **deriving** clause must be attached to the data type declaration. This separation is useful, because the class might not even be defined in the scope where the type is declared.
- The compiler only fills in a method definition if the programmer omits it. For example, if we said

  $$\begin{array}{lcl}\multicolumn{3}{l}{\textbf{instance}\ \mathit{Eq}\ \mathit{Wibble}\ \textbf{where}}\\ \quad(\mathit{Wag}\ \_) == (\mathit{Wag}\ \_) & = & \mathit{True}\\ \quad w_2 == w_2 & = & \mathit{False}\end{array}$$

  then this programmer-supplied code is, of course, used. This is one way in which our proposal differs from others. In most generic-programming systems, a generic function works generically over all types. In our design, *the programmer can override the generic definition on a type-by-type basis.*
  In particular, this is how we specify instances for primitive types such as *Int*, *Double*, "$\rightarrow$": we just use ordinary instance declarations.
- A **deriving** clause can now be seen as shorthand (albeit now not so much shorter) for an instance declaration. There is a difference, though. Consider

  $$\textbf{data}\ \mathit{Tree}\ a \quad = \quad \mathit{Node}\ a\ [\mathit{Tree}\ a]\ \textbf{deriving}\ (\mathit{Eq})\ .$$

  In our design, the **deriving** clause is shorthand for

  $$\textbf{instance}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Eq}\ (\mathit{Tree}\ a)$$

  Note that in an instance declaration we must explicitly specify the context ($\mathit{Eq}\ a$), which is inferred automatically by the **deriving** mechanism.
  There are two quite separate questions here:
  — Inferring the right context for the instance declaration.
  — Filling in the code for the methods.

  This paper addresses the second of these questions, and not the first. The first is certainly an interesting question in its own right, and not just for instance declarations. For example, it might be desirable to allow one to say

$$f \quad :: \quad (...) \Rightarrow a \rightarrow a$$

where the "(...)" means "please infer the appropriate context". The ability to write such partial type signatures, with the ellipsis filled in by type inference, has been discussed on the Haskell mailing list. Such an ellipsis makes sense wherever a context is explicitly written in Haskell. In the case of instance declarations, we might like to be able to write

$$\textbf{instance}\ (...) \Rightarrow \mathit{Eq}\ (\mathit{Tree}\ a)$$

and have Haskell fill in the ellipsis, as well as the method declarations. This is quite feasible (albeit involving a fixpoint iteration) for first-order kinded types, but we believe that it is infeasible for higher-order types (see Section 7.3).

3.3 Generic representation types

The arguments in braces on the left hand side of a generic definition are *types*. The idea is, of course, that these generic definitions can be specialised for any particular type. Suppose, for example, we have a data type *List*, and we make *List* an instance of *Binary*:

$$\begin{array}{l}\textbf{data}\ \mathit{List}\ a \quad = \quad \mathit{Cons}\ a\ (\mathit{List}\ a) \mid \mathit{Nil} \\ \textbf{instance}\ (\mathit{Binary}\ a) \Rightarrow \mathit{Binary}\ (\mathit{List}\ a)\ .\end{array}$$

How is the compiler to fill in the missing method definitions?

First, we define the *generic representation type* for *List*, which we will call *List*°:

$$\textbf{type}\ \mathit{List}^{\circ}\ a \quad = \quad (a \times \mathit{List}\ a) + 1\ .$$

We will have more to say about representation types in Section 6.2, but for now we can just think of *List*° as a type that is more-or-less isomorphic to *List*, but one that uses only a small, fixed set of type constructors, namely sums, products, and unit, defined like this:

$$\begin{array}{lcl}\textbf{data}\ a \times b & = & a :\!*\!: b \\ \textbf{data}\ a + b & = & \mathit{Inl}\ a \mid \mathit{Inr}\ b \\ \textbf{data}\ 1 & = & \mathit{Unit}\ .\end{array}$$

Of course, infix (×) is not a legal Haskell type constructor, and nor are (+) and 1. We give them special syntax to distinguish them from their "normal" counterparts, (a, b), *Either a b* and (), and extend the syntax of types to accommodate them.

In our example, a *List* is a sum (+) of two types: the unit type (1), and a product (×) of the element type a and a *List*. To make the isomorphism explicit, let us write functions that convert to and fro:

$$\begin{array}{lcl}\mathit{to\text{-}List} & :: & \forall a\, .\, \mathit{List}^{\circ}\ a \rightarrow \mathit{List}\ a \\ \mathit{to\text{-}List}\ (\mathit{Inl}\ (x :\!*\!: \mathit{xs})) & = & \mathit{Cons}\ x\ \mathit{xs} \\ \mathit{to\text{-}List}\ (\mathit{Inr}\ \mathit{Unit}) & = & \mathit{Nil}\end{array}$$

$$\begin{aligned}
&\mathit{from\text{-}List} && :: && \forall a\,.\,\mathit{List}\ a \to \mathit{List}^{\circ}\ a\\
&\mathit{from\text{-}List}\ (\mathit{Cons}\ x\ xs) && = && \mathit{Inl}\ (x :\!*\!:\ xs)\\
&\mathit{from\text{-}List}\ \mathit{Nil} && = && \mathit{Inr}\ \mathit{Unit}\ .
\end{aligned}$$

3.4 The generic instances

The idea is that *by regarding a List as a List*$^{\circ}$*, the generic code explains what to do*. The generic method for *showBin*, for example, says what to do if the argument is a sum, what to do if it is a product, and what to do if it is a unit type.

It's useful to imagine re-expressing these default methods as three ordinary instance declarations:

```
instance Binary 1 where
  showBin Unit      =  []
  readBin bs        =  (Unit, bs)
instance (Binary a, Binary b) ⇒ Binary (a + b) where
  showBin (Inl x)   =  0 : showBin x
  showBin (Inr y)   =  1 : showBin y
  readBin []        =  error "readBin:␣unexpected␣end␣of␣input"
  readBin (0 : bs)  =  let (x, bs′) = readBin bs in (Inl x, bs′)
  readBin (1 : bs)  =  let (y, bs′) = readBin bs in (Inr y, bs′)
instance (Binary a, Binary b) ⇒ Binary (a × b) where
  showBin (x :*: y) =  showBin x ++ showBin y
  readBin bs        =  let (x, bs₁) = readBin bs
                           (y, bs₂) = readBin bs₁
                       in ((x :*: y), bs₂) .
```

We describe these instance declarations for generic representation types as *generic instance declarations*. They are not written explicitly by the programmer, but instead are derived by the compiler from a class declaration that has generic default methods. We discuss generic instance declarations further in Section 4.2.

3.5 Filling in the missing methods

We are now ready to say more precisely how the compiler fills in the missing methods. In this section we sketch the idea using an example, while Section 6 deals with the general case.

When the programmer writes

```
instance (Binary a) ⇒ Binary (List a)
```

the compiler will fill in the method declarations as follows:

```
showBin xs  =  showBin (from-List xs)
readBin bs  =  case readBin bs of (xs, bs′) → (to-List xs, bs′) .
```

Let us focus on the definition for *showBin*. It works in two stages:

1. First, $\mathit{from\text{-}List} :: \forall a\,.\, \mathit{List}\ a \to \mathit{List}^{\circ}\ a$ converts the input list of type *List a* into a value of type $\mathit{List}^{\circ}\ a$.
2. Second, we call the overloaded *showBin* function to complete the job, using the methods of the generic instance declarations.

At first this looks utterly bizarre. We are defining *showBin* in terms of *showBin*. But look at the definition one would write by hand:

```
instance (Binary a) ⇒ Binary (List a) where
  showBin Nil           =  0 : [ ]
  showBin (Cons x xs)   =  1 : showBin x ++ showBin xs  .
```

The first call is to *showBin* at the list element type; the second is a recursive call to the same *showBin* at the list type.

Something similar happens with the generic definition. Here *showBin* is called on an argument of type $\mathit{List}^{\circ}\ a$. This is a sum type, so the sum instance of *Binary* kicks in (Section 3.4). It in turn will call *showBin*, once at type 1, and once at type $a \times \mathit{List}\ a$. The former has an instance declaration, while the latter uses the product instance and makes calls to *showBin* at type *a* and *List a*. But the former is just like the *showBin x* in the hand-written instance, while the latter is like the *showBin xs*. So everything works out.

Let us return briefly to the first step above. In the case of *showBin* it was fairly simple to convert the argument to its generic representation type. On the other hand *readBin* was a bit more complicated because it returned a pair, only one component of which had to be converted. How, in general, does the compiler perform this conversion? We devote the whole of Section 5 to this topic. First, though, we elaborate on the programmer-visible aspects of our design.

4 Discussion and elaboration

We have now sketched the bones of our design. In this section we elaborate on some of the details.

4.1 Constructor names and record labels

Annoyingly, sum, product, and unit are not quite enough. Consider, for example, the standard Haskell class *Show*. To be able to give generic definitions for *showsPrec*, the *names* of the constructors, and their fixities, must be made accessible.

To this end we provide an additional generic representation type, of the form *c* **of** *a* where *c* is a *value* variable of type *ConDescr* and *a* is a *type* variable. The type

ConDescr is a new primitive type that comprises all constructor names. To manipulate constructor names the following operations among others can be used — for an exhaustive list see (Hinze, 1999).

```
data ConDescr       -- abstract
data Fixity     =   Nofix | Infix Int | Infixl Int | Infixr Int
conName         ::  ConDescr → String      -- primitive
conArity        ::  ConDescr → Int         -- primitive
conFixity       ::  ConDescr → Fixity      -- primitive
instance Show ConDescr where
    show        =   conName
```

Using *conName* and *conArity* we can implement a simple variant of Haskell's *Show* class — for the full-fledged version see (Hinze, 1999).

```
class Show t where
  show                          ::  t → String
  showsPrec                     ::  Int → t → String
  show x                        =   showsPrec 0 x
  showsPrec{ a + b }d (Inl x)   =   showsPrec d x
  showsPrec{ a + b }d (Inr y)   =   showsPrec d y
  showsPrec{ c of a }d (Con c x)
     | conArity c == 0          =   show c
     | otherwise                =   showParen (d ⩾ 10)
                                      (show c ++ "␣" ++ showsPrec 10 x)
  showsPrec{ a × b }d (x :*: y) =   showsPrec d x ++ "␣" ++ showsPrec d y
```

The representation type c **of** a is defined by the following pseudo-Haskell declaration:

```
newtype c of a   =   Con c a .
```

Uniquely for Haskell, c is a *value* that is carried by a *type*. It is best to think of the above declaration as defining a family of types: for each c there is a type constructor "c **of**" of kind $\star \to \star$ with a value constructor "*Con* c" of type $a \to (c$ **of** $a)$. Now, why does the type c **of** a incorporate information about c? One might suspect that it is sufficient to supply this information on the value level. Doing so would work for *show*, but would fail for *read*:

```
class Read t where
  read             ::  String → [(t, String)]
  read{ c of a }s  =   [(Con c x, s₃) | (s₁, s₂) ← lex s, s₁ == conName c,
                                        (x, s₃) ← read s₂] .
```

The important point is that *read produces* (not consumes) the value, and yet it requires access to the constructor name.

Haskell allows the programmer to assign labels to the components of a constructor, and these, too, are needed by *read* and *show*. For the purpose of presentation, however, we choose to ignore field names. In fact, they can be handled completely analogously to constructor names.

4.2 Generic instance declarations

In Section 3.4 we said that the generic definitions in a class declaration are re-expressed by the compiler as a set of instance declarations, one for each generic representation type. One might ask: why not get the programmer to write these instance declarations directly?

Our answer is stylistic rather than technical. We want to present generic programming in Haskell as a richer language in which to write default method declarations, and scattering them over several instance declarations does not convey that message. The question about whether a generic-default declaration was available to use would become more diffuse, because some parts, but not others, might be available. Writing the declaration all at once, in the class declaration, seems to be the simplest and most direct thing to do, even though it does involve a little new syntax.

Another stylistic reason for our decision is that it is rather easy to confuse the generic instance declaration for, say, products $a \times b$ with “ordinary” instance declarations for the corresponding “ordinary” product type (a, b). For example, in the case of *Show*, the ordinary instance declaration for products might look like this:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Show}\ a, \mathit{Show}\ b) \Rightarrow \mathit{Show}\ (a, b)\ \textbf{where}\\ \quad \mathit{showsPrec}\ d\ (x, y) \quad = \quad \texttt{"("} +\!\!+\ \mathit{showsPrec}\ 0\ x\ +\!\!+\ \texttt{","}\ +\!\!+\ \mathit{showsPrec}\ 0\ y\ +\!\!+\ \texttt{")"}.\end{array}$$

Because tuples are typically shown using distfix notation, we choose to over-ride the generic definition. Nevertheless, the class declaration for *Show* will have given rise to the generic instance declaration

$$\begin{array}{l}\textbf{instance}\ (\mathit{Show}\ a, \mathit{Show}\ b) \Rightarrow \mathit{Show}\ (a \times b)\ \textbf{where}\\ \quad \mathit{showsPrec}\ d\ (x :\!*\!: y) \quad = \quad \mathit{showsPrec}\ d\ x\ +\!\!+\ \texttt{"␣"}\ +\!\!+\ \mathit{showsPrec}\ d\ y\ \ .\end{array}$$

Recall that products $a \times b$ are used to represent the arguments of a constructor. Consequently, the generic instance declaration specifies the layout of constructor arguments: they are shown consecutively separated by spaces.

4.3 Generic default declarations

In general, a class declaration consists of a signature, which specifies the types of the class methods, and an implementation part, which gives default definitions for the class methods. The type signature has the general form:

$$\begin{array}{l}\textbf{class}\ \mathit{ctx} \Rightarrow C\ a\ \textbf{where}\\ \quad \begin{array}{lcl} op_1 & :: & Op_1\ a\\ \ldots & & \\ op_n & :: & Op_n\ a\ \ .\end{array}\end{array}$$

The implementation part consists of generic and non-generic default definitions. A non-generic definition is an ordinary Haskell definition

$$op \quad = \quad \ldots \ .$$

A generic definition can be recognised by the *type patterns* on the left hand side, enclosed in curly braces. It has the schematic form

$$\begin{aligned} op\{\,()\,\} &= \ldots \\ op\{\,a + b\,\} &= \ldots \\ op\{\,a \times b\,\} &= \ldots \\ op\{\,c \ \mathbf{of}\ a\,\} &= \ldots \ . \end{aligned}$$

The type patterns are mandatory, so that the equations can be correctly grouped. For example, consider the generic definition of (==) given earlier:

$$\begin{aligned} (==)\{\,1\,\}\mathit{Unit}\ \mathit{Unit} &= \mathit{True} \\ (==)\{\,a + b\,\}(\mathit{Inl}\ x_1)\ (\mathit{Inl}\ x_2) &= x_1 == x_2 \\ (==)\{\,a + b\,\}(\mathit{Inr}\ y_1)\ (\mathit{Inr}\ y_2) &= y_1 == y_2 \\ (==)\{\,a + b\,\}\_\ \_ &= \mathit{False} \\ (==)\{\,a \times b\,\}(x_1 :\!*\!: y_1)\ (x_2 :\!*\!: y_2) &= x_1 == x_2 \wedge y_1 == y_2 \ . \end{aligned}$$

Without the type patterns there is no way to decide whether the second but last equation belongs to the $(+)$ or to the $(\times)$ case.

Apart from the type patterns, an generic definition has exactly the form of a normal Haskell definition.

We note the following points:

- A class declaration may specify an arbitrary mixture of generic and non-generic default-method declarations. In the case of *Eq* above, (==) is defined by induction over the argument type, while (/=) is non-generic. The generic and non-generic methods may be mutually recursive.
- Class declarations are the *only* place that generic definitions appear in our design. There are no "free-standing" generic definitions.
- At the moment, generic default declarations may only be given for *type* classes, that is, for classes whose type parameter ranges over types of kind $\star$. For example, we cannot specify a generic default method for the *Functor* class:

  $$\begin{aligned} &\mathbf{class}\ \mathit{Functor}\ f\ \mathbf{where} \\ &\quad \mathit{fmap} \ :: \ (a \to b) \to (f\ a \to f\ b) \ . \end{aligned}$$

  This is the first extension we plan to add in the future.
- For a multi-parameter type class there would be multiple type arguments. We do not consider this complication in this paper.

4.4 Why our scheme is so beautiful

We have described a design for adding generic programming to Haskell. We are now in a position to briefly enumerate its merits:

- It fits with the "spirit of Haskell". At first sight, generic programming and Haskell type classes are in competition, but we use generic programming to smoothly extend the power of type classes.
- We are able to explain what "**deriving**" means in a systematic way. The *ad hoc* nature of **deriving** has long been considered a wart, and programmers often want to add new "derivable" classes — that is, classes for which you can say "**deriving** (C)". Now they can.
- Generic definitions can be over-ridden at particular types by programmer-supplied instance declarations. This sets our approach apart from other generic programming schemes. Not only is this useful for primitive types, but generic methods are often inapplicable for abstract types — consider equality on sets represented as unordered lists, for example.
- There is no run-time passing or case-analysis of types, beyond Haskell's existing dictionary passing. Of course, dictionary-passing *is* a sort of type passing, but it already exists in Haskell, and it would be extremely tiresome to introduce another, overlapping mechanism.
  Nor are there any new requirements to inspect the run-time representation of a value, a feature of some proposals. Our proposal is a 100% compile-time transformation.
- Like Haskell's type classes, static specialisation is possible to eliminate run-time overhead (see Section 6.6).
- Our scheme deals successfully with constructor names and labels. We have to admit, though, that this is one of the trickier corners of the design.
- We cunningly re-use Haskell's type-class mechanism to instantiate the generic methods for particular types, by expressing the generic methods as generic instance declarations (Section 4.2). This approach means that we do not need to explain or implement exactly how this code instantiation takes place (e.g. how much is done at compile time). Instead we just piggy-back on an existing piece of implementation technology. (This is really a point about the implementation, not about the design.)

5 Mapping functions

We now turn our attention to the implementation of our design. As a first step, we discuss so-called *bidirectional maps*. Recall from Section 3.5 our general plan for filling in the generic method of an instance declaration. Suppose we have the following class declaration:

```
class C a where
  op :: Op a .
```

We will deal only with single-parameter type classes, but see Section 10. We also assume, for notational clarity, that the type of method *op* is given simply by *Op a*. We can always introduce a type synonym to make this so[1]. Now suppose that the programmer writes the instance declaration

$$\textbf{instance}\ \mathit{ctx} \Rightarrow C\ (T\ \bar{a})\ .$$

where *ctx* is a context, and $\bar{a}$ is a sequence of type variables. How is the compiler to fill in the definition of method *op*? Following Section 3.5 it can fill in thus:

$$\begin{array}{l}\textbf{instance}\ \mathit{ctx} \Rightarrow C\ (T\ \bar{a})\ \textbf{where}\\ \quad \mathit{op} \quad = \quad \mathit{adapt\text{-}Op}\ (\mathit{op} :: \mathit{Op}\ (T^{\circ}\ \bar{a}))\ .\end{array}$$

That is, we call *op* at type $T^{\circ}\ \bar{a}$, to produce a value of type $\mathit{Op}\ (T^{\circ}\ \bar{a})$, and then convert the value to $\mathit{Op}\ (T\ \bar{a})$. The function *adapt-Op* does this impedance-matching by converting a function that works on values of type $T^{\circ}\ \bar{a}$ to one that works on $T\ \bar{a}$.

$$\mathit{adapt\text{-}Op} \quad :: \quad \forall \bar{a}\,.\,\mathit{Op}\ (T^{\circ}\ \bar{a}) \to \mathit{Op}\ (T\ \bar{a})$$

Clearly, how *adapt-Op* works depends on the form of *Op*, the type of the method. Here are some examples:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{In}\ a & = & a \to \mathit{String}\\
\mathit{adapt\text{-}In} & :: & \forall \bar{a}\,.\,\mathit{In}\ (T^{\circ}\ \bar{a}) \to \mathit{In}\ (T\ \bar{a})\\
\mathit{adapt\text{-}In} & = & \lambda f \to f \cdot \mathit{from\text{-}T}\\
\textbf{type}\ \mathit{Out}\ a & = & \mathit{String} \to a\\
\mathit{adapt\text{-}Out} & :: & \forall \bar{a}\,.\,\mathit{Out}\ (T^{\circ}\ \bar{a}) \to \mathit{Out}\ (T\ \bar{a})\\
\mathit{adapt\text{-}Out} & = & \lambda f \to \mathit{to\text{-}T} \cdot f\\
\textbf{type}\ \mathit{InOut}\ a & = & a \to a\\
\mathit{adapt\text{-}InOut} & :: & \forall \bar{a}\,.\,\mathit{InOut}\ (T^{\circ}\ \bar{a}) \to \mathit{InOut}\ (T\ \bar{a})\\
\mathit{adapt\text{-}InOut} & = & \lambda f \to \mathit{to\text{-}T} \cdot f \cdot \mathit{from\text{-}T}\ .
\end{array}$$

These *adapt* functions use the functions *to-T* and *from-T*, that convert between $T\ \bar{a}$ and $T^{\circ}\ \bar{a}$; they were introduced in Section 3.3. Notice that both *to-T* and *from-T* are needed; one by itself will not do. Furthermore, while we define the class, and hence the *Op* types, once, we may write instances of that class at many different types, *T*. So we want to abstract out the *to-T* and *from-T* functions from the *adapt* functions (note that a° is a type variable below):

$$\begin{array}{lcl}
\mathit{adapt\text{-}InOut}' & :: & \forall a\ a^{\circ}\,.\,(a^{\circ} \to a) \to (a \to a^{\circ}) \to (\mathit{InOut}\ a^{\circ} \to \mathit{InOut}\ a)\\
\mathit{adapt\text{-}InOut}'\ \mathit{to}\ \mathit{from} & = & \lambda f \to \mathit{to} \cdot f \cdot \mathit{from}\\
\mathit{adapt\text{-}InOut} & = & \mathit{adapt\text{-}InOut}'\ \mathit{to\text{-}T}\ \mathit{from\text{-}T}
\end{array}$$

[1] Technically, Haskell type synonyms are not powerful enough to do such an abbreviation for a method like *properFraction*:

$$\begin{array}{l}\textbf{class}\ \mathit{RealFrac}\ a\ \textbf{where}\\ \quad \mathit{properFraction} \quad :: \quad (\mathit{Integral}\ b) \Rightarrow a \to (b, a)\ .\end{array}$$

since its type has a context at the beginning. But we will pretend that such an abbreviation is possible.

It turns out to be convenient to package up the *to-T* and *from-T* functions into an *embedding-projection pair*:

$$\textbf{data}\ EP\ a\ a^{\circ} \quad = \quad EP\{\, to :: a^{\circ} \to a, from :: a \to a^{\circ}\,\}\ .$$

Here $EP\ a\ a^{\circ}$ is just a pair of functions, one to convert in one direction and one to convert back. Now we can write

$$\begin{array}{lcl} \textit{conv-T} & :: & \forall \bar{a}\,.\,EP\ (T\ \bar{a})\ (T^{\circ}\ \bar{a}) \\ \textit{conv-T} & = & EP\{\, to = \textit{to-T}, from = \textit{from-T}\,\} \\ \textit{adapt-InOut}'' & :: & \forall a\ a^{\circ}\,.\,EP\ a\ a^{\circ} \to InOut\ a^{\circ} \to InOut\ a \\ \textit{adapt-InOut}''\ \textit{ep-a} & = & \lambda f \to to\ \textit{ep-a} \cdot f \cdot from\ \textit{ep-a} \\ \textit{adapt-InOut} & = & \textit{adapt-InOut}''\ \textit{conv-T} \end{array}$$

The last step is to make the *adapt* function itself return an embedding-projection pair, rather than just the “to” function; and at this stage we adopt the name *bimap* — for “bidirectional mapping”:

$$\begin{array}{lcl} \textit{bimap-InOut} & :: & \forall a\ a^{\circ}\,.\,EP\ a\ a^{\circ} \to EP\ (InOut\ a)\ (InOut\ a^{\circ}) \\ \textit{bimap-InOut}\ \textit{ep-a} & = & EP\{\, to = \lambda f \to from\ \textit{ep-a} \cdot f \cdot to\ \textit{ep-a}, \\ & & \quad from = \lambda f \to to\ \textit{ep-a} \cdot f \cdot from\ \textit{ep-a}\,\} \\ \textit{adapt-InOut} & = & to\ (\textit{bimap-InOut}\ \textit{conv-T})\ . \end{array}$$

It is not at all obvious why we construct mappings in both directions, only to discard one of them when we use it, but it turns out to be essential when constructing *bimap* for arbitrary types, as we will see in the next section.

5.1 Generating bidirectional mapping functions

In the last section we generated *bimap-InOut* for a particular method type $InOut\ a$. We also observed that appropriate code depends on the structure of the type of the method, so the million-dollar question is: how do we generate the bidirectional maps for *arbitrary* method types? We do it simply by induction over the type of the method, thus:

$$\begin{array}{lcl} \textit{bimap-Op} & :: & \forall a\ a^{\circ}\,.\,EP\ a\ a^{\circ} \to EP\ (Op\ a)\ (Op\ a^{\circ}) \\ \textit{bimap-Op}\ \textit{ep-a} & = & bimap\{\, Op\ a\,\}[\, a := \textit{ep-a}\,]\ . \end{array}$$

This definition is not proper Haskell; *bimap* should be thought of as a meta-function, evaluated at compile time, that returns a Haskell expression. It takes as arguments: a type (written in curly braces), and an environment ϱ mapping type variables to expressions. The syntax $[\, a := \textit{ep-a}\,]$ means an environment that binds a to *ep-a*.

We define *bimap* by induction on the structure of type expressions:

$$\begin{array}{lcl} bimap\{\, a\,\}\varrho & = & \varrho(a) \\ bimap\{\, (\to)\,\}\varrho & = & \textit{bimap-Arrow} \\ bimap\{\, T\,\}\varrho & = & \textit{bimap-T} \\ bimap\{\, t\ u\,\}\varrho & = & (bimap\{\, t\,\}\varrho)\ (bimap\{\, u\,\}\varrho) \end{array}$$

where

$$\begin{aligned}
&\mathit{bimap\text{-}Arrow} \;::\; \forall a\ a^\circ\ b\ b^\circ\,.\,\mathit{EP}\ a\ a^\circ \to \mathit{EP}\ b\ b^\circ \to \mathit{EP}\ (a \to b)\ (a^\circ \to b^\circ)\\
&\mathit{bimap\text{-}Arrow}\ \mathit{ep\text{-}a}\ \mathit{ep\text{-}b}\\
&\qquad = \mathit{EP}\{\mathit{to} = \lambda f \to \mathit{to}\ \mathit{ep\text{-}b} \cdot f \cdot \mathit{from}\ \mathit{ep\text{-}a},\\
&\qquad\qquad \mathit{from} = \lambda f \to \mathit{from}\ \mathit{ep\text{-}b} \cdot f \cdot \mathit{to}\ \mathit{ep\text{-}a}\}\ .
\end{aligned}$$

The $\mathit{bimap}\{T\}$ case deals with type constructors other than $(\to)$, which we discuss in Section 5.2. Let us take an example. Recall our *InOut* type:

$$\textbf{type}\ \mathit{InOut}\ a \;=\; a \to a\ .$$

Setting $\varrho = [a := \mathit{ep\text{-}a}]$ we have

$$\begin{aligned}
\mathit{bimap\text{-}InOut}\ \mathit{ep\text{-}a} &= \mathit{bimap}\{a \to a\}\varrho\\
&= (\mathit{bimap}\{(\to)\ a\}\varrho)\ (\mathit{bimap}\{a\}\varrho)\\
&= ((\mathit{bimap}\ (\to)\ \varrho)\ (\mathit{bimap}\{a\}\varrho))\ (\mathit{bimap}\{a\}\varrho)\\
&= \mathit{bimap\text{-}Arrow}\ \mathit{ep\text{-}a}\ \mathit{ep\text{-}a}\\
&= \mathit{EP}\{\mathit{to} = \lambda f \to \mathit{from}\ \mathit{ep\text{-}a} \cdot f \cdot \mathit{to}\ \mathit{ep\text{-}a},\\
&\qquad \mathit{from} = \lambda f \to \mathit{to}\ \mathit{ep\text{-}a} \cdot f \cdot \mathit{from}\ \mathit{ep\text{-}a}\}\ .
\end{aligned}$$

5.2 Mapping over data types

What if there is a data type involved? For example in the type of *readBin*, there is a pair in the result type:

$$\textbf{type}\ \mathit{ReadBin}\ a \;=\; \mathit{Bin} \to (a, \mathit{Bin})\ .$$

If we just try our current scheme we get stuck:

$$\begin{aligned}
\mathit{bimap\text{-}ReadBin}\ \mathit{ep} &= \mathit{bimap}\{\mathit{Bin} \to (a, \mathit{Bin})\}\varrho\\
&= \mathit{bimap\text{-}Arrow}\ (\mathit{bimap}\{\mathit{Bin}\}\varrho)\ (\mathit{bimap}\{(a, \mathit{Bin})\}\varrho)\ .
\end{aligned}$$

Since *Bin* is not a parameterised type, there is nothing to do,

$$\begin{aligned}
\mathit{bimap\text{-}Bin} &:: \mathit{EP}\ \mathit{Bin}\ \mathit{Bin}\\
\mathit{bimap\text{-}Bin} &= \mathit{id\text{-}EP}\\
\mathit{id\text{-}EP} &:: \forall a\,.\,\mathit{EP}\ a\ a\\
\mathit{id\text{-}EP} &= \mathit{EP}\{\mathit{to} = \lambda x \to x, \mathit{from} = \lambda x \to x\}
\end{aligned}$$

whereas pairs are parameterised over two types, so we must push the mapping functions inside:

$$\begin{aligned}
&\mathit{bimap\text{-}Pair} \;::\; \forall a\ a^\circ\ b\ b^\circ\,.\,\mathit{EP}\ a\ a^\circ \to \mathit{EP}\ b\ b^\circ \to \mathit{EP}\ (a, b)\ (a^\circ, b^\circ)\\
&\mathit{bimap\text{-}Pair}\ \mathit{ep\text{-}a}\ \mathit{ep\text{-}b}\\
&\qquad = \mathit{EP}\{\mathit{to} = \lambda(x^\circ, y^\circ) \to (\mathit{to}\ \mathit{ep\text{-}a}\ x^\circ, \mathit{to}\ \mathit{ep\text{-}b}\ y^\circ),\\
&\qquad\qquad \mathit{from} = \lambda(x, y) \to (\mathit{from}\ \mathit{ep\text{-}a}\ x, \mathit{from}\ \mathit{ep\text{-}b}\ y)\}\ .
\end{aligned}$$

In general, we can define *bimap-T* by induction on the structure of datatype declarations. The mapping function for the data type

$$\textbf{data}\ T\ a_1\ \ldots\ a_k \quad = \quad K_1\ t_{11}\ \ldots\ t_{1m_1} \mid \cdots \mid K_n\ t_{n1}\ \ldots\ t_{nm_n}$$

is given by

$$\begin{array}{lcl}
\mathit{bimap\text{-}T}\ \mathit{bimap\text{-}a}_1\ \ldots\ \mathit{bimap\text{-}a}_k & = & \mathit{EP}\{\mathit{to} = \mathit{to\text{-}T}, \mathit{from} = \mathit{from\text{-}T}\} \\
\quad \textbf{where} & & \\
\quad \mathit{to\text{-}T}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) & = & K_1\ (\mathit{to}\{t_{11}\}x_{11})\ \ldots\ (\mathit{to}\{t_{1m_1}\}x_{1m_1}) \\
\quad \ldots & & \\
\quad \mathit{to\text{-}T}\ (K_n\ x_{n1}\ \ldots\ x_{nm_n}) & = & K_n\ (\mathit{to}\{t_{n1}\}x_{n1})\ \ldots\ (\mathit{to}\{t_{1m_1}\}x_{nm_n}) \\
\quad \mathit{from\text{-}T}\ (K_1\ x_{11}\ \ldots\ x_{1m_1}) & = & K_1\ (\mathit{from}\{t_{11}\}x_{11})\ \ldots\ (\mathit{from}\{t_{1m_1}\}x_{1m_1}) \\
\quad \ldots & & \\
\quad \mathit{from\text{-}T}\ (K_n\ x_{n1}\ \ldots\ x_{nm_n}) & = & K_n\ (\mathit{from}\{t_{n1}\}x_{n1})\ \ldots\ (\mathit{from}\{t_{1m_1}\}x_{nm_n})
\end{array}$$

where

$$\begin{array}{lcl}
\mathit{to}\{t\} & = & \mathit{to}\ (\mathit{bimap}\{t\}[a_1 := \mathit{bimap\text{-}a}_1, \ldots, a_k := \mathit{bimap\text{-}a}_k]) \\
\mathit{from}\{t\} & = & \mathit{from}\ (\mathit{bimap}\{t\}[a_1 := \mathit{bimap\text{-}a}_1, \ldots, a_k := \mathit{bimap\text{-}a}_k])\ .
\end{array}$$

Now, what is type of this bidirectional map? The answer is simple yet mind-boggling: the type of *bimap-T* depends on the kind of T. Assume that T has kind $\star$ as, for instance, *Bin*. Then the bidirectional map simply has type $\mathit{EP}\ T\ T$ (and, in fact, $\mathit{bimap\text{-}T} = \mathit{id\text{-}EP}$). If T has kind $\star \to \star$ as all the *Op*'s have, then *bimap-T*'s type is close to that of an "ordinary" mapping function:

$$\mathit{bimap\text{-}T} \quad :: \quad \forall a\ a^\circ . \mathit{EP}\ a\ a^\circ \to \mathit{EP}\ (T\ a)\ (T\ a^\circ)\ .$$

A more involved kind, say $(\star \to \star) \to (\star \to \star)$, gives rise to a more complicated type:

$$\begin{array}{lcl}
\mathit{bimap\text{-}T} & :: & \forall f\ f^\circ . (\forall b\ b^\circ . \mathit{EP}\ b\ b^\circ \to \mathit{EP}\ (f\ b)\ (f^\circ\ b^\circ)) \\
& & \qquad \to (\forall a\ a^\circ . \mathit{EP}\ a\ a^\circ \to \mathit{EP}\ (T\ f\ a)\ (T\ f^\circ\ a^\circ))\ .
\end{array}$$

Now *bimap-T* has a so-called rank-2 type signature (Leivant, 1983). Roughly speaking, *bimap-T* takes bidirectional maps to bidirectional maps (this is why the arguments of *bimap-T* are called $\mathit{bimap\text{-}a}_i$). In general, if T has kind κ, then *bimap-T* has type $\mathit{Bimap}\{\kappa\}T\ T$ where *Bimap* is defined by induction on the structure of kinds:

$$\begin{array}{lcl}
\mathit{Bimap}\{\star\}t\ t^\circ & = & \mathit{EP}\ t\ t^\circ \\
\mathit{Bimap}\{\kappa_1 \to \kappa_2\}t\ t^\circ & = & \forall a\ a^\circ . \mathit{Bimap}\{\kappa_1\}a\ a^\circ \to \mathit{Bimap}\{\kappa_2\}(t\ a)\ (t^\circ\ a^\circ)\ .
\end{array}$$

We will say a bit more about higher-order kinded types in Section 7. For further information on kind-indexed types such as *Bimap* the reader if referred to (Hinze, 2000c).

6 Implementing generic default methods

In this section we discuss the implementation of our design.

We describe our implementation as a Haskell source-to-source translation, performed (at least notionally) prior to type checking. Why? The type checker already does a lot of what we require. Also we probably have a better chance that generic default methods will work smoothly with complications such as multi-parameter type classes (Peyton Jones *et al.*, 1997), implicit parameters (Lewis *et al.*, 2000), and functional dependencies (Jones, 2000).

The source-to-source translation goes as follows. For each *datatype declaration*, T, we generate the following:

- For each constructor K a value $\mathit{con\text{-}K}$ of type $\mathit{ConDescr}$ that describes the properties of the constructor (Section 6.1).
- A type synonym, T°, for T's generic representation type (Section 6.2).
- An embedding-projection pair $\mathit{conv\text{-}T} :: \forall \bar{a} \,.\, \mathit{EP}\ (T\ \bar{a})\ (T^{\circ}\ \bar{a})$, that converts between T and its generic representation T° (Section 6.3).

For each *class declaration*, for class C, we generate the following (see Section 6.4):

- An emaciated class declaration for C, generated simply by omitting the generically-defined methods.
- For each generic method $\mathit{op} :: \mathit{Op}\ a$ in the class declaration, a bidirectional map $\mathit{bimap\text{-}Op} :: \forall a\ a^{\circ} \,.\, \mathit{EP}\ a\ a^{\circ} \to \mathit{EP}\ (\mathit{Op}\ a)\ (\mathit{Op}\ a^{\circ})$ (see Section 5).
- Instance declarations for $C\ 1$, $C\ (a{+}b)$, $C\ (a{\times}b)$ and $C\ (c\ \textbf{of}\ a)$, all obtained by selecting the eponymous equations from the original class declaration (see Section 3.4).

For each *instance declaration* we generate (see Section 6.5):

- An extended instance declaration, obtained by adding definitions for the generic methods that are not specified explicitly in the instance declaration.

6.1 Constructors

For each constructor, K, in a datatype declaration, we produce a value of type *ConDescr* that gives information about the constructor (in fact, the type *ConDescr* used in the compiler is slightly more elaborate):

$$\textbf{data}\ \mathit{ConDescr} \quad = \quad \mathit{ConDescr}\{\mathit{name} :: \mathit{String}, \mathit{arity} :: \mathit{Int}, \mathit{fixity} :: \mathit{Fixity}\}\ .$$

As an example, for the *List* datatype we generate:

$$\begin{aligned}
&\mathit{con\text{-}Cons}, \mathit{con\text{-}Nil} && :: && \mathit{ConDescr}\\
&\mathit{con\text{-}Cons} && = && \mathit{ConDescr}\ \texttt{"Cons"}\ 2\ \mathit{NoFixity}\\
&\mathit{con\text{-}Nil} && = && \mathit{ConDescr}\ \texttt{"Nil"}\ 0\ \mathit{NoFixity}\ .
\end{aligned}$$

6.2 Generic representation types

For each data type, T, we produce a type synonym T°, for its *generic representation type.* For example, for the data type

$$\textbf{data}\ \mathit{List}\ a \quad = \quad \mathit{Cons}\ a\ (\mathit{List}\ a) \mid \mathit{Nil}$$

we generate the representation type

$$\textbf{type}\ \mathit{List}^\circ\ a \quad = \quad (\mathit{con\text{-}Cons}\ \textbf{of}\ (a \times \mathit{List}\ a)) + (\mathit{con\text{-}Nil}\ \textbf{of}\ 1)$$

Our generic representation type constructors are just sum, product, unit and "c **of**". In particular, there is no recursion operator. Thus, we observe that List° is just a non-recursive type synonym: *List* (not List°) appears on the right-hand side. So List° is not a recursive type; rather, it expresses just the top "layer" of a list structure, leaving the original *List* to do the rest. But as we have seen, this is enough: a recursive function just does one "layer" of recursion at a time.

This is unusual compared to other approaches. In PolyP (Jansson & Jeuring, 1997), for instance, there is an additional type pattern for type recursion (at kind $\star \to \star$). A very significant advantage here is that there is no problem with mutually-recursive data types, nor with data types with many parameters, both of which make explicit recursion operators extremely clumsy and hard to use in practice.

Our design makes do with just *binary* sum and product. Algebraic data types with many constructors, each of which has many fields, are encoded as nested uses of sum and product. The exact way in which the nesting is done is unimportant to our method. For example:

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Color} & = & \mathit{Red} \mid \mathit{Blue} \mid \mathit{Green} \\
\textbf{type}\ \mathit{Color}^\circ & = & \mathit{con\text{-}Red}\ \textbf{of}\ 1 + (\mathit{con\text{-}Blue}\ \textbf{of}\ 1 + \mathit{con\text{-}Green}\ \textbf{of}\ 1) \\
\textbf{data}\ \mathit{Tree}\ a\ b & = & \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{Tree}\ a\ b)\ b\ (\mathit{Tree}\ a\ b) \\
\textbf{type}\ \mathit{Tree}^\circ\ a\ b & = & \mathit{con\text{-}Leaf}\ \textbf{of}\ a + (\mathit{con\text{-}Node}\ \textbf{of}\ (\mathit{Tree}\ a\ b \times (b \times \mathit{Tree}\ a\ b)))
\end{array}$$

One may wonder about the efficiency of translating a user-defined data type into a generic form before operating on it, especially if everything is encoded with only binary sums and products. However, sufficiently vigorous inlining means that the generic data representations never exist at run-time (see Section 6.6). But, in fact, we might want to explore space-time trade-offs, by getting much more compact code in exchange for some data translation. Our design allows this trade-off to be made on a case-by-case basis.

Whether the encoding into sums and products is done in a linear or binary-sub-division fashion may or may not affect efficiency, depending on how vigorous the inlining is.

6.3 Embedding-projection pairs

For each datatype T, we also generate functions to convert between T and T°. We saw the conversion functions for *List* in Section 3.3. The process is entirely straightforward, driven by the encoding. For example:

$$\begin{array}{lcl}
\mathit{from\text{-}Color} & :: & \mathit{Color} \to \mathit{Color}^\circ \\
\mathit{from\text{-}Color}\ \mathit{Red} & = & \mathit{Inl}\ (\mathit{Con}\ \mathit{con\text{-}Red}\ \mathit{Unit}) \\
\mathit{from\text{-}Color}\ \mathit{Blue} & = & \mathit{Inr}\ (\mathit{Inl}\ (\mathit{Con}\ \mathit{con\text{-}Blue}\ \mathit{Unit})) \\
\mathit{from\text{-}Color}\ \mathit{Green} & = & \mathit{Inr}\ (\mathit{Inr}\ (\mathit{Con}\ \mathit{con\text{-}Green}\ \mathit{Unit}))
\end{array}$$

$$\begin{array}{lcl}
\mathit{to\text{-}Color} & :: & \mathit{Color}^\circ \to \mathit{Color} \\
\mathit{to\text{-}Color}\ (\mathit{Inl}\ (\mathit{Con}\ \mathit{con\text{-}Red}\ \mathit{Unit})) & = & \mathit{Red} \\
\mathit{to\text{-}Color}\ (\mathit{Inr}\ (\mathit{Inl}\ (\mathit{Con}\ \mathit{con\text{-}Blue}\ \mathit{Unit}))) & = & \mathit{Blue} \\
\mathit{to\text{-}Color}\ (\mathit{Inr}\ (\mathit{Inr}\ (\mathit{Con}\ \mathit{con\text{-}Green}\ \mathit{Unit}))) & = & \mathit{Green}\ .
\end{array}$$

For *bimap* we have to package the two conversion functions into a single value:

$$\begin{array}{lcl}
\mathit{conv\text{-}List} & :: & \forall a\,.\,\mathit{EP}\ (\mathit{List}\ a)\ (\mathit{List}^\circ\ a) \\
\mathit{conv\text{-}List} & = & \mathit{EP}\{\mathit{to} = \mathit{to\text{-}List}, \mathit{from} = \mathit{from\text{-}List}\} \\
\mathit{conv\text{-}Color} & :: & \mathit{EP}\ \mathit{Color}\ \mathit{Color}^\circ \\
\mathit{conv\text{-}Color} & = & \mathit{EP}\{\mathit{to} = \mathit{to\text{-}Color}, \mathit{from} = \mathit{from\text{-}Color}\}\ .
\end{array}$$

6.4 Translating class declarations

For each generic method $op :: Op\ a$ contained in a class declaration we generate a bidirectional map

$$\mathit{bimap\text{-}Op}\ \ ::\ \ \forall a\ a^\circ\,.\,\mathit{EP}\ a\ a^\circ \to \mathit{EP}\ (\mathit{Op}\ a)\ (\mathit{Op}\ a^\circ)$$

that allows us to convert between types and representation types (the definition of *bimap-Op* is given in Section 5).

Furthermore, we produce instance declarations

$$\begin{array}{l}
\textbf{instance}\ C\ 1 \\
\textbf{instance}\ (C\ a, C\ b) \Rightarrow C\ (a + b) \\
\textbf{instance}\ (C\ a, C\ b) \Rightarrow C\ (a \times b) \\
\textbf{instance}\ (C\ a) \Rightarrow C\ (c\ \textbf{of}\ a)
\end{array}$$

whose bodies are filled with the generic methods from the original class declaration (see Section 3.4). If an equation for a type pattern is missing, the method of the corresponding instance is undefined. There is, however, one important exception to this rule: if no equation is given for the type pattern c **of** a as, for example, in the classes *Eq* and *Binary*, we define the generic methods of the $C\ (c\ \textbf{of}\ a)$ instance by:

$$\mathit{op}\{c\ \textbf{of}\ a\}\ \ =\ \ \mathit{to}\ (\mathit{bimap\text{-}Op}\ (\mathit{con\text{-}EP}\ c))\ (\mathit{op} :: \mathit{Op}\ a)$$

where *con-EP* is given by the following pseudo-Haskell code (which defines a family of functions):

$$\begin{array}{lcl} \mathit{con\text{-}EP}\ c & :: & \forall a\,.\,\mathit{EP}\ a\ (c\ \mathbf{of}\ a) \\ \mathit{con\text{-}EP}\ c & = & \mathit{EP}\{\mathit{to} = \lambda x \to \mathit{Con}\ c\ x, \mathit{from} = \lambda(\mathit{Con}\ c\ x) \to x\}\ . \end{array}$$

Again, we employ the bidirectional map to convert between two isomorphic types.

6.5 Translating instance declarations

An instance declaration for type T is extended by filling in implementations for the methods. More specifically, if the method *op* is not specified and if it enjoys a generic default definition, then the following equation is supplemented:

$$\mathit{op} \quad = \quad \mathit{to}\ (\mathit{bimap\text{-}Op}\ \mathit{conv\text{-}T})\ (\mathit{op} :: \mathit{Op}\ T^{\circ})\ .$$

That's it.

6.6 Inlining

It does not sound very efficient to translate a value from $T\ \bar{a}$ to $T^{\circ}\ \bar{a}$ and then to operate on it, but we believe that a bit of judicious inlining can yield more or less the code one would have written by hand. Let us consider, for example, *showBin* at type *List*. The *showBinList* method will look something like this:

$$\begin{array}{lcl} \mathit{showBinList} & :: & (\mathit{Binary}\ a) \Rightarrow \mathit{List}\ a \to \mathit{Bin} \\ \mathit{showBinList}\ \mathit{xs} & = & \mathit{showBin}\ (\mathit{from\text{-}List}\ \mathit{xs}) \\ \mathbf{type}\ \mathit{List}^{\circ}\ a & = & (a \times \mathit{List}\ a) + 1 \\ \mathit{from\text{-}List} & :: & \mathit{List}\ a \to \mathit{List}^{\circ}\ a\ . \end{array}$$

The call to *showBin* is at type $\mathit{List}^{\circ}\ a$, so the overloading can be resolved statically. Assuming that the method bodies (given in Section 3.1) are inlined, we get:

$$\begin{array}{lcl} \mathit{showBinList}\ \mathit{xs} & = & \mathbf{case}\ \mathit{from\text{-}List}\ \mathit{xs}\ \mathbf{of} \\ & & \quad \mathit{Inl}\ z \to \mathsf{0} : \mathbf{case}\ z\ \mathbf{of} \\ & & \qquad\qquad (x :\!*\!: y) \to \mathit{showBin}\ x +\!\!+\ \mathit{showBin}\ y \\ & & \quad \mathit{Inr}\ z \to \mathsf{1} : \mathbf{case}\ z\ \mathbf{of}\ \mathit{Unit} \to [\,]\ . \end{array}$$

But remember that *from-List* also has a simple, non-recursive definition:

$$\begin{array}{lcl} \mathit{from\text{-}List}\ (\mathit{Cons}\ x\ \mathit{xs}) & = & \mathit{Inl}\ (x :\!*\!: \mathit{xs}) \\ \mathit{from\text{-}List}\ \mathit{Nil} & = & \mathit{Inr}\ \mathit{Unit}\ . \end{array}$$

If we inline this definition in *showBinList* and simplify using standard transformations, we get

$$\begin{array}{lcl} \mathit{showBinList}\ \mathit{xs} & = & \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\ & & \quad \mathit{Cons}\ x\ y \to \mathsf{0} : \mathit{showBin}\ x +\!\!+\ \mathit{showBin}\ y \\ & & \quad \mathit{Nil} \to \mathsf{1} : [\,] \end{array}$$

which is about as good as we can hope for.

7 Higher-order kinded types

Functional programmers love abstraction. In Haskell we can, for instance, abstract over the *List* datatype in

$$\textbf{data}\ \mathit{Rose}\ a \quad = \quad \mathit{Branch}\ a\ (\mathit{List}\ (\mathit{Rose}\ a))$$

obtaining the more general type:

$$\textbf{data}\ \mathit{GRose}\ f\ a \quad = \quad \mathit{GBranch}\ a\ (f\ (\mathit{GRose}\ f\ a))\ .$$

Here, the type variable "*f*" ranges over type constructors, rather than types. Formally, *GRose* has kind $(\star \to \star) \to \star \to \star$. There are numerous examples of such type definitions in (Okasaki, 1998; Hinze, 2000a). Alas, it is impossible to define many instance declarations for *GRose* in Haskell at all. In this section we describe the problem and a solution. This section is quite independent of the rest of the paper. Though we became aware of the issue when working on generic programming, we propose an extension to Haskell that is completely orthogonal to generic programming.

7.1 What's the problem?

Consider first defining an instance for *Binary* (*Rose a*) by hand — we ignore *readBin* here:

$$\begin{array}{l}\textbf{instance}\ (\mathit{Binary}\ a) \Rightarrow \mathit{Binary}\ (\mathit{Rose}\ a)\ \textbf{where}\\ \quad \mathit{showBin}\ (\mathit{Branch}\ x\ ts) \quad = \quad \mathit{showBin}\ x +\!\!+ \mathit{showBin}\ ts\ .\end{array}$$

The first call to *showBin* on the right hand side requires that *Binary a* should hold; the context, (*Binary a*), takes care of that. The second call is at type *List* (*Rose a*). Assuming we have an instance elsewhere of the form

$$\textbf{instance}\ (\mathit{Binary}\ t) \Rightarrow \mathit{Binary}\ (\mathit{List}\ t)$$

the second call requires *Binary* (*Rose a*), and there is an instance declaration for that too — it gives rise to a recursive call to *showBin*.

But matters are not so simple when we want to write an instance *Binary* (*GRose f a*). We might try

$$\begin{array}{l}\textbf{instance}\ (???) \Rightarrow \mathit{Binary}\ (\mathit{GRose}\ f\ a)\ \textbf{where}\\ \quad \mathit{showBin}\ (\mathit{GBranch}\ x\ ts) \quad = \quad \mathit{showBin}\ x +\!\!+ \mathit{showBin}\ ts\ .\end{array}$$

The context (???) must account for the calls to *showBin* on the right-hand side. The first one is fine: it requires *Binary a* as before. But the latter is bad news: it requires *Binary* (*f* (*GRose f a*)), and we certainly cannot write

$$\begin{array}{l}\textbf{instance}\ (\mathit{Binary}\ a, \mathit{Binary}\ (f\ (\mathit{GRose}\ f\ a))) \Rightarrow \mathit{Binary}\ (\mathit{GRose}\ f\ a)\ \textbf{where}\\ \quad \mathit{showBin}\ (\mathit{GBranch}\ x\ ts) \quad = \quad \mathit{showBin}\ x +\!\!+ \mathit{showBin}\ ts\ .\end{array}$$

This is not legal Haskell and, even if it were, the typechecker would diverge. Indeed, no ordinary Haskell context will do.

7.2 A solution

What we need is a way to simplify the predicate $f\ (GRose\ f\ a)$. The trick is to take the "constant" instance declaration that we assumed for $Binary\ (List\ a)$ above, and abstract over it:

instance $(Binary\ a, \forall b\,.\,(Binary\ b) \Rightarrow Binary\ (f\ b)) \Rightarrow Binary\ (GRose\ f\ a)$ **where**
  $showBin\ (GBranch\ x\ ts) \quad = \quad showBin\ x + \!\!+\ showBin\ ts$.

Now, as well as $(Binary\ a)$, the context also contains a *polymorphic predicate*. This predicate can be used to reduce the predicate $Binary\ (f\ (GRose\ f\ a))$ to just $Binary\ (GRose\ f\ a)$, and we have an instance declaration for that.

Viewed in operational terms, the predicate $(Binary\ a)$ in a context corresponds to passing a *dictionary* for class *Binary*. A predicate $\forall b\,.\,Binary\ b \Rightarrow Binary\ (f\ b)$ corresponds to passing a *dictionary transformer* to the function.

7.3 Deriving instance declarations

Of course, since *Binary* is a derivable class by virtue of the generic default definitions, we need not define *showBin* at all. We can simply write

  instance $(Binary\ a, \forall b\,.\,(Binary\ b) \Rightarrow Binary\ (f\ b)) \Rightarrow Binary\ (GRose\ f\ a)$

and get just the same effect as before. In other words, the deriving mechanism works happily for types of arbitrary kinds.

Can we use a **deriving** clause to get the same effect? Unfortunately, the answer is in the negative. The problem is that the above instance declaration is not the most general one and, in fact, there is no "most general instance declaration". To illustrate the point consider the following instance declaration for the abstract type *Set*:

instance $(Binary\ a, Ord\ a) \Rightarrow Binary\ (Set\ a)$.

Note that we additionally require that a is an instance of *Ord*. Now, given the instance declaration for *GRose* above, we cannot infer $Binary\ (GRose\ Set\ Int)$ since *Set* does not satisfy $\forall b\,.\,(Binary\ b) \Rightarrow Binary\ (Set\ b)$. If we require such an instance, we must *generalize* the *GRose* instance:

instance $(Binary\ a, \forall b\,.\,(Binary\ b, Ord\ b) \Rightarrow Binary\ (f\ b)) \Rightarrow Binary\ (GRose\ f\ a)$.

By adding further class constraints to f's context, we can generalize the instance declaration even more. Sadly, this implies that there is no "most general" instance which **deriving** could infer. Note that this problem does not crop up for first-order kinded types.

7.4 Formalising the extension

Here is the grammar for generalized instance declarations:

$$\begin{array}{llll} \text{instance head} & & ::= & \textbf{instance}\ (ctx_1, \ldots, ctx_n) \Rightarrow C\ t \\ \text{context} & ctx & ::= & \forall \bar{a}\,.\,(ctx_1, \ldots, ctx_n) \Rightarrow C\ t\ . \end{array}$$

A context of the form $\forall \bar{a}\,.\,(ctx_1, \ldots, ctx_n) \Rightarrow C\ t$ with $n \geqslant 1$ is called a *polymorphic predicate*. Note that for $n = 0$ we have "ordinary" Haskell 98 predicates.

8 Implementing generalized instance declarations

How do we translate a method call $op :: Op\ T$? We must create a C-dictionary for T if op is a method of class C. In the higher-order kinded situation, we may need to create a dictionary *transformer* to pass to op. Fortunately, it turns out that the now-standard machinery to construct the correct dictionary to pass can easily be extended to construct dictionary transformers too.

At a call site we have to solve the following problem: we have a set of *assumptions* $\mathbb{H}$ and a single clause H, the dictionary (transformer) required, and we want to know whether H is a logical consequence of $\mathbb{H}$. Additionally we return an expression for the dictionary (transformer) for H. We use the following notation: $\mathbb{H} \vdash H \longmapsto d$ means that d is a dictionary (transformer) expression that shows how H can be deduced from $\mathbb{H}$.

The assumptions $\mathbb{H}$ embody:

- Any instance declarations in scope. For example:

$$\begin{array}{c} Eq\ Int \longmapsto dict\text{-}Eq\text{-}Int \\ \forall a\,.\,Eq\ a \Rightarrow Eq\ (List\ a) \longmapsto dict\text{-}Eq\text{-}List \quad . \end{array}$$

- Information about superclasses. For example:

$$\forall a\,.\,Ord\ a \Rightarrow Eq\ a \longmapsto dict\text{-}Eq\text{-}Ord\ \ .$$

  This says that if we have $Ord\ a$ we can deduce $Eq\ a$; in concrete terms we witness this fact by the selector function $dict\text{-}Eq\text{-}Ord$ which selects the Eq dictionary from the Ord one.
- Constraints from the type signature. For example, if we are checking types for the function

$$\begin{array}{lcl} f & :: & \bar{H} \Rightarrow T \\ f\ x & = & \ldots \end{array}$$

  then we put the assumptions $\bar{H}$ in our assumption set, and try to deduce all the dictionaries that are needed by calls in the body of f.

We use the following inference rules:

$$\frac{(H \longmapsto d) \in \mathbb{H}}{\mathbb{H} \vdash H \longmapsto d} \qquad (\textsc{Assumption})$$

$$\frac{\mathbb{H} \vdash H_1 \longmapsto d_1 \qquad \cdots \qquad \mathbb{H} \vdash H_n \longmapsto d_n}{\mathbb{H} \vdash (H_1, \ldots, H_n) \longmapsto (d_1, \ldots, d_n)} \quad (\textsc{Conjunction})$$

$$\frac{\mathbb{H} \vdash (\forall \bar{a} \,.\, \bar{H} \Rightarrow Q) \longmapsto f \qquad \mathbb{H} \vdash \bar{H}\varrho\theta \longmapsto d}{\mathbb{H} \vdash P \longmapsto (f \;\; d)} \quad (\textsc{Modus Ponens})$$

where $\varrho = [\bar{a} := \bar{x}]$ is a renaming substitution (the x_i are fresh variables) and $\theta = \mathit{match}(Q\varrho, P)$ is the result of matching $Q\varrho$ against P (note that only the variables in $Q\varrho$ are bound).

So far, these rules are entirely standard, see, for instance, (Jones, 1994). To these we add one new rule.

$$\frac{\mathbb{H}, (\bar{H}\,\varrho \longmapsto v) \vdash Q\,\varrho \longmapsto d}{\mathbb{H} \vdash (\forall \bar{a} \,.\, \bar{H} \Rightarrow Q) \longmapsto (\lambda v \to d)} \quad (\textsc{Deduction Rule})$$

where $\varrho = [\bar{a} := \bar{c}]$ is a Skolem substitution, that is, the c_i are Skolem constants. Thus, to deduce the polymorphic predicate $\forall \bar{a} \,.\, \bar{H} \Rightarrow Q$ we add the body $\bar{H}$ to the set of assumptions and try to deduce Q. The Skolem substitution ensures that this derivation works for all $\bar{a}$.

The new rule is called Deduction Rule because it resembles the deduction theorem of first-order logic. It is also reminiscent of the usual typing rule for λ-abstraction while Modus Ponens corresponds to the typing rule for application. In fact, these two rules capture dictionary abstraction and dictionary application.

Here is an example of a deduction using these rules. Later lines are deduced from earlier ones using the specified rule (A for Assumption, etc).

$$\begin{aligned} \mathbb{H} \;=\; \{ & \mathit{Ord\ Int} \longmapsto \textit{d-O-I}, \\ & \mathit{Binary\ Int} \longmapsto \textit{d-B-I}, \\ & (\forall a \,.\, (\mathit{Binary}\ a, \mathit{Ord}\ a) \Rightarrow \mathit{Binary}\ (\mathit{Set}\ a)) \longmapsto \textit{d-B-S} \} \end{aligned}$$

(5)	$\mathbb{H} \vdash \mathit{Ord\ Int} \longmapsto \textit{d-O-I}$	A
(4)	$\mathbb{H} \vdash \mathit{Binary\ Int} \longmapsto \textit{d-B-I}$	A
(3)	$\mathbb{H} \vdash (\mathit{Binary\ Int}, \mathit{Ord\ Int}) \longmapsto (\textit{d-B-I}, \textit{d-O-I})$	C(4,5)
(2)	$\mathbb{H} \vdash (\forall a \,.\, (\mathit{Binary}\ a, \mathit{Ord}\ a) \Rightarrow \mathit{Binary}\ (\mathit{Set}\ a)) \longmapsto \textit{d-B-S}$	A
(1)	$\mathbb{H} \vdash \mathit{Binary}\ (\mathit{Set\ Int}) \longmapsto \textit{d-B-S}\ (\textit{d-B-I}, \textit{d-O-I})$	MP(2,3)

Here is another example, this time of a higher-order case:

$$\begin{aligned} \mathbb{H} \;=\; \{ & \mathit{Binary\ Int} \longmapsto \textit{d-B-I}, \\ & (\forall a \,.\, (\mathit{Binary}\ a) \Rightarrow \mathit{Binary}\ (\mathit{List}\ a)) \longmapsto \textit{d-B-L}, \\ & (\forall f\ a \,.\, (\mathit{Binary}\ a, \forall b \,.\, (\mathit{Binary}\ b) \Rightarrow \mathit{Binary}\ (f\ b)) \\ & \qquad \Rightarrow \mathit{Binary}\ (\mathit{GRose}\ f\ a)) \longmapsto \textit{d-B-G} \} \ . \end{aligned}$$

In the following we abbreviate *Binary* by B.

$$
\begin{array}{rll}
(10) & \mathbb{H}, (B\ c \longmapsto v) \vdash (\forall b\,.\,(B\ b) \Rightarrow B\ (\mathit{List}\ b)) \longmapsto \mathit{d\text{-}B\text{-}L} & \mathrm{A} \\
(9) & \mathbb{H}, (B\ c \longmapsto v) \vdash B\ c \longmapsto v & \mathrm{A} \\
(8) & \mathbb{H}, (B\ c \longmapsto v) \vdash (B\ c, \forall b\,.\,(B\ b) \Rightarrow B\ (\mathit{List}\ b)) \longmapsto (v, \mathit{d\text{-}B\text{-}L}) & \mathrm{C}(9,10) \\
(7) & \mathbb{H}, (B\ c \longmapsto v) \vdash (\forall f\ a\,.\,(\ldots) \Rightarrow B\ (\mathit{GRose}\ f\ a)) \longmapsto \mathit{d\text{-}B\text{-}G} & \mathrm{A} \\
(6) & \mathbb{H}, (B\ c \longmapsto v) \vdash B\ (\mathit{GRose}\ \mathit{List}\ c) \longmapsto \mathit{d\text{-}B\text{-}G}\ (v, \mathit{d\text{-}B\text{-}L}) & \mathrm{MP}(7,8) \\
(5) & \mathbb{H} \vdash (\forall b\,.\,(B\ b) \Rightarrow B\ (\mathit{GRose}\ \mathit{List}\ b)) & \\
 & \quad \longmapsto (\lambda v \to \mathit{d\text{-}B\text{-}G}\ (v, \mathit{d\text{-}B\text{-}L})) & \mathrm{DR}(6) \\
(4) & \mathbb{H} \vdash B\ \mathit{Int} \longmapsto \mathit{d\text{-}B\text{-}I} & \mathrm{A} \\
(3) & \mathbb{H} \vdash (B\ \mathit{Int}, \forall b\,.\,(B\ b) \Rightarrow B\ (\mathit{GRose}\ \mathit{List}\ b)) & \\
 & \quad \longmapsto (\mathit{d\text{-}B\text{-}I}, \lambda v \to \mathit{d\text{-}B\text{-}G}\ (v, \mathit{d\text{-}B\text{-}L})) & \mathrm{C}(4,5) \\
(2) & \mathbb{H} \vdash (\forall f\ a\,.\,(\ldots) \Rightarrow B\ (\mathit{GRose}\ f\ a)) \longmapsto \mathit{d\text{-}B\text{-}G} & \mathrm{A} \\
(1) & \mathbb{H} \vdash B\ (\mathit{GRose}\ (\mathit{GRose}\ \mathit{List})\ \mathit{Int}) & \\
 & \quad \longmapsto \mathit{d\text{-}B\text{-}G}\ (\mathit{d\text{-}B\text{-}I}, \lambda v \to \mathit{d\text{-}B\text{-}G}\ (v, \mathit{d\text{-}B\text{-}L})) & \mathrm{MP}(2,3)
\end{array}
$$

The new inference rule kicks in at line (5) and introduces a new assumption, $B\ c \longmapsto v$, that is used in line (2).

9 Related work

This paper improves on our earlier work (Hinze, 1999) in several respects. First, generic definitions now appear solely in class declarations as generic default methods. In the previous design generic definitions and classes were two competing features. We feel that the new proposal fits better with "the spirit of Haskell". Second, we have spelled out the implementation in considerable detail. Especially, the notion of generic representation types and the conversion between types and representation types has been made precise. Third, we have described a separate extension that allows the programmer to define instance declarations for higher-order kinded types. The need for this extension was noted in (Hinze, 1999) but no solution was given.

Though there is a considerable amount of work on generic programming (Ruehr, 1992; Cockett & Fukushima, 1992; Jay *et al.*, 1998) this is the first paper we are aware of — apart from PolyP (Jansson & Jeuring, 1997) — that aims at adding generic features to an *existing* functional language. The PolyP extension offers a special construct (essentially, a type case) for defining generic functions. The resulting definitions are similar to ours (modulo notation) except that the generic programmer must additionally consider cases for type composition and for type recursion. Furthermore, PolyP is restricted to regular datatypes of kind $\star \to \star$, whereas our proposal works for all types of all kinds. This is quite a significant advantage. In particular, our proposal deals gracefully with mutually-recursive data types and with data types with many parameters, both of which make explicit recursion operators clumsy and hard to use in practice.

10 Conclusions and further work

This paper describes two separate extensions to Haskell. The first extension supports generic programming through a new form of default method declaration. The second extension allows one to define instance declarations for higher-order kinded types through the notion of polymorphic predicates. Though both extensions are orthogonal to each other, the second ensures that one gets the most out of the first one (surely, one wants to derive instances for higher-order kinded types).

We are currently implementing the proposal and we hope to make the new features available in the next release of the Glasgow Haskell Compiler.

There are several directions we plan to explore in the future:

- Currently, generic default declarations may be given only for type classes. However, the theory (Hinze, 2000b) also deals with constructor classes whose type parameter range over types of first-order kind. Consequently, we plan to lift this restriction.
- In Haskell 98 instance heads must have the general form $C\ (T\ \bar{a})$ where $\bar{a}$ is a sequence of distinct variables. The Glasgow Haskell Compiler, however, allows for non-general instance heads such as $C\ (List\ Char)$. We are confident that the implementation scheme for generic methods can be extended to deal with this extra complication.
- Multi-parameter type classes are on the wish list of many Haskell programmers. So it would be a shame if the generic extension failed to support them. Now, multi-parameter classes correspond to generic definitions with multiple type arguments, which are theoretically well understood. So we are confident that we can also deal with this generalization.

References

Backhouse, R., Jansson, P., Jeuring, J. and Meertens, L. (1999) Generic Programming — An Introduction —. Swierstra, S. D., Henriques, P. R. and Oliveira, J. N. (eds), *3rd International Summer School on Advanced Functional Programming, Braga, Portugal.* Lecture Notes in Computer Science 1608, pp. 28–115. Springer-Verlag.

Bird, R., de Moor, O. and Hoogendijk, P. (1996) Generic functional programming with types and relations. *Journal of Functional Programming* **6**(1):1–28.

Cockett, R. and Fukushima, T. (1992) *About Charity.* Yellow Series Report 92/480/18. Dept. of Computer Science, Univ. of Calgary.

Hinze, R. (1999) A generic programming extension for Haskell. Meijer, E. (ed), *Proceedings of the 3rd Haskell Workshop, Paris, France.* The proceedings appear as a technical report of Universiteit Utrecht, UU-CS-1999-28.

Hinze, R. (2000a) Manufacturing datatypes. *Journal of Functional Programming, Special Issue on Algorithmic Aspects of Functional Programming Languages.* To appear.

Hinze, R. (2000b) A new approach to generic functional programming. Reps, T. W. (ed), *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21* pp. 119–132.

Hinze, R. (2000c) Polytypic values possess polykinded types. Wadler, P. (ed), *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000), July 3-5, 2000.*

Jansson, P. and Jeuring, J. (1997) PolyP—a polytypic programming language extension. *Conference Record 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'97, Paris, France* pp. 470–482. ACM-Press.

Jay, C., Bellè, G. and Moggi, E. (1998) Functorial ML. *Journal of Functional Programming* **8**(6):573–619.

Jones, M. P. (1994) *Qualified Types: Theory and Practice.* Cambridge University Press.

Jones, M. P. (2000) Type classes with functional dependencies. Smolka, G. (ed), *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany.* Lecture Notes in Computer Science 1782, pp. 230–244. Springer-Verlag.

Leivant, D. (1983) Polymorphic type inference. *Proc. 10th Symposium on Principles of Programming Languages.*

Lewis, J. R., Shields, M. B., Meijer, E. and Launchbury, J. (2000) Implicit parameters: Dynamic scoping with static types. Reps, T. W. (ed), *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, January 19-21* pp. 108–118.

Okasaki, C. (1998) *Purely Functional Data Structures.* Cambridge University Press.

Peyton Jones, S. and Hughes, J. (eds). (1999) *Haskell 98 — A Non-strict, Purely Functional Language.* Available from `http://www.haskell.org/definition/`.

Peyton Jones, S., Jones, M. and Meijer, E. (1997) Type classes: Exploring the design space. *Proceedings of the Haskell Workshop.*

Ruehr, K. F. (1992) *Analytical and Structural Polymorphism Expressed using Patterns over Types.* PhD thesis, University of Michigan.