# TypeCase: A Design Pattern for Type-Indexed Functions

Bruno C. d. S. Oliveira and Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{bruno,jg}@comlab.ox.ac.uk

## Abstract

A *type-indexed function* is a function that is defined for each member of some family of types. Haskell's type class mechanism provides collections of *open type-indexed functions*, in which the indexing family can be extended by defining a new type class instance but the collection of functions is fixed. The purpose of this paper is to present *TypeCase*: a design pattern that allows the definition of *closed type-indexed functions*, in which the index family is fixed but the collection of functions is extensible. It is inspired by Cheney and Hinze's work on lightweight approaches to generic programming. We generalise their techniques as a *design pattern*. Furthermore, we show that *type-indexed functions* with *type-indexed types*, and consequently *generic functions* with *generic types*, can also be encoded in a lightweight manner, thereby overcoming one of the main limitations of the lightweight approaches.

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

***General Terms*** Languages

***Keywords*** Generic programming, type classes, type-indexed functions

## 1. Introduction

A *type-indexed function* is a function that is defined for each member of a family of types. One of the most popular mechanisms implementing this notion is the Haskell [31] *type class* system. A type class consists of a collection of related type-indexed functions; the family of index types is the set of instances of the type class. Type classes provide just one possible interpretation of the notion of type-indexed functions. In particular, they assume an *open-world* perspective: the family of index types is extensible, by defining a new type class instance for that type, but the collection of type-indexed functions is fixed in the type class interface so needs to be known in advance. For some applications — particularly when providing a framework for generic programming — the family of index types is fixed (albeit large) and the collection of type-indexed functions is not known in advance, so a closed-world perspective would make more sense.



*Haskell'05* September 30, 2005, Tallinn, Estonia.



The original concept of a *design pattern* has its origins in Christopher Alexander's work in architecture, but it has been picked up with enthusiasm by the object-oriented programming community. The idea of design patterns is to capture, abstract and record beneficial recurring patterns in software design. Sometimes those patterns can be captured formally, as programming language constructs or software library fragments. Often, however, the appropriate abstraction cannot be directly stated, either because of a lack of expressiveness in the language, or because there is inherent ambiguity in the pattern — Alexander describes a pattern as a solution 'you can use [. . . ] a million times over, without ever doing it the same way twice' [1]. In this case, one must resort to an informal description. Even if the abstraction itself can be captured formally, one might argue that a complete description of the pattern includes necessarily informal information: a name, motivation, examples, consequences, implementation trade-offs, and so on.

In this paper, we present a technique that allows the definition of closed type-indexed functions, as opposed to the open type-indexed functions provided by type classes; we do so in the format of a design pattern. Our inspiration comes from previous research on lightweight approaches to generic programming (LAGP). In particular, Hinze's two papers "A Lightweight Implementation of Generics and Dynamics" [4] (LIGD, with James Cheney) and "Generics for the Masses" [19] (GM) provide our motivation and basis.

Those two papers focus on the particular context of generic programming, and provide a number of techniques that can be used to encode first-class generic functions in Haskell. However, those techniques have a wider applicability, not addressed by Hinze. We propose a generalisation of the technique, and demonstrate its use in a variety of applications. Our specific contributions are:

**Generalisation of the lightweight approaches.** We provide templates for designing closed type-indexed functions, abstracting away from generic programming. The techniques in LIGD and GM are instances of these templates.

**A design pattern for type-indexed functions.** We document this generalisation as a design pattern.

**Type-indexed functions with type-indexed types.** We show that with our more general interpretation of the design pattern, type-indexed functions with type-indexed types are also instances of the design pattern. As a consequence, generic functions with generic types can also be encoded in a lightweight manner. Thus, we remove one of the main limitations of the lightweight approaches.

**Other applications.** We present two other interesting applications of the pattern: PolyP in Haskell 98, and a very flexible *printf* function.

The remainder of this paper is structured as follows. In Section 2 we review the lightweight approaches to generic programming. In Section 3 we abstract the essence of the technique as a design pattern. Section 4 presents two other small applications of the design

pattern, and Section 5 uses it to model type-indexed functions with type-indexed types. Section 6 concludes.

## 2. Lightweight generic programming

We start by summarising the earlier work on lightweight approaches to generic programming underlying our generalisation.

### 2.1 "A Lightweight Implementation of Generics and Dynamics"

Cheney and Hinze [4] show how to do a kind of generic programming, using only the standard Hindley-Milner type system extended with existential types. The index family consists of hierarchical sums and products of integers and characters. This family is enough to represent a large subset of Haskell 98 datatypes (including mutually recursive and nested datatypes).

$$\begin{aligned}
&\textbf{data}\ \mathit{Sum}\ a\ b &&= \mathit{Inl}\ a \mid \mathit{Inr}\ b\\
&\textbf{data}\ \mathit{Prod}\ a\ b &&= \mathit{Prod}\ a\ b\\
&\textbf{data}\ \mathit{Unit} &&= \mathit{Unit}
\end{aligned}$$

This style of generic programming requires a representation of types as values in order to support typecase analysis. The key idea of the LIGD paper is to use a parametrised type as the type representation, ensuring that the type parameter reflects the type being represented. Some Haskell implementations have recently been extended with *generalised algebraic datatypes* (GADTs) [32], which can be used for this purpose; but LIGD predates that extension, and depends only on existential quantification.

$$\begin{aligned}
&\textbf{data}\ \mathit{Rep}\ t =\\
&\quad \mathit{RUnit} && (t \leftrightarrow \mathit{Unit})\\
&\mid \mathit{RInt} && (t \leftrightarrow \mathit{Int})\\
&\mid \mathit{RChar} && (t \leftrightarrow \mathit{Char})\\
&\mid \forall\, a\ b.\ \mathit{RSum}\ (\mathit{Rep}\ a)\ (\mathit{Rep}\ b)\ (t \leftrightarrow (\mathit{Sum}\ a\ b))\\
&\mid \forall\, a\ b.\ \mathit{RProd}\ (\mathit{Rep}\ a)\ (\mathit{Rep}\ b)\ (t \leftrightarrow (\mathit{Prod}\ a\ b))\\
&\textbf{data}\ a \leftrightarrow b = \mathit{EP}\{\mathit{from} :: a \to b, \mathit{to} :: b \to a\}
\end{aligned}$$

(Note that the universal quantifications are in contravariant positions, so act existentially.)

The intention is that the equivalence type $a \leftrightarrow b$ represents embedding/projection pairs witnessing to an isomorphism between types $a$ and $b$, thereby enforcing a correspondence between types $t$ and $\mathit{Rep}\ t$. Of course, within Haskell, it is not possible to automatically verify the isomorphisms ($\mathit{from} \circ \mathit{to} = \mathit{id}$ and $\mathit{to} \circ \mathit{from} = \mathit{id}$), so these laws should be externally checked. Furthermore, we follow the convention of ignoring the 'ugly fact' of bottom values destroying the 'beautiful theory' of many such isomorphisms [8].

A common case is with the trivial embedding/projections.

$$\begin{aligned}
&\mathit{self} :: a \leftrightarrow a\\
&\mathit{self} = \mathit{EP}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\}
\end{aligned}$$

Using *self*, we can provide a set of smart constructors for the *Rep* type, yielding representations of types by themselves.

$$\begin{aligned}
&\mathit{rUnit} :: \mathit{Rep}\ \mathit{Unit}\\
&\mathit{rUnit} = \mathit{RUnit}\ \mathit{self}\\
&\\
&\mathit{rInt} :: \mathit{Rep}\ \mathit{Int}\\
&\mathit{rInt} = \mathit{RInt}\ \mathit{self}\\
&\\
&\mathit{rChar} :: \mathit{Rep}\ \mathit{Char}\\
&\mathit{rChar} = \mathit{RChar}\ \mathit{self}\\
&\\
&\mathit{rSum} :: \mathit{Rep}\ a \to \mathit{Rep}\ b \to \mathit{Rep}\ (\mathit{Sum}\ a\ b)\\
&\mathit{rSum}\ \mathit{ra}\ \mathit{rb} = \mathit{RSum}\ \mathit{ra}\ \mathit{rb}\ \mathit{self}\\
&\\
&\mathit{rProd} :: \mathit{Rep}\ a \to \mathit{Rep}\ b \to \mathit{Rep}\ (\mathit{Prod}\ a\ b)\\
&\mathit{rProd}\ \mathit{ra}\ \mathit{rb} = \mathit{RProd}\ \mathit{ra}\ \mathit{rb}\ \mathit{self}
\end{aligned}$$

Using these smart constructors, we can build representations for recursive datatypes, by making explicit the structure isomorphism of the datatype. For instance, the isomorphism defining lists is $[a] \cong 1 + a \times [a]$, and so the corresponding type representation is as follows.

$$\begin{aligned}
&\mathit{rList} :: \forall\, a.\ \mathit{Rep}\ a \to \mathit{Rep}\ [a]\\
&\mathit{rList}\ \mathit{ra} = \mathit{RSum}\ \mathit{rUnit}\ (\mathit{rProd}\ \mathit{ra}\ (\mathit{rList}\ \mathit{ra}))\ (\mathit{EP}\ \mathit{from}\ \mathit{to})\\
&\quad \textbf{where}\ \mathit{from}\ [\,] &&= \mathit{Inl}\ \mathit{Unit}\\
&\qquad\quad \mathit{from}\ (x : \mathit{xs}) &&= \mathit{Inr}\ (\mathit{Prod}\ x\ \mathit{xs})\\
&\qquad\quad \mathit{to}\ \ (\mathit{Inl}\ \mathit{Unit}) &&= [\,]\\
&\qquad\quad \mathit{to}\ \ (\mathit{Inr}\ (\mathit{Prod}\ x\ \mathit{xs})) &&= x : \mathit{xs}
\end{aligned}$$

Note that the representation of a recursive datatype is an infinite value; but, because of laziness, this poses no problem.

Having constructed representation values for arbitrary types, the final step is to define generic functions. Using the representation as a basis for structural case analysis, it is possible to simulate a typecase [16]. For example, here is a definition of generic equality:

$$\begin{aligned}
&\mathit{eq} :: \forall\, t.\ \mathit{Rep}\ t \to t \to t \to \mathit{Bool}\\
&\mathit{eq}\ (\mathit{RInt}\ \mathit{ep}) && t_1\ t_2 = \mathit{from}\ \mathit{ep}\ t_1 == \mathit{from}\ \mathit{ep}\ t_2\\
&\mathit{eq}\ (\mathit{RChar}\ \mathit{ep}) && t_1\ t_2 = \mathit{from}\ \mathit{ep}\ t_1 == \mathit{from}\ \mathit{ep}\ t_2\\
&\mathit{eq}\ (\mathit{RUnit}\ \mathit{ep}) && _\ _ = \mathit{True}\\
&\mathit{eq}\ (\mathit{RSum}\ \mathit{ra}\ \mathit{rb}\ \mathit{ep}) && t_1\ t_2 = \textbf{case}\ (\mathit{from}\ \mathit{ep}\ t_1, \mathit{from}\ \mathit{ep}\ t_2)\ \textbf{of}\\
&&&\qquad (\mathit{Inl}\ x, \mathit{Inl}\ y) \to \mathit{eq}\ \mathit{ra}\ x\ y\\
&&&\qquad (\mathit{Inr}\ x, \mathit{Inr}\ y) \to \mathit{eq}\ \mathit{rb}\ x\ y\\
&&&\qquad _ \to \mathit{False}\\
&\mathit{eq}\ (\mathit{RProd}\ \mathit{ra}\ \mathit{rb}\ \mathit{ep})\ t_1\ t_2 = \textbf{case}\ (\mathit{from}\ \mathit{ep}\ t_1, \mathit{from}\ \mathit{ep}\ t_2)\ \textbf{of}\\
&&&\qquad (\mathit{Prod}\ x\ y, \mathit{Prod}\ x'\ y') \to\\
&&&\qquad\quad \mathit{eq}\ \mathit{ra}\ x\ x' \wedge \mathit{eq}\ \mathit{rb}\ y\ y'
\end{aligned}$$

Using Haskell type classes, it is possible to make the use of generic functions even more convenient: the class *TypeRep* can be used to build values of type *Rep t* implicitly.

$$\begin{aligned}
&\textbf{class}\ \mathit{TypeRep}\ t\ \textbf{where}\\
&\quad \mathit{rep} :: \mathit{Rep}\ t\\
&\\
&\textbf{instance}\ \mathit{TypeRep}\ \mathit{Unit}\ \textbf{where}\\
&\quad \mathit{rep} = \mathit{rUnit}\\
&\\
&\textbf{instance}\ \mathit{TypeRep}\ \mathit{Int}\ \textbf{where}\\
&\quad \mathit{rep} = \mathit{rInt}\\
&\\
&\textbf{instance}\ \mathit{TypeRep}\ \mathit{Char}\ \textbf{where}\\
&\quad \mathit{rep} = \mathit{rChar}\\
&\\
&\textbf{instance}\ (\mathit{TypeRep}\ a, \mathit{TypeRep}\ b) \Rightarrow \mathit{TypeRep}\ (\mathit{Sum}\ a\ b)\ \textbf{where}\\
&\quad \mathit{rep} = \mathit{rSum}\ \mathit{rep}\ \mathit{rep}\\
&\\
&\textbf{instance}\ (\mathit{TypeRep}\ a, \mathit{TypeRep}\ b) \Rightarrow \mathit{TypeRep}\ (\mathit{Prod}\ a\ b)\ \textbf{where}\\
&\quad \mathit{rep} = \mathit{rProd}\ \mathit{rep}\ \mathit{rep}\\
&\\
&\textbf{instance}\ \mathit{TypeRep}\ a \Rightarrow \mathit{TypeRep}\ [a]\ \textbf{where}\\
&\quad \mathit{rep} = \mathit{rList}\ \mathit{rep}
\end{aligned}$$

For example, we can now express generic equality with an implicit rather than explicit dependence on the representation.

$$\begin{aligned}
&\mathit{ceq} :: \forall\, t.\ \mathit{TypeRep}\ t \Rightarrow t \to t \to \mathit{Bool}\\
&\mathit{ceq}\ t_1\ t_2 = \mathit{eq}\ \mathit{rep}\ t_1\ t_2
\end{aligned}$$

### 2.2 "Generics for the Masses"

Hinze's later GM approach [19] has a very similar flavour to LIGD; however, somewhat surprisingly, Hinze shows how to do generic programming strictly within Haskell 98, which does not support rank-*n* types or even existential types. Nevertheless, there is a close relationship between type classes and polymorphic records (for example, one possible translation of type classes into System F uses polymorphic records), and these require something like existential types for their encoding. Thus, type class instances can be seen as implicitly-passed records. Hinze uses this observation to deliver two implementations of generics.

#### 2.2.1 Generic functions on types

The first implementation of generics in GM ("GM1", from now on) can be seen as a direct descendent of LIGD. Instead of using a

datatype with an existential quantification, Hinze uses a type class *Generic*.

```
class Generic g where
  unit     :: g Unit
  sum      :: (TypeRep a, TypeRep b) ⇒ g (Sum a b)
  prod     :: (TypeRep a, TypeRep b) ⇒ g (Prod a b)
  datatype :: TypeRep a ⇒ (b ↔ a) → g b
  char     :: g Char
  int      :: g Int
```

The parameter *g* of the type class represents the generic function, and each of the member functions of the type class encodes the behaviour of that generic function for one structural case. Generic functions over user-defined types can also be defined using the *datatype* type case. In this case, the isomorphism between the datatype and its structural representation must be provided.

The type class *TypeRep* is used to select the appropriate behaviour of the generic function, based on the type structure of its argument. The role of this type class is somewhat analogous to the synonymous one in Section 2.1. One contrast with LIGD is that *TypeRep* for GM1 is not optional, because the type representations are always implicitly passed.

```
class TypeRep a where
  typeRep :: Generic g ⇒ g a

instance TypeRep Unit where
  typeRep = unit

instance (TypeRep a, TypeRep b) ⇒ TypeRep (Sum a b) where
  typeRep = sum

instance (TypeRep a, TypeRep b) ⇒ TypeRep (Prod a b) where
  typeRep = prod

instance TypeRep Char where
  typeRep = char

instance TypeRep Int where
  typeRep = int
```

For GM, the type class *TypeRep* directly selects the appropriate behaviour for a particular structural case from the generic function. In contrast, for LIGD, the corresponding type class *TypeRep* builds a value as a type representation for a particular structural case, and this representation is then used by a generic function to select the appropriate behaviour. The effect is the same, but GM is more direct.

A new generic function is defined via an instance of *Generic*, providing an implementation for each structural case. For instance, the generic function *gSize* that counts all the elements of type *Int* and *Char* in some structure could be encoded as follows.

```
newtype GSize a = GSize{ appGSize :: a → Int }

instance Generic GSize where
  unit       = GSize (λ_ → 0)
  sum        = GSize (λt → case t of
                       Inl x → gSize x
                       Inr y → gSize y)
  prod       = GSize (λt → case t of
                       Prod x y → gSize x + gSize y)
  datatype iso = GSize (λt → gSize (from iso t))
  char       = GSize (λ_ → 1)
  int        = GSize (λ_ → 1)

gSize :: TypeRep a ⇒ a → Int
gSize = appGSize typeRep
```

A record of type *GSize a* contains a single function *appGSize* of type $a \to Int$, which can be used to compute the number of elements in some structure of type *a*. The function *gSize*, which is the actual generic function, simply extracts the sole *appGSize* field from a record of the appropriate type, built automatically by *typeRep*.

### 2.2.2 Generic functions on type constructors

The second implementation of generics in GM ("GM2") permits parametrisation by type constructors rather than by types. For example, whereas the generic function *gSize* of the previous section has type $a \to Int$ for all first-order types *a* in the type class *TypeRep*, in this section we show a generic function *gSize* with type $f\ a \to Int$ for all type constructors *f* in the constructor class *FunctorRep*.

Lifting in this fashion introduces the possibility of ambiguity: a type $g\ (f\ a)$ may be considered a type constructor *g* applied to a type $f\ a$, or the composition of constructors *g* and *f* applied to type *a*. Therefore we must explicitly pass type representations, increasing flexibility but decreasing brevity. This is reflected in the analogous type class *Generic*, where the implicitly-passed *TypeRep* contexts are now changed to explicitly-passed functions.

```
class Generic g where
  unit     :: g Unit
  sum      :: g a → g b → g (Sum a b)
  prod     :: g a → g b → g (Prod a b)
  datatype :: (b ↔ a) → g a → g b
  char     :: g Char
  int      :: g Int
```

However, this modification of the type class restricts expressivity, since the only generic function we can call is the one being defined, recursively. Consequently, generic functions that perform calls to other generic functions (as when defining generic membership in terms of generic equality) become harder to define.

With the new *Generic* class it is also possible to build the values for type representations automatically, using another type class *TypeRep*. Just as with LIGD, this class now becomes optional. Alternatively, we can use a type class *FunctorRep* to capture the notion of unary type constructor or *functor*.

```
class FunctorRep f where
  functorRep :: Generic g ⇒ g a → g (f a)
```

We have to define similar classes for each arity of type constructor.

Generic functions are defined in a very similar fashion to GM1. For instance, the type *Count a* below represents a generic function that counts zero for each occurrence of a value of type *Int* or *Char* in some structure of type *a*.

```
newtype Count a = Count{ applyCount :: a → Int }

instance Generic Count where
  unit       = Count (λ_ → 0)
  sum a b    = Count (λx → case x of
                 Inl l → applyCount a l
                 Inr r → applyCount b r)
  prod a b   = Count (λ(Prod x y) →
                 applyCount a x + applyCount b y)
  datatype iso a = Count (λx →
                 applyCount a (from iso x))
  char       = Count (λ_ → 0)
  int        = Count (λ_ → 0)
```

While this function by itself approximates *const* 0, it is the basis for other more useful functions that really count the number of elements in some structure in some way, by overriding the behaviour of the basic generic function for occurrences of the type parameter:

```
gSize :: FunctorRep f ⇒ f a → Int
gSize = applyCount (functorRep (Count (λ_ → 1)))
```

The payback of using *FunctorRep* is that we can define the behaviour of the generic function for its parameters. For instance, we could sum all the integers in some integer-parametrised datatype by using the identity function to define the behaviour of the generic function for the type parameter.

```
gSum :: FunctorRep f ⇒ f Int → Int
gSum = applyCount (functorRep (Count id))
```

## 3. Closed type-indexed functions

In LIGD and GM, we are shown three methods for implementing closed type-indexed functions. Those three variations give us different expressive power, and impose different constraints on the type system. A choice of implementation techniques, together with technical trade-offs making no one method superior in all circumstances, is characteristic of design patterns.

In this section, we introduce the TypeCase design pattern, capturing the different techniques for implementing closed type-indexed functions.

**The TypeCase design pattern**

***Intent:*** Allowing the definition of *closed type-indexed functions*.

***Motivation:*** The *typecase* design pattern captures a closed-world view of *ad-hoc polymorphism*. In Haskell, the type class system is a mechanism that supports ad-hoc polymorphism, but from an open-world point of view: they can be extended with cases for new datatypes, at the cost of a non-extensible set of functions. Under the closed-world assumption, there is a fixed set of type-structural cases but arbitrarily many type-indexed functions ranging over those cases. An example where the closed-world perpective works better than the open-world one is *generic programming*, in which we take a structural perspective on types as opposed to the more traditional nominal one. Using just a few operations on types, it is possible to represent the whole family of structural definitions of interest. For instance, here is a possible definition for a generic function that counts all the elements of some structure $t$:

$$
\begin{array}{llll}
gsize\langle t :: \star \rangle & & :: t \to Int \\
gsize\langle Unit \rangle & \_ & = 0 \\
gsize\langle Int \rangle & \_ & = 1 \\
gsize\langle Sum\ \alpha\ \beta \rangle & (Inl\ x) & = gsize\langle \alpha \rangle\ x \\
gsize\langle Sum\ \alpha\ \beta \rangle & (Inr\ y) & = gsize\langle \beta \rangle\ y \\
gsize\langle Prod\ \alpha\ \beta \rangle & (Prod\ x\ y) & = gsize\langle \alpha \rangle\ x + gsize\langle \beta \rangle\ y
\end{array}
$$

With an open-world perspective, we can present a fixed number of type-indexed definitions that range over those few cases; but we cannot easily introduce new definitions. This is clearly not appropriate for generic programming. In fact, what we expect from a generic programming facility is the ability to a introduce new generic definition without affecting the surrounding context. This is precisely what the closed-world perspective provides us.

***Applicability:*** Use this pattern:

- to encode collections of definitions that are *indexed by some fixed family of types*, while allowing new definitions to be added to the collection without affecting modularity;
- when a definition is *variadic*, that is, it has a variable number of arguments (see Section 4.2 for an example);
- to try to avoid *type-class trickery*, such as multiple-parameter type classes, functional dependencies, overlapping instances or even duplicate instances (just consider a direct encoding of the examples presented in the paper into type classes [30]);
- to capture some *shape invariants*, like the ones captured by some nested types or phantom types [29, 18].

***Structure:*** See Figure 1.

***Participants:***

- *Structural Cases*: a set of datatypes which represent the possible structural cases for the type-indexed function;
- *Typecase*: representing the structure of a type-indexed function;
- *Dispatcher*: a type class, containing a single function, that is responsible for dispatching a value of one of the structural cases into the corresponding branch of the *typecase*, based on the type of the value;
- *Type-indexed function*: defining the type-indexed function using an instance of the *typecase*.

***Collaborations:***

- The *typecase* uses the *structural cases* in order to create a corresponding number of cases that can be used to define the *type-indexed function*.
- The *dispatcher* uses the *structural cases* in order to create a corresponding number of instances that will forward some value of that family of structural cases into the corresponding case in the *typecase* component.
- The *type-indexed function* (TIF) uses an instance of the *typecase* in order to implement the desired functionality for the type-indexed function.

***Implementation:*** Typically, a *typecase* component is created using the *structural cases*. There are three main variations for the implementation of a *typecase*: two of them are based on type classes and the other one on a *smart datatype*. A smart datatype is a parametrised type where the type parameters are dependent on the constructors. The idea of a smart datatype can be represented in various forms: *existential* datatypes with an equivalence type (à la LIGD), *GADTs*, *phantom types*, among others.

The goal of this design pattern is to simulate a closed type-indexed function. In general, a type-indexed function $f$ has the following structure.

$$
\begin{array}{l}
f\langle t :: \kappa \mid d_1\ \ldots\ d_k \rangle :: \psi \\
f\langle t_1\ a_1\ \ldots\ a_i \rangle = \lambda\, x_{11}\ \ldots\ x_{1n} \to e_1 \\
\qquad \vdots \\
f\langle t_m\ z_1\ \ldots\ z_j \rangle = \lambda\, x_{m1}\ \ldots\ x_{mn} \to e_m
\end{array}
$$

The type signature tells us that $f$ has one type parameter $t$ and optional type parameters $d_1\ \ldots\ d_k$ with the same structure and kind as $t$. The type $\psi$ of the TIF may depend on $t$ and $d_1\ \ldots\ d_k$.

We should note that this is not the same as having a TIF with multiple type arguments. There is no problem, in principle, in having multiple-parameter type arguments, but it would lead to an explosion in the number of typecases. This would be a generalisation of this design pattern. For simplicity, we will only consider type parameters with the same structure. The usefulness of this simpler case is reflected in applications such as *generic map* where the input and output structures of the generic map function are the same.

The body of $f$ contains (at least) $m$ branches, providing the behaviour of the TIF for each member of the family of types $t$ (that is, $t_1\ a_1\ \ldots\ a_i, \ldots, t_m\ z_1\ \ldots\ z_j$). This family of types corresponds to the *structural cases* participant of the design pattern. For each branch of the definition, we bind possible variables $x_{11}\ \ldots\ x_{1n}, \ldots, x_{m1}\ \ldots\ x_{mn}$ and define each typecase of $f$ with $e_1, \ldots, e_m$.

We now discuss the three main variations of the design pattern.

1. *Smart datatypes:* This variation is inspired by the LIGD approach. Hindley-Milner typing extended with existential datatypes (supported in most Haskell compilers) is enough to encode it. However, with extensions such as GADTs (supported by GHC 6.4) the encoding becomes much more direct. Unfortunately, neither of those extensions conforms to Haskell 98. We will present this version of the design pattern using a GADT syntax for simplicity.

   Using the *structural cases* given by $t_1\ a_1\ \ldots\ a_i, \ldots, t_m\ z_1\ \ldots\ z_j$, we can derive the *typecase* and *dispatcher* seen in Figure 1. Since there are $m$ structural cases in a standard instance of the design pattern, one would create $m$ constructors $c_{t_1}, \ldots, c_{t_m}$ and also $m$ instances for $Rep_\Gamma$. TIFs can now be defined using those components, by creating some function $f$ that takes a first argument of type $Rep_\Gamma$ and returns a value of type $\psi$.

| | *Smart Datatype* | *Implicit/Explicit Representations* |
|---|---|---|
| *Typecase* | **data** $\Gamma\ t\ d_1\ \ldots\ d_k$ **where**<br>$c_{t_1} :: \Sigma_{(a_1\ \ldots\ a_i)} \rightsquigarrow \Gamma\ (t_1\ a_1\ \ldots\ a_i)\ d_{11}\ \ldots\ d_{1k}$<br>$\vdots$<br>$c_{t_m} :: \Sigma_{(z_1\ \ldots\ z_j)} \rightsquigarrow \Gamma\ (t_m\ z_1\ \ldots\ z_j)\ d_{m1}\ \ldots\ d_{mk}$ | **class** $\Gamma\ (g :: \kappa^{k+1} \to \star)$ **where**<br>$case_{t_1} :: \Sigma_{(a_1\ \ldots\ a_i)} \rightsquigarrow g\ (t_1\ a_1\ \ldots\ a_i)\ d_{11}\ \ldots\ d_{1k}$<br>$\vdots$<br>$case_{t_m} :: \Sigma_{(z_1\ \ldots\ z_j)} \rightsquigarrow g\ (t_m\ z_1\ \ldots\ z_j)\ d_{m1}\ \ldots\ d_{mk}$ |
| *Dispatcher* | **class** $Rep_\Gamma\ t\ d_1\ \ldots\ d_k$ **where**<br>$rep :: Rep_\Gamma\ t\ d_1\ \ldots\ d_k$<br>**instance** $\Omega_{(a_1\ \ldots\ a_i)} \Rightarrow$<br>$Rep_\Gamma\ (t_1\ a_1\ \ldots\ a_i)\ d_{11}\ \ldots\ d_{1k}$ **where**<br>$rep = c_{t_1}\ rep^i$<br>$\vdots$<br>**instance** $\Omega_{(z_1\ \ldots\ z_j)} \Rightarrow$<br>$Rep_\Gamma\ (t_m\ z_1\ \ldots\ z_j)\ d_{m1}\ \ldots\ d_{mk}$ **where**<br>$rep = c_{t_m}\ rep^j$ | **class** $Rep_\Gamma\ t\ d_1\ \ldots\ d_k$ **where**<br>$rep :: \Gamma\ g \Rightarrow g\ t\ d_1\ \ldots\ d_k$<br>**instance** $\Omega_{(a_1\ \ldots\ a_i)} \Rightarrow$<br>$Rep_\Gamma\ (t_1\ a_1\ \ldots\ a_i)\ d_{11}\ \ldots\ d_{1k}$ **where**<br>$rep = case_{t_1}\{rep^i\}$<br>$\vdots$<br>**instance** $\Omega_{(z_1\ \ldots\ z_j)} \Rightarrow$<br>$Rep_\Gamma\ (t_m\ z_1\ \ldots\ z_j)\ d_{m1}\ \ldots\ d_{mk}$ **where**<br>$rep = case_{t_m}\{rep^j\}$ |
| *Type-indexed function* | $f :: \Gamma\ t\ d_1\ \ldots\ d_k \to \psi$<br>$f\ (c_{t_1}\ r_{a_1}\ \ldots\ r_{a_i}) = \lambda\, x_{11}\ \ldots\ x_{1n} \to [\![e_1]\!]$<br>$\vdots$<br>$f\ (c_{t_m}\ r_{z_1}\ \ldots\ r_{z_j}) = \lambda\, x_{m1}\ \ldots\ x_{mn} \to [\![e_m]\!]$<br>$f' :: Rep_\Gamma\ t\ d_1\ \ldots\ d_k \Rightarrow \psi$<br>$f' = f\ rep$ | **newtype** $F\ t\ d_1\ \ldots\ d_k = F\{f :: \psi\}$<br>$f' :: Rep_\Gamma\ t\ d_1\ \ldots\ d_k \Rightarrow \psi$<br>$f' = f\ rep$<br>**instance** $Rep_\Gamma\ F$ **where**<br>$case_{t_1}\{r_{a_1}\ \ldots\ r_{a_i}\} = \lambda\, x_{11}\ \ldots\ x_{1n} \to [\![e_1]\!]$<br>$\vdots$<br>$case_{t_m}\{r_{z_1}\ \ldots\ r_{z_j}\} = \lambda\, x_{m1}\ \ldots\ x_{mn} \to [\![e_m]\!]$ |

**Figure 1.** The structure of the *TypeCase* design pattern.

The *dispatcher* component is optional in this variation. The TIFs created with this variation are fully closed to extension; no customisation is possible. This means that if we want to add extra functionality we need to modify the smart datatype (and the dispatcher if we have one). However, TIFs that call other TIFs are trivial to achieve; there is no need for tupling.

2. *Implicit representations:* The implicit representation version of the design pattern is inspired by GM1. Perhaps surprisingly, some implementations of this instance require only Haskell 98. However, if we need to have structurally-dependent variables, then we also require multiple-parameter type classes.

   Proceeding in a similar fashion to the *smart datatype* approach, we use the *structural cases* to derive the *typecase* and *dispatcher* seen in Figure 1. Again, because we have $m$ structural cases, we create $m$ functions $case_{t_1}, \ldots, case_{t_m}$ and $m$ instances of $Rep_\Gamma$.

   The *dispatcher* is not an optional component: it always needs to be defined in this variation. As with the smart datatype variation, TIFs defined in this way are fully closed to extension, and calls to other TIFs are trivial.

3. *Explicit representations:* The explicit representation variation of the design pattern is inspired by GM2. Like the implicit approach, Haskell 98 is enough to handle the simpler forms (one type parameter). However, if we discard the optional *dispatcher*, then Haskell 98 can handle all forms.

   Using the *structural cases* to derive the *typecase* and *dispatcher* seen in Figure 1, we would obtain a very similar structure to the implicit representation version. The most noticeable difference is that, with the explicit representation, the definition of *rep* needs to provide the corresponding *case* function with the representations for each of its type parameters. The second difference is that $\Sigma$, which corresponds to the representations of the type parameters, reflects the fact that we are providing explicit representations. Thus, $\Sigma$ corresponds in this instance to explicit arguments of the function, while with the implicit representation it corresponds to (implicitly passed) type class constraints. The dispatcher is an optional component.

Variations of this instance of the design pattern can also be found in the literature [10, 37], as described in Section 4.2. TIFs defined in this fashion are not fully closed to extension: it is possible to override default behaviour. However, the extra flexibility comes at a cost: recursive calls to other TIFs are not possible. One common solution for this problem is to *tuple* together into a record the mutually-dependent functions. Another possibility would be to have a notion of dependencies: if a TIF $f$ requires calls to another TIF $g$, then the record that defines $f$ has a field that is an instance of $g$. Although this work is quite tedious, Löh [26] shows how a type system can lighten the burden.

An associated problem for TIFs in this setting is the issue of *composability*. If two TIFs are defined using different instances (this is, they are not tupled together), then we cannot, in a straightforward manner, use the same representation to compose them. To illustrate the problem, consider:

**newtype** $F\ v_1\ \ldots\ v_n = F\{f :: \alpha\}$
**newtype** $G\ v_1\ \ldots\ v_n = G\{g :: \beta\}$
**instance** *Generic* $F$ **where** $\ldots$
**instance** *Generic* $G$ **where** $\ldots$

Now let us suppose that we define a *type-indexed abstraction* (that is, a function that uses one or more TIFs and is not defined over the structure of types):

$h\ rep = \ldots\ f\ rep\ \ldots\ g\ rep\ \ldots$

The interpretation of this definition as a type-indexed function could be thought of as: $h\langle a\rangle = \ldots\ f\langle a\rangle \ldots\ g\langle a\rangle \ldots$. While this is a perfectly reasonable interpretation, in practice $f$ requires inconsistent types $F\ v_1\ \ldots\ v_n$ and $G\ v_1\ \ldots\ v_n$ for *rep*: $F$ and $G$ are two different type constructors, so in a Hindley-Milner type system, unification obviously fails. However, $F$ and $G$ do have something in common. In particular, they are both

instances of *Generic*. So, in Haskell extended with higher-order polymorphism, we can capture this relation with a rank-2 type, thus providing a possible solution for the problem of composability.

```
h :: (∀ g. Generic g ⇒ g v1 ... vn) → ψ
h rep = ... f rep ... g rep ...
```

We should note that even though we have presented three main variations of the design pattern, the concept of a design pattern is, by itself, quite informal and thus prone to different interpretations. For instance, as we will see later, applications of the pattern (such as GM) can have more type cases than there are datatype variants, because some cases overlap. It is important to note that, depending on the context of a problem, a design pattern can be adapted to better fit that problem.

## 4. Applications

We present two applications of the design pattern. In Section 4.1, still within the context of generic programming, we show how one can build a library inspired by PolyP [21, 22] but working in Haskell 98. In Section 4.2, we present a very flexible version of a C-style *printf* function.

### 4.1 Light PolyP

It probably comes as no surprise to the reader that the technique introduced in GM and LIGD can be applied to other generic programming approaches as well. *PolyP* was one of the first attempts to produce a generic programming language. It is a simpler language than Generic Haskell, working in a much more restricted family of datatypes, namely one-parameter regular types. But this restriction allows stronger properties to be stated: its simplicity and strong theoretical background make it an appropriate language for teaching both the theory [3] and practice of generic programming. Our proposal *Light PolyP* encourages this, because no external PolyP compiler is required (although one might still be desirable, for a more convenient syntax).

Norell [30] shows how to use the Haskell type class system (extended with multiple-parameter type classes and functional dependencies) to obtain first-class PolyP generic functions in Haskell. In this section, we will present a "lighter" version of PolyP, requiring only Haskell 98 (without extensions such as multiple-parameter type classes and functional dependencies) but with the same expressive power.

Instead of using sums of products like LAGP or Generic Haskell, PolyP uses lifted *pattern functors* as *structural cases*. The pattern functors *Empty*, *Plus* and *Prod* have counterparts in LAGP. The pattern functors *Rep* and *Par* correspond respectively to the recursive argument and the parameter of the unary regular datatype. The pattern functor *Const t* for some type *t* represents the constant functor, and *Comp* handles the composition of functors required for regular types.

```
data Empty p r          = Empty
data Plus g h p r       = Inl (g p r) | Inr (h p r)
data Prod g h p r       = Prod (g p r) (h p r)
newtype Par p r         = Par{unPar :: p}
newtype Rec p r         = Rec{unRec :: r}
newtype Comp d h p r    = Comp{unComp :: d (h p r)}
newtype Const t p r     = Const{unConst :: t}
```

The equivalence type is used to establish the isomorphism between a regular datatype and its top-level structure. The embedding/projection functions are traditionally called *inn* and *out*.

```
data Iso a b = Iso{inn :: a → b, out :: b → a}

listIso = Iso inL outL
  where
    inL  (Inl Empty)                = []
    inL  (Inr (Prod (Par x) (Rec xs))) = x : xs
    outL []         = Inl Empty
    outL (x : xs)   = Inr (Prod (Par x) (Rec xs))
```

In PolyP no generic customisation is allowed, thus we can use an implicit representation version of the design pattern and consequently, it is possible for one generic function to use other generic functions in its definition. The *typecase* component corresponds to:

```
class Generic f where
  empty    :: f Empty
  plus     :: (Rep g, Rep h) ⇒ f (Plus g h)
  prod     :: (Rep g, Rep h) ⇒ f (Prod g h)
  par      :: f Par
  rec      :: f Rec
  comp     :: (Functor d, Rep h) ⇒ f (Comp d h)
  constant :: f (Const t)
```

The *dispatcher* simply selects the corresponding case based on the type of the argument of the generic function *g*.

```
class Rep g where
  rep :: Generic f ⇒ f g
instance Rep Empty where
  rep = empty
instance (Rep g, Rep h) ⇒ Rep (Plus g h) where
  rep = plus
instance (Rep g, Rep h) ⇒ Rep (Prod g h) where
  rep = prod
instance Rep Par where
  rep = par
instance Rep Rec where
  rep = rec
instance (Functor d, Rep h) ⇒ Rep (Comp d h) where
  rep = comp
instance Rep (Const t) where
  rep = constant
```

Like GM, defining a generic function is a matter of declaring a record with a single field, a function of the appropriate type. As an example, we could define *fmap2*, the map operation for binary functors, as follows.

```
newtype FMap2 a b c d f = FMap2{
  appFMap2 :: (a → c) → (b → d) → f a b → f c d}

instance Generic (FMap2 a b c d) where
  empty    = FMap2 (λ_ _ _ → Empty)
  plus     = FMap2 (λf g t → case t of
               Inl x  → Inl (fmap2 f g x)
               Inr y  → Inr (fmap2 f g y))
  prod     = FMap2 (λf g t → case t of
               Prod x y → Prod (fmap2 f g x) (fmap2 f g y))
  par      = FMap2 (λf g (Par t) → Par (f t))
  rec      = FMap2 (λf g (Rec t) → Rec (g t))
  comp     = FMap2 (λf g (Comp t) →
               Comp (fmap (fmap2 f g) t))
  constant = FMap2 (λ_ _ (Const t) → (Const t))

fmap2 :: Rep f ⇒ (a → c) → (b → d) → f a b → f c d
fmap2 = appFMap2 rep
```

With *fmap2* it is now possible to define several widely-applicable recursion operators [28, 14] using PolyP. For example, the *catamorphism* operator could be defined as:

```
cata iso f = f ∘ fmap2 id (cata iso f) ∘ out iso
```

Note that one must give explicitly the isomorphism that converts between the datatype and its representation. This contrasts with the original PolyP approach, in which that translation is inferred. This is the common trade-off of brevity for flexibility; being forced to state the isomorphism allows the programmer to choose a different one, giving something analogous to Wadler's ideas about

*views* [34]. We might say that this style of generic programming is *isomorphism-parametrised* instead of *datatype-parametrised*.

In the original PolyP, the *polytypic* construct provides a convenient syntax for encoding generic functions. Furthermore, combinators for pointfree programming may be provided, making generic definitions even more compact. These combinators are just normal Haskell functions, and so there is no problem in implementing them in pure Haskell; but to keep the example short, we have stuck with pointwise definitions.

The advantages of this translation when compared with the one proposed in [30] are that it requires only Haskell 98, and that the types of the generic functions are much closer to what one would expect. In Norell's translation, the type class constraints posed some problems because both the two-parameter class *FunctorOf* and the classes for the generic functions propagated throughout the code. With the Light PolyP approach, only instances of *Rep* propagate, leading usually to just one type class constraint.

### 4.2 Printf

The C-style *printf* function, which takes a variable number of parameters, has always been a challenge for programmers using strongly and statically typed languages. The problem with *printf* is that, in its true essence, it requires dependent types. This happens because the value of the format string determines the type of the function. However, it has been shown by Danvy [10] that by changing the representation of the control string it is possible to encode *printf* in any language supporting a standard Hindley-Milner type system.

#### 4.2.1 A solution using explicit representations

In this section, we will demonstrate that Danvy's solution is another instance of the TypeCase design pattern, using an explicit representation. Furthermore, we will show a new use of the *printf* function by making use of the fact that we can (in some cases) infer the format string.

Danvy's original solution had the following combinators:

```
lit       :: String → (String → a) → String → a
lit x k s  = k (s ++ x)

eol       :: (String → a) → String → a
eol k s    = k (s ++ "\n")

int       :: (String → a) → String → Int → a
int k s x  = k (s ++ show x)

str       :: (String → a) → String → String → a
str k s x  = k (s ++ x)

eod       :: String → String
eod        = id
```

If we capture all the occurrences of the form $String \to t$ with a newtype *Printf*, and modify the definitions in order to reflect this newtype, we obtain the following code.

```
newtype Printf t = Printf {printfApp :: String → t}

lit       :: String → Printf a → Printf a
lit x k    = Printf (λs → printfApp k (s ++ x))

eol       :: Printf a → Printf a
eol k      = Printf (λs → printfApp k (s ++ "\n"))

int       :: Printf a → Printf (Int → a)
int k      = Printf (λs x → printfApp k (s ++ show x))

str       :: Printf a → Printf (String → a)
str k      = Printf (λs x → printfApp k (s ++ x))

eod       :: Printf String
eod        = Printf id
```

Taking one step further, we can now abstract over *Printf* and create a type class that replaces it with some functor $f$.

```
class Format f where
  lit  :: String → f r → f r
  eol  :: f r → f r
  int  :: f r → f (Int → r)
  str  :: f r → f (String → r)
  eod  :: f String
```

With this last transformation, we can start seeing an instance of the TypeCase design pattern. The *structural cases* participant consists of functions of the form $Int \to r$ or $String \to r$, or a *String — lit* and *eol* are overlapping cases. The class *Format* constitutes the *typecase* participant. Because the *dispatcher* is optional in explicit versions of the design pattern, there is no obligation to define it. Now, using the newtype *Printf*, we can define an instance of *Format* that implements the functionality of *printf*.

```
instance Format Printf where
  lit x k = Printf (λs → printfApp k (s ++ x))
  eol k   = Printf (λs → printfApp k (s ++ "\n"))
  int k   = Printf (λs x → printfApp k (s ++ show x))
  str k   = Printf (λs x → printfApp k (s ++ x))
  eod     = Printf id
```

The final touch is provided by the definition of *printf* in terms of *printfApp*. The *printf* function is expected to receive the formatting argument of type *Printf t* as its first parameter. The parameter *t* defines the type of *printf*, which can involve a variable number of arguments. Analysing the type of *printfApp*, we see that the first parameter is the formatting argument, the resulting type is the type that we expect for *printf*, and there is a second argument which is a *String*. Now, what does that *String* represent? Danvy's solution uses a continuation-passing style and the second argument of *printfApp* corresponds to the value fed to the initial continuation. Thus using the string `""` for that argument does the trick.

```
printf   :: Printf t → t
printf p = printfApp p ""
```

We have shown, informally, that Danvy's solution is indeed an instance of the TypeCase design pattern. However, some questions might be asked at this point. Do we really need to create a class in order to implement *printf*? What other instances of the class would we be able to provide? In fact there are not many other uses for the type class; *printf* seems to be the only natural instance. Perhaps we could consider *scanf*, another C function that uses the same format string; but the derived type for *scanf* would be different, and so it is not possible to reuse the same type class. Another possibility would be considering other versions of *printf*, such as one for the *IO* monad. However, if we think that *printf* is really the only useful instance of the type class, why not get rid of the type class all together?

A design pattern is a flexible design, and depending on the context of the problem, it can be adapted to fit the problem. If a type-indexed function is used at *just one type index*, it is reasonable to simplify the pattern and *eliminate the type class*. The result would be the specialised solution using the newtype *Printf t* presented before. We could go even further and argue that Danvy's original solution is already an instance of the design pattern, corresponding to one further simplification of the design pattern, namely getting rid of the newtype.

#### 4.2.2 An alternative solution using smart datatypes

In the previous section, we have argued that Danvy's version of *printf* is an instance of the TypeCase design pattern. However, Danvy's solution and explanation for *printf* is not, perhaps, very intuitive to understand. In this section, we take a different perpective and will look at the formatting parameter of *printf* as a special kind of list. This perpective corresponds to an instance of the design pattern using a *smart datatype*. The datatype (the *typecase* participant) encodes a list, which has an empty case that corres-

ponds to the combinator *eod*, and a number of recursive cases that correspond to *lit*, *eol*, *int* and *str*.

```
data Printf t where
  Lit  :: String → Printf t → Printf t
  Eol  :: Printf t → Printf t
  Int  :: Printf t → Printf (Int → t)
  Str  :: Printf t → Printf (String → t)
  Eod  :: Printf String
```

Informally speaking, we have reused the types from the newtype solution and lifted the functions to constructors. However, using a datatype instead of a number of functions makes it easier to view the format parameter of *printf* as a list. For instance, the *Lit* constructor takes the literal string that we wish to print and also the list corresponding to the rest of the format parameter of *printf*.

The *printfApp* from the previous section would, in this setting, correspond to a dependently-typed function (in the sense that the types of its branches are determined by the constructors used to perform pattern matching).

```
printfApp              :: Printf t → String → t
printfApp (Lit x k) s  = printfApp k (s ++ x)
printfApp (Eol k) s    = printfApp k (s ++ "\n")
printfApp (Int k)  s   = (λx → printfApp k (s ++ show x))
printfApp (Str k)  s   = (λx → printfApp k (s ++ x))
printfApp Eod      s   = s
```

The final step is to define *printf*. Little effort is required; we just need to copy the definition of *printf* from the previous section. The only apparent difference between the two versions is that, where the first version uses functions like *lit* and *int*, this version uses constructors like *Lit* and *Int*. However, despite the similarity of the two solutions, their expressive power is not the same. The smart datatype solution in this section is fully closed to extension. That is, in order to add another case in the formatting list, such as a constructor *Chr* that handles characters, we would need to modify the GADT itself. On the other hand, the solution in the previous section using the explicit version of the design pattern allows some form of extensibility. Adding a new case for *printf* that handles characters corresponds to adding a new function, which could even be in a different module.

#### 4.2.3 Making use of a dispatcher

The two solutions that we presented did not make any use of a *dispatcher*. In this section we will show how the dispatcher can be useful. The version of the dispatcher presented here is for the explicit representation solution in Section 4.2.1, but could be easily adapted to the smart datatype solution in Section 4.2.2.

Suppose that we want to define a function that prints a pair of integers. Equipped with *printf*, we could try to encode that with either one of the following two functions.

```
printPair x y = printf fmt "(" x ", " y ")"
  where
    fmt = str $ int $ str $ int $ str $ eod

printPair2 x y = printf fmt x y
  where
    fmt = lit "(" $ int $ lit ", " $ int $ lit ")" $ eod
```

The function *printPair* tackles the problem using a *printf* that takes a format argument expecting five arguments: three strings and two integers. The function *printPair2*, on the other hand, makes use of the fact that the string arguments are constants, and uses *lit* instead. Thus, in this case, *printf* takes the format argument and two integer arguments. Although relatively compact, the format argument is not as convenient to use as it would be in C, where one would write something like `"(%d, %d)"`.

The role of the dispatcher is to infer automatically the corresponding type representation for some type *t*. In the case of *printf*, it is not possible to infer all possible representations. Consider, for instance, the end of line case $eol :: f\ r \to f\ r$, which takes an existing format with some type *r*, adds a newline and returns a format of the same type. Clearly, there is no way to deduce that there is an occurrence of *eol* based on the type alone. Similarly, the *lit* case has no effect on the type. Nevertheless, the other, more type-informative, cases of *printf* can be inferred.

```
class Rep t where
  rep :: Format f ⇒ f t

instance Rep String where
  rep = eod

instance Rep r ⇒ Rep (Int → r) where
  rep = int rep

instance Rep r ⇒ Rep (String → r) where
  rep = str rep
```

We should note that these instance declarations are outside the scope of Haskell 98 — types are used where type variables should occur. However, this is a quite mild extension, and is supported by most Haskell compilers.

Making use of the fact that now we can infer some cases of the string format, we could define:

```
printPair :: Int → Int → String
printPair x y = printf rep "(" x ", " y ")"

printTrio :: Int → Int → Int → String
printTrio x y z = printf rep "(" x ", " y ", " z ")"
```

The function *printPair* does the same as before. However, with this new definition, the format directive is automatically inferred. The function *printTrio* is doing the same as *printPair*, except that it does it for triples. We should emphasise that the occurrences of *printf* in those two functions use different numbers of arguments. We should also mention that, in some situations, we will need to provide explicit types, otherwise the type checker would not be able to infer the correct instances of the type class *Rep*.

This use of *printf* seems to be practical, and for this simple version of it we might even argue that everything that we could do with a manually-provided parameter could be done with an automatically-inferred one. We simply do not need *lit* and *eol*, because those can be simulated using *str* (with, of course, extra *String* arguments). Nevertheless, if we decided to go for a more powerful version of *printf*, this might not be the case. Consider, for instance, the formatting directive `"%2d"`. In this case the number 2 is specifying the minimum width of the string that represents that number. If we wanted to allow this kind of behaviour, we could add an extra parameter of type *Int* to the *int* case. However, the problem now is to choose a value for that parameter when we automatically build the format directive. In this case we need to use some default value (for instance 1). However we are no longer able, for all possible cases, to simulate the functionality of *printf* with manual format strings using only automatically-built ones.

## 5. Type-indexed types

Until now we have been discussing *type-indexed functions*, that is, families of functions indexed by types. We turn now to *type-indexed types*, that is, families of types indexed by types. In the context of generic programming, we call these *generic types*. Generic functions with generic types are functions that have different result types for each structural case.

In this section, we will show how to implement type-indexed types as another variation of the *TypeCase* design pattern. We do this by translating a standard example of Generic Haskell [20], namely generic tries [17], into our approach.

### 5.1 Encoding type-indexed types

Section 3 presents templates for encoding type-indexed functions. In this section, we show how to translate a type-indexed type into an instance of the TypeCase design pattern.

In general, a type-indexed type has the form

$$\begin{aligned}
&\Gamma\langle t :: \kappa\rangle &&:: \tau\\
&\Gamma\langle t_1\ a_1\ \ldots\ a_i\rangle &&= \Lambda\, d_{11}\ \ldots\ d_{1n} \to \nu_1\\
&&&\ \vdots\\
&\Gamma\langle t_m\ z_1\ \ldots\ z_j\rangle &&= \Lambda\, d_{m1}\ \ldots\ d_{mn} \to \nu_m
\end{aligned}$$

where $\Gamma$ is the type-level function that defines the type-indexed type; $t$ is the family of types (or type constructors) $(t_1\ a_1\ \ldots\ a_i), \ldots, (t_m\ z_1\ \ldots\ z_j)$ of kind $\kappa$ that corresponds to the *structural cases* of the design pattern; and, finally, $\tau$ is the kind of $\Gamma\langle t :: \kappa\rangle$. For each type that is member of that family, we have a corresponding branch for $\Gamma$. The type-level lambda abstraction on the right side of each branch is optional, and corresponds to possible parametrically polymorphic variables $d_1\ \ldots\ d_n$ that the type-indexed type might depend on. Finally, $\nu_1\ \ldots\ \nu_m$ corresponds to the family of types (or type constructors) that defines the type-indexed type.

#### 5.1.1 Type class translation

We can now derive an instance of the TypeCase design pattern to capture type-indexed functions with type-indexed types. The *typecase* participant, for instances of the design pattern using either implicit or explicit representations, could be defined as follows.

**class** $\Gamma\ (g :: \kappa \to \tau' \to \star)$ **where**
  $case_{t_1} :: \Sigma_{(a_1\ \ldots\ a_i)} \rightsquigarrow g\ (t_1\ a_1\ \ldots\ a_i)\ d_{11}\ \ldots\ d_{1n}\ \nu_1$
  $\vdots$
  $case_{t_m} :: \Sigma_{(z_1\ \ldots\ z_j)} \rightsquigarrow g\ (t_m\ z_1\ \ldots\ z_j)\ d_{m1}\ \ldots\ d_{mn}\ \nu_m$

We reuse the name $\Gamma$ for the name of the type class that encodes the *typecase* component. The parameter $g$ is a type constructor with kind $\kappa \to \tau' \to \star$, where $\tau'$ is the literal occurrence of $\tau$ (if we were to use $\tau$ instead of its literal occurrence, we would obtain the wrong kind). There are $m$ functions $case_{t_1}, \ldots, case_{t_m}$ that correspond to the typecases for each type $(t_1\ a_1\ \ldots\ a_i), \ldots, (t_m\ z_1\ \ldots\ z_j)$. Each case of the typecase function is defined by providing the type constructor $g$ with the corresponding types. Finally, $\Sigma_{(a_1\ \ldots\ a_i)}, \ldots, \Sigma_{(z_1\ \ldots\ z_j)}$ corresponds to the representations for the types $(a_1\ \ldots\ a_i), \ldots, (z_1\ \ldots\ z_j)$.

The only difference between explicit and implicit versions of the design pattern for the typecase component is that in the explicit version the occurrences of $\Sigma$ are expanded into explicitly-passed representations of the form $g\ a_\kappa\ \ldots$, whereas with the implicit representations those occurrences are replaced by type class constraints of the form $Rep_\Gamma\ a_\kappa\ \ldots$.

The *dispatcher* can also be derived; but to do so requires extensions to Haskell 98 — specifically, multiple-parameter type classes with functional dependencies. The problem is that, even in its simplest form, a type-indexed type requires at least two type arguments: the first one corresponding to the index type, and the second one that is the resulting type-indexed type for that index, and thus depending on the index. This problem is not too serious if we use the explicit representations variant of the pattern, since the dispatcher is optional, but using implicit representations forces us outside Haskell 98.

**class** $Rep_\Gamma\ t\ d_1\ \ldots\ d_n\ \nu \mid t\ d_1\ \ldots\ d_n \to \nu$ **where**
  $rep :: \Gamma\ g \Rightarrow g\ t\ d_1\ \ldots\ d_n\ \nu$

**instance** $\Omega_{(a_1\ \ldots\ a_i)} \Rightarrow$
  $Rep_\Gamma\ (t_1\ a_1\ \ldots\ a_i)\ d_{11}\ \ldots\ d_{1n}\ \nu_1$ **where**
    $rep = case_{t_1}\{rep^i\}$
  $\vdots$

**instance** $\Omega_{(z_1\ \ldots\ z_j)} \Rightarrow$
  $Rep_\Gamma\ (t_m\ z_1\ \ldots\ z_j)\ d_{m1}\ \ldots\ d_{mn}\ \nu_m$ **where**
    $rep = case_{t_m}\{rep^j\}$

The type class $Rep_\Gamma$ has at least two type arguments: $t$ and $\nu$. If there are parametric types that $\nu$ depends on, then the type class also needs to account for those types $(d_1\ \ldots\ d_n)$. The class contains just one member function, *rep*, used to build representations for $\Gamma$. The function *rep* has a type class constraint ensuring that $g$ is an instance of $\Gamma$. There are, at least, $m$ instances of $Rep_\Gamma$, and those instances define *rep* with the corresponding $case_{t_\kappa}$ function. If we are implementing an implicit version of the design pattern, then the definition of *rep* is complete; otherwise, for an explicit version, we need to apply $case_{t_\kappa}$ to a number $i$ of *rep* functions (where $i$ is the number of type parameters of $t_\kappa$). The constraints $\Omega_{(a_1\ \ldots\ a_i)}, \ldots, \Omega_{(z_1\ \ldots\ z_j)}$ are very similar to the constraints $\Sigma$, and in fact for implicit representations they coincide: they correspond to representations for the types $a_1\ \ldots\ a_i, \ldots, z_1\ \ldots\ z_n$.

#### 5.1.2 Smart datatype translations

Encoding type-indexed functions with *smart datatypes* proceeds in a similar fashion to the encoding with type classes. We will demonstrate how to do this translation using a GADT syntax (as found in the new GHC 6.4 Haskell compiler).

A type-indexed type generates a smart datatype of the following form.

**data** $\Gamma\ t\ d_1\ \ldots\ d_n\ \nu$ **where**
  $c_{t_1} :: \Sigma_{(a_1\ \ldots\ a_i)} \rightsquigarrow \Gamma\ (t_1\ a_1\ \ldots\ a_i)\ d_{11}\ \ldots\ d_{1n}\ \nu_1$
  $\vdots$
  $c_{t_m} :: \Sigma_{(z_1\ \ldots\ z_j)} \rightsquigarrow \Gamma\ (t_m\ z_1\ \ldots\ z_j)\ d_{m1}\ \ldots\ d_{mn}\ \nu_m$

Instead of being parametrised by a "function" (like the type class approach), a smart datatype is parametrised by all the types on which it depends. Another difference from the type class approach is that the functions that represent each case are now replaced by constructors $c_{t_1}, \ldots, c_{t_m}$ that can just be pattern matched (in a dependent manner) by functions defined over those datatypes. A final difference is that $\Sigma_{(a_1\ \ldots\ a_i)}, \ldots, \Sigma_{(z_1\ \ldots\ z_j)}$ need to reflect the fact that we are now using a smart datatype.

The changes to $Rep_\Gamma$ are minimal; the only change to the type class version is that in the definition of *rep* we now use the constructors $c_{t_1}, \ldots, c_{t_m}$ instead of the functions $case_{t_1}, \ldots, case_{t_m}$.

**class** $Rep_\Gamma\ t\ d_1\ \ldots\ d_n\ \nu \mid t\ d_1\ \ldots\ d_n \to \nu$ **where**
  $rep :: \Gamma\ t\ d_1\ \ldots\ d_n\ \nu$

**instance** $\Omega_{(a_1\ \ldots\ a_i)} \Rightarrow$
  $Rep_\Gamma\ (t_1\ a_1\ \ldots\ a_i)\ d_{11}\ \ldots\ d_{1n}\ \nu_1$ **where**
    $rep = c_{t_1}\ rep^i$
  $\vdots$

**instance** $\Omega_{(z_1\ \ldots\ z_j)} \Rightarrow$
  $Rep_\Gamma\ (t_m\ z_1\ \ldots\ z_j)\ d_{m1}\ \ldots\ d_{mn}\ \nu_m$ **where**
    $rep = c_{t_m}\ rep^j$

### 5.2 Tries

*Tries* or *digital search trees* are a traditional example of a generic type. Tries make use of the structure of search keys in order to organise information, which can then be efficiently queried. In this section we will show how to implement generic tries using a variation of the LAGP type representations. For a more theoretical presentation of tries, see [20, 17]; the implementation of tries presented here follows closely the implementations found in those papers.

In [20], the generic type for tries is given as follows.

```
FMap⟨t :: ⋆⟩          :: ⋆ → ⋆
FMap⟨Unit⟩        v = Maybe v
FMap⟨Int⟩         v = MapInt v
FMap⟨Plus t1 t2⟩  v = OptPair (FMap⟨t1⟩ v) (FMap⟨t2⟩ v)
FMap⟨Prod t1 t2⟩  v = FMap⟨t1⟩ (FMap⟨t2⟩ v)
```

It is clear that the type-indexed function *FMap* takes a type parameter $t :: \star$ and another type of kind $\star$ and returns another type of kind $\star$. Only the shape of parameter $t$ is analysed; the other parameter $v$ needs to be used in the definition because the resulting type is parametrically polymorphic in relation to $v$.

We encode this characterisation of *FMap* as follows.

```
class FMap g where
  unit  :: g Unit v Maybe
  plus  :: g a v c → g b v d → g (Plus a b) v (PlusCase c d)
  prod  :: g a (d v) c → g b v d → g (Prod a b) v (ProdCase c d)
  data  :: g a v c → Iso b a → Iso (d v) (c v) → g b v d
  int   :: g Int v MapInt
```

This class forms the *typecase* participant of an explicit representation variant of the TypeCase pattern. The class *FMap* is a variation of the *Generic* class from Section 2.2.2. The functor $g :: \star \to \star \to (\star \to \star) \to \star$ takes the necessary information to rebuild the type-indexed type. The three parameters of the functor correspond, respectively, to the type parameter $t$, the second parameter and the resulting type of *FMap*. (The kind of the resulting type is now $\star \to \star$. We could have used kind $\star$ as in *FMap*, but we believe this version is slightly more readable.) The function *unit* just reflects the change of the functor $g$ and adds the information for the parametric type $v$ and the functor *Maybe* that is used to define the trie for the *Unit* case. The cases for *plus* and *prod* have explicit arguments that correspond to the recursive calls of the function; and the functors *PlusCase c d* and *ProdCase c d* correspond to the respective cases of the type-indexed type. The *data* function handles user-defined datatypes, having a recursive case and two isomorphisms: the first between the structural cases and a second between the tries corresponding to those cases. Finally, we could also define some extra base cases to handle primitive types such as *Int* and *Char*.

The auxiliary definitions for the newtypes *PlusCase a b v* and *ProdCase a b v* are defined as follows.

```
data OptPair a b = Null | Pair a b
newtype PlusCase a b v =
          PlusCase{unPlus :: OptPair (a v) (b v)}
newtype ProdCase a b v =
          ProdCase{unProd :: a (b v)}
```

The introduction of *OptPair a b* is for efficiency reasons [20].

In order to use a user-defined type (or a built-in type that does not have a special case for it), we need to do much the same work as for GM2 in Section 2.2.2. As an example, we show what to do for Haskell's built-in lists.

```
list :: FMap g ⇒ g a (FList c v) c → g [a] v (FList c)
list ra = data (plus unit (prod ra (list ra))) listEP
                (Iso unFList FList)
listEP :: Iso [a] (Plus Unit (Prod a [a]))
listEP = Iso fromList toList
  where
    fromList []       = Inl Unit
    fromList (x : xs) = Inr (Prod x xs)
    toList (Inl Unit) = []
    toList (Inr (Prod x xs)) = x : xs
newtype FList c v = FList{
  unFList :: (PlusCase Maybe (ProdCase c (FList c))) v}
```

The function *list* defines the encoding for the representation of *lists*. Because lists are a parametrised datatype with one type parameter, *list* is a function that takes one argument; this argument corresponds to the representation of the list type argument, and *list* returns the representation for lists. The definition is nearly the same as the equivalent for GM, but it takes an extra isomorphism describing the mapping between the structural representation of a list trie and a newtype *FList c v* that is introduced to represent the resulting list trie. The function *listEP* is just the isomorphism $[a] \cong 1 + a \times [a]$. This means that *listEP* can be shared with other versions of generics that use the same structural cases. However, *list* and *FList c v* still have to be introduced for each type-indexed datatype. Nevertheless, that is boilerplate code, and, with compiler support, it is should be possible to avoid writing it.

Having set up the main components of the design pattern, we can now move on to define our first function over tries. The function *empty* creates a new empty trie and can be defined as follows.

```
newtype EmptyTrie a v t = EmptyTrie{empty :: t v}
instance FMap EmptyTrie where
  unit          = EmptyTrie Nothing
  int           = EmptyTrie (MapInt [])
  plus ra rb    = EmptyTrie (PlusCase Null)
  prod ra rb    = EmptyTrie (ProdCase (empty ra))
  data ra iso iso2 = EmptyTrie (to iso2 (empty ra))
```

This function is very simple but, nonetheless, it has a type-indexed type: the *unit* case returns *Nothing*; the *int* case returns a value of a user-defined type for integer tries; the cases for *prod* and *plus* return, respectively, values for the previously defined *ProdCase* and *PlusCase* types; finally, the *data* returns a value of the newtype used to represent the trie of some user-defined datatype.

Another function that we will probably want to have in a library for tries is the *lookUp* function which, given a key, returns the corresponding value stored in the trie.

```
newtype LUp a v t = LUp{lookUp :: a → t v → Maybe v}
instance FMap LUp where
  unit          = LUp (λ_ fm → fm)
  int           = LUp (λi fm → lookUpInt i fm)
  plus ra rb    = LUp (λt fm →
    case (unPlus fm) of
      Null          → Nothing
      (Pair fma fmb) → case t of
                         (Inl l) → lookUp ra l fma
                         (Inr r) → lookUp rb r fmb)
  prod ra rb    = LUp (λt (ProdCase fma) →
    case t of
      (Prod x y) → (lookUp ra x ⋄ lookUp rb y) fma)
  data ra iso iso2 =
    LUp (λt r → lookUp ra (from iso t) (from iso2 r))
```

(The operator $\diamond$ represents monadic composition.) The functions *empty* and *lookUp* have definitions that only have generic function calls to themselves. However, that is not the case for all generic functions. One such function is the generic function that creates a trie containing a single element; a possible definition makes use of the generic function *empty*. We discussed in Section 3 that, using an explicit version of the design pattern, there are some issues with generic functions calling generic functions other than themselves. One solution for this problem is using tupling. Just as one does with a type class, we would choose a fixed set of functions and group them together in a record. For instance, in the case of tries, we could have the following.

```
data Tries a v t = Tries{
  empty   :: t v,
  isempty :: t v → Bool,
  single  :: a → v → t v,
  lookup  :: a → t v → Maybe v,
  insert  :: (v → v → v) → a → v → t v → t v,
  merge   :: (v → v → v) → t v → t v → t v,
  delete  :: a → t v → t v}
```

With our definition we could, for any function in the record, make mutual generic calls.

Whilst we could have used a multiple-parameter type class with functional dependencies in order to implement this library of functions over tries, there would be one important disadvantage in doing so (apart from the fact that we need to leave Haskell 98): we can only have functions on types of kind $\star$. With type classes, contexts are implicitly passed, and there is no way to redefine those implicit behaviours. In other words, type classes have the same limitation as implicit representations as a version of the TypeCase design pattern, in that they can only work on types. On the other hand, derived from the fact that we use external representations, with this implementation we can define generic functions over type constructors.

Tupling is not the only option to solve the problem of generic function calls. Another possibility is to have the notion of *dependencies*: instead of tupling all functions together, we can, for each generic function that we need to use, include one instance of that function. Here is a possible definition of *single* using this strategy.

```
data Single a v t = Single{
  emptyT :: EmptyTrie a v t,
  single  :: a → v → t v}
instance FMap Single where
  unit        = Single unit (λ_ v → Just v)
  int         = Single int (λi v → MapInt [(i,v)])
  plus ra rb  = Single (plus (emptyT ra) (emptyT rb))
    (λi v →
      case i of
        Inl l → PlusCase (Pair  (single ra l v)
                                (empty (emptyT rb)))
        Inr r → PlusCase (Pair (empty (emptyT ra))
                               (single rb r v)))
  prod ra rb = Single (prod (emptyT ra) (emptyT rb))
    (λi v →
      case i of
        Prod x y → ProdCase (single ra x (single rb y v)))
  data ra iso iso2 = Single (data (emptyT ra) iso iso2)
    (λi v → to iso2 (single ra (from iso i) v))
```

The idea of dependencies is motivated by Dependency-Style Generic Haskell [26, 27]. In this version of Generic Haskell, the type system reflects the uses of generic functions in the definitions by keeping track of constraints that identify such uses. With this definition, we have to manually introduce those dependencies by adding extra fields to the record that keep track of all the functions on which the definition depends. That change is also reflected in the instance that defines the generic function, where we need to provide values for the extra fields; the values for those fields just reconstruct the dependent functions with their values for those fields.

## 6. Discussion and conclusions

The goal of design patterns is not to come up with a miraculous solution for a problem. Instead, design patterns capture good techniques that appear in the literature or in practice, in a variety of contexts, and document them to make them easier to identify and implement. In this paper we have generalised the technique found in LIGD and GM to a design pattern, and presented a number of applications of the pattern. Furthermore, we have identified other occurrences of the design pattern in the literature.

### 6.1 Related work

The technique used by Danvy [10] and generalised by Yang [37] allows us to encode *type-indexed values* in a Hindley-Milner type system. This encoding is directly related to the explicit representation version of the TypeCase pattern. This technique influenced many other works, ranging from type-directed partial evaluation [37, 9, 12], through embedded interpreters [2], to a generalisation of families of functions like *zipWith* [13] — these are all possible applications of the TypeCase design pattern. Our paper revises that technique and shows how slightly richer type systems can be used to improve it. In particular, the use of a *dispatcher* makes it possible to automatically built the values encoding types. Moreover, the issue of *composability* (identified by Yang), while still a problem, can benefit from stronger type systems: the use of rank-two types combined with type classes provides a good solution.

The work on *extensional polymorphism* [11] presents an approach that allows functions to implicitly bind the types of their arguments in a modified version of ML. Furthermore, using a *typecase* construct it is possible to support generic programming. Harper and Morrisett's work on *intensional type analysis* [16] presents an intermediate language where run-time type analysis is permitted, using *typecase* and *Typecase* constructs to define type-indexed functions and type-indexed types, respectively. However, approaches based on run-time type analysis have important drawbacks; for instance, they cannot support abstract datatypes, and they do not respect the parametricity theorem [35, 33]. Subsequent approaches to intensional type analysis by Crary and others [7, 6] use a type-erasure semantics that does not suffer from those problems. Still, those approaches were limited to first-order type analysis. More recently, Weirich [36] proposed a version of intensional type analysis covering higher-order types with a type-erasure semantics. Furthermore, she presented an implementation in Haskell (augmented with rank-two types). This work inspired Hinze's implementation of GM, which shows, in essence, how to avoid rank-two types by using Haskell's class system. Our work makes use of those results and explains how to simulate *typecase* constructs. Furthermore, we show that the limitation of GM that generic functions with generic types cannot be defined can be lifted with our more general interpretation.

Generic programming (or perhaps datatype-generic programming [15]) is about defining functions and types that depend on the structure of other types. One of the first attempts to produce a generic programming language was PolyP [21]. This language allowed the definition of generic functions over regular datatypes with one type parameter. In Section 4.1 we show that, using our design pattern, it is possible to define PolyP-like generic functions just using Haskell 98. A previous attempt [30] to define first-class PolyP functions in Haskell required extensions to the language. The Generic Haskell [26, 5] project is more ambitious than PolyP, and aims at defining generic functions for nearly all types definable in Haskell 98. Furthermore, Generic Haskell features generic types and generic function customisation (which were not present in PolyP). Dependency-Style Generic Haskell [26, 27] introduces a rather complex type system that keeps track of dependencies on generic function calls. The need for this sophisticated type system is a consequence of a model for generic programming that allows generic function customisation. The approach presented in [24] is another kind of lightweight approach to generic programming, relying on a run-time type-safe cast operator. With that operator it is possible to define a number of traversals that allow a very interesting model of generic programming based on nominal typing. Our design pattern can be used to encode many of the generic definitions that these generic programming techniques allow. However, it can be less practical than approaches providing a special-purpose compiler. Nevertheless, the advantage of our technique is that we do not need to commit in advance to a model of generic programming: we have the freedom to choose our own model of generic programming.

Design patterns in the object-oriented programming community have been given a great deal of attention. Whilst amongst the

functional programming community there has been some work on — or, at least, involving the concept of — design patterns [24, 23, 25], the concept is still much less popular than in the object-oriented community. Moreover, most of this work presents patterns that are really more like algorithmic patterns rather than design patterns. Perhaps the reason why this happens is that functional languages are very expressive, and often natural features of those languages, such as laziness or higher-order functions, can be used to remove the need for complex designs. Nevertheless, we believe that our design pattern is more related to the OO concept of a design pattern with type classes/datatypes taking the role of OO interfaces and class instances taking the role of OO concrete classes. One difficulty found in this work had to do with the fact that, unlike OO design patterns which are documented using informal notations such as UML, we do not have a notation to "talk" about the design of Haskell programs. The notation that we used is quite ad-hoc and it can be difficult to read.

### 6.2 Future work

We mentioned that this design pattern seems to be very similar to OO design patterns. It would be interesting to explore the applicability of this design pattern in an OO environment.

Design patterns are useful to overcome the lack of certain features in programming languages. In our case, we overcome the lack of a *typecase* construct. The work on intensional type analysis investigates the possibility of languages supporting *typecase* constructs directly in the language. Combining these results in order to extend Haskell with a more natural support for *typecase* programming is something we would like to try in the future.

Problems that use multiple instances of the design pattern are not composable. For instance, in a generic programming context, we could have a class *Generic* that allowed us to define generic functions with one type parameter; and we could also have a class *FMap* for working with tries. Although, those classes are structured in a similar way, they require two distinct representations of types, one for each of the classes; we hope to address this impracticality.

#### Acknowledgements

We would like to thank Ralf Hinze for the discussion that inspired this paper. Stefan Holdermans, the anonymous referees and the members of the *Algebra of Programming* group at Oxford and the EPSRC-funded *Datatype-Generic Programming* project made a number of helpful suggestions.

## References

[1] C. Alexander. *A Pattern Language*. Oxford University Press, 1977.

[2] N. Benton. Embedded interpreters. Microsoft Research, Cambridge, Jan. 2005.

[3] R. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[4] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell Workshop*, pages 90–104, 2002.

[5] D. Clarke and A. Löh. Generic Haskell, specifically. In *Generic Programming*, pages 21–47. Kluwer, B.V., 2003.

[6] K. Crary and S. Weirich. Flexible type analysis. In *International Conference on Functional Programming*, pages 233–248, 1999.

[7] K. Crary, S. Weirich, and J. G. Morrisett. Intensional polymorphism in type-erasure semantics. In *International Conference on Functional Programming*, pages 301–312, 1998.

[8] N. A. Danielsson and P. Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In D. Kozen, editor, *LNCS 3125: Mathematics of Program Construction*, pages 85–109. Springer-Verlag, 2004.

[9] O. Danvy. Type-directed partial evaluation. In *Principles of Programming Languages*, 1996.

[10] O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

[11] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Principles of Programming Languages*, pages 118–129, 1995.

[12] P. Dybjer and A. Filinski. Normalization and partial evaluation. In *LNCS 2395: Applied Semantics*, pages 137–192. Springer, 2002.

[13] D. Fridlender and M. Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, 2000.

[14] J. Gibbons. Calculating functional programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 149–202, 2000.

[15] J. Gibbons. Patterns in datatype-generic programming. In *Declarative Programming in the Context of Object-Oriented Languages*, 2003.

[16] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Principles of Programming Languages*, pages 130–141, San Francisco, California, 1995.

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

[18] R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave, 2003.

[19] R. Hinze. Generics for the masses. In *International Conference on Functional Programming*, pages 236–243. ACM Press, 2004.

[20] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, 2004.

[21] P. Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology, May 2000.

[22] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *LNCS 1129: Advanced Functional Programming*, pages 68–114. Springer-Verlag, 1996.

[23] T. Kühne. *A Functional Pattern System for Object-Oriented Design*. Verlag Dr. Kovač, ISBN 3-86064-770-9, Hamburg, Germany, 1999.

[24] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types in Language Design and Implementation*, 2003.

[25] R. Lämmel and J. Visser. Design patterns for functional strategic programming. In *Workshop on Rule-Based Programming*, 2002.

[26] A. Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

[27] A. Löh, D. Clarke, and J. Jeuring. Dependency-style Generic Haskell. In *International Conference on Functional Programming*, pages 141–152, 2003.

[28] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *LNCS 523: Functional Programming Languages and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.

[29] D. Menendez. Fixed-length vectors in Haskell. `http://www.haskell.org/pipermail/haskell/2005-May/015815.html`.

[30] U. Norell and P. Jansson. Polytypic programming in Haskell. In *Implementing Functional Languages*, 2003.

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

[32] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: Type inference for generalised algebraic data types. Microsoft Research, Cambridge, 2004.

[33] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier, 1983.

[34] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, pages 307–313. ACM Press, 1987.

[35] P. Wadler. Theorems for free! In *Functional Programming and Computer Architecture*, 1989.

[36] S. Weirich. Higher-order intensional type analysis in type-erasure semantics. `http://www.cis.upenn.edu/~sweirich/papers/erasure/erasure-paper-july03.pdf`, 2003.

[37] Z. Yang. Encoding types in ML-like languages. In *International Conference on Functional Programming*, pages 289–300, 1998.