

Bialgebra Views

A Way for Polytypic Programming to Cohabit with Data Abstraction

Pablo Nogueira Juan José Moreno-Navarro

Babel Research Group, Universidad Politécnica de Madrid, Spain

{pablo,jjmoreno}@babel.ls.fi.upm.es

Abstract

Polytypic programming and data abstraction are important concepts in designing functional programs, but they do not fit well together. Polytypic programming currently depends on making public a free data type representation, while data abstraction depends on hiding the representation. This paper proposes the *bialgebra views* mechanism as a means of reconciling this conflict. Bialgebra views enable the specification of type structure according to interfaces, not representations, thus combining the genericity of polytypic programming with the information hiding of data abstraction, and narrowing the gap between generic programming in the functional and object-oriented paradigms.

Categories and Subject Descriptors D.1.1 [Language Classifications]: Applicative (functional) Languages; D.1.m [Programming Techniques]: Generic Programming.

General Terms Languages, Theory, Design.

Keywords Polytypic programming, abstract types, bialgebras, program generation.

1. Introduction

Generic programming increases the flexibility of strongly and statically type-checked programming languages by expanding their possibilities for parametrisation, instantiation, and encapsulation. Parametrisation is about ‘allowing a wider variety of entities as parameters than is available in more traditional programming languages’ [4]. Implicit in parametrisation is the dual notion of instantiation which describes the mechanics of parameter passing. Finally, encapsulation is information hiding with respect to component (function, data, etc) internals. A specification describes the component’s behaviour irrespective of its implementation. A consequence of encapsulation is the interchangeability of components that satisfy the specification [16].

In object-oriented languages, generic programming is identified with libraries of algorithms on first-class, first-order, parametrically polymorphic abstract data types. A conspicuous representative is the C++ Standard Template Library [22]. Abstract types (*container* classes) have minimal interfaces. Generic algorithms rely on *iterator* classes which abstract over container structure and enable their linear traversal.

In functional languages, generic programming is identified with *polytypic* programming [11]. Polytypic functions are parametric on types, like parametrically polymorphic functions. However, instantiation exploits the type’s structure and the function’s behaviour is not uniform on its type parameter. More concretely, polytypic functions capture families of functions in a single definition. Examples are calculating the size, map, equality, encoding and decoding, pretty-printing, etc, which are defined once for all types.¹ A technique for obtaining the family member for a specific type consists in specialising at compile time the polytypic function for that type (Section 2).

There are functional languages or language extensions supporting polytypic programming, e.g., Generic Haskell [7, 17], Clean [1], Functorial ML [10], and Scrap Your Boilerplate [13], the latter a blend of polytypic and strategic programming techniques [15]. They all have in common that the notion of data type supported is that of a free algebra whose structure is given by its representation.

In contrast, *abstract data types* (ADTs) are seldom free algebras and their representation is immaterial. More precisely, ADTs are not defined by a set of value constructors but by a set of functions and conditional equations on them, which constitute the ADT’s *specification* [19]. The ADT’s implementation is encapsulated behind the specification. The names and type signatures of functions constitute the ADT’s *interface*. The set of equations describes the ADT’s *axiomatic semantics*. There are habitually equations among constructor functions (non-freeness). ADTs are important in large applications, for they simultaneously afford representation independence and a principled means of dealing with non-freeness.

None of the aforementioned languages support polytypic programming with ADTs and hence fall short of data encapsulation. This perhaps explains why recent attempts to transfer polytypism to mainstream object-oriented languages rely on literal encodings of algebraic types, e.g. [20, 21]. This is unrealistic for two reasons. First, data abstraction is a *sine qua non* in object-oriented languages. Data is often abstract, e.g., stacks, queues, drawing shapes, buttons, etc. Second, defining algebraic types such as lists as abstract classes with public subclasses encoding value constructors is unsatisfactory. From an object-oriented perspective, lists are also encapsulated abstractions whose interface provides a contract and whose implementation is immaterial and may be subject to change. The way to narrow the gap between generic programming in the functional and object-oriented paradigms is to make polytypic programming support abstract types.

Our contributions are threefold. First, we elaborate why polytypic programming currently conflicts with data abstraction. In particular, we explain why the categorical model of data types assumed by polytypic programming [9] is inappropriate for ADTs. We also

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WGP’08, September 20, 2008, Victoria, BC, Canada.
Copyright © 2008 ACM 978-1-60558-060-9/08/09...\$5.00

¹ Polytypic map and polytypic equality are defined once, but they are only applicable to functor and non-function types respectively.

discuss why modular extension and the adoption of pattern matching techniques for ADTs are not solutions. Second, we propose *bialgebra views* as a means of specifying type structure according to interfaces, not representations. We define polytypic functions on ADTs that are parametric on the structure provided by bialgebra views and on a set of operator names that instantiate (adapt) the views. Bialgebra views assume a bialgebra model of types [3, 24] which we also explain. Third, we detail how to implement bialgebra views and polytypic functions on ADTs in Generic Haskell extended with language features for supporting their declarations. One ingredient of the extension is support for *context-parametric polykinded types* [23] which are explained in the appendix to make the paper self-contained. We have chosen Generic Haskell because it supports polytypic programming on algebraic types of arbitrary kind. However, the bialgebra view idea is independent of Generic Haskell and can be instrumented in other polytypic languages. The implementation mechanisms suggested are not the only ones possible. We are mostly interested in demonstrating implementability and not in showing a definitive implementation.

Terminology: In conformity with the ‘functions are values’ principle, we refer to type constructors (i.e., type-level functions) such as lists, trees, etc, as *data types* or, simply, as *types*. We refer to (free) algebraic data types as *concrete types* to underline the contrast with *abstract types*.

2. Polytypic Programming in Generic Haskell

Polytypic programming hinges on the fact that data structure often determines program structure and therefore it makes sense to abstract over the former while exploiting it to drive program behaviour. Let us elaborate this idea using the Generic Haskell language as a vehicle² and the concrete types of lists and binary trees as running examples:

```
data List a = Nil | Cons a (List a)
data Tree a = Empty | Node a (Tree a) (Tree a)
```

The Generic Haskell compiler represents the top-level structure of these types in so-called ‘structure representation types’ which are, by default, nested, right-deep binary sums of binary products:

```
data Unit = Unit
data a + b = Inl a | Inr b -- sum
type a × b = (a, b) -- product
type List◦ a = Unit + (a × (List a))
type Tree◦ a = Unit + (a × ((Tree a) × (Tree a)))
```

Types *Unit*, *+*, and type synonym *×* are predefined. *List[◦]* and *Tree[◦]* are respectively the structure representation types of *List* and *Tree*. The Generic Haskell compiler also generates translation functions automatically (we only show the ones for lists):

```
fromList :: List a → List◦ a
fromList Nil = Inl Unit
fromList (Cons x xs) = Inr (x, xs)

toList :: List◦ a → List a
toList (Inl Unit) = Nil
toList (Inr (x, xs)) = Cons x xs
```

Translation function *fromList* is an embedding whereas *toList* is a projection. Formally, all translation functions must satisfy *to ◦ from = id* and *from ◦ to ⊆ id* [7].

Structure representation types provide a common, universal representation for concrete types. Polytypic (that is, generic) functions

² We use classic-style Generic Haskell [7] because we believe it makes polytypism easier to grasp and because it is used internally by the Generic Haskell compiler [17, p104].

```
type Map{*} t1 t2 = t1 → t2
type Map{k → v} t1 t2 =
  ∀ a1 a2. Map{k} a1 a2 → Map{v} (t1 a1) (t2 a2)

gmap{t :: k} :: Map{k} t t
gmap{Int} = id
gmap{Bool} = id
gmap{Unit} = id
gmap{+} f g (Inl x) = Inl (f x)
gmap{+} f g (Inr y) = Inr (g y)
gmap{×} f g (x, y) = (f x, g y)
```

Figure 1. Polykinded type *Map* and polytypic function *gmap*.

are defined on structure representation types. More concretely, by cases on base types, *Unit*, *+*, and *×*. (Function space is also allowed but we omit it in this presentation.) Figure 1 shows the example of *gmap*, the polytypic version of the *map* function for lists. Unlike ordinary *map*, polytypic *gmap* can work not only with lists but with all (functor) concrete types in the program for which structure representation types and translation functions have been generated. Exceptions are existential and constrained types (Appendix A) which are currently not supported. Type parametrisation is explicit, with type parameters written between special braces *{·}*. In words, *gmap* for *Unit* and base types *Int* and *Bool* is the identity,³ whereas for sums and products it applies argument functions *f* and *g* appropriately. Each case in the definition is specialised by the compiler to an ordinary function. For example, cases *gmap{Int}* and *gmap{×}* specialise to:

```
gmapInt = id
gmapProd f g (x, y) = (f x, g y)
```

The type signature of *gmap* is given by *polykinded type Map*, that is, a type parametric on a *kind* and on one or more type arguments of that kind. Kind parametrisation is explicit, with kind parameters written between special braces *{·}*. The type is called polykinded because it is defined by induction on its kind argument [6]. (We recall that Haskell kinds capture the arity and order of types. They conform to the grammar *k, v ::= * | k → v* where *** is the kind of non-parametric types and *k → v* is the kind of parametric types taking types of kind *k* to types of kind *v*. For example, *Bool* and *List Int* have kind *** whereas *List* and *Tree* have kind ** → **.)

The application of a polytypic function *g* to a type *T* is written *g{ T }* and referred to as *polytypic application*. Broadly, the compiler replaces the polytypic application by a call to the statically-generated specialisation of *g* for *T*. The structure of the specialised function follows almost to the letter the structure of *T*’s structure representation type. For example, each occurrence in the program of the polytypic application *gmap{List}* is replaced by a call to its specialisation *gmapList* whose definition is generated by the compiler:

```
gmapList :: ∀ a1 a2. (a1 → a2) → List a1 → List a2
gmapList f = lift (gmapPlus gmapUnit (gmapProd f (gmapList f)))
```

Its type signature is obtained from the instantiation and expansion of the polykinded type:

```
Map{* → *} List List
=
  ∀ a1 a2. Map{*} a1 a2 → Map{*} (List a1) (List a2)
=
  ∀ a1 a2. (a1 → a2) → Map{*} (List a1) (List a2)
```

³ Cases for other base types are omitted for reasons of space.

```

type Size $\{\{*\}\}$  t = t → Int
type Size $\{\{k \rightarrow v\}\}$  t = ∀a. Size $\{\{k\}\}$  a → Size $\{\{v\}\}$  (t a)
gsize $\{\{t :: k\}\}$  :: Size $\{\{k\}\}$  t
gsize $\{\{Int\}\}$  = const 0
gsize $\{\{Bool\}\}$  = const 0
gsize $\{\{Unit\}\}$  = const 0
gsize $\{\{+\}\}$  f g (Inl x) = f x
gsize $\{\{+\}\}$  f g (Inr y) = g y
gsize $\{\{\times\}\}$  f g (x, y) = f x + g y

```

Figure 2. Polykinded *Size* and polytypic *gsize*.

=
 $\forall a_1 a_2. (a_1 \rightarrow a_2) \rightarrow (List\ a_1 \rightarrow List\ a_2)$

Function *lift* computes the map for *List* given the map for *List*^o. It is implemented in terms of a built-in map function for a record type of translation-function pairs [7]. Its details are technical and we omit them for reasons of space.

Figure 2 shows another polytypic function example: *gsize* which computes data type sizes. By convention, base types and *Unit* have no size. The size of sums is computed by applying the appropriate argument function that delivers the size of the component, and the size of products adds the results.

Some examples of usage: *gmap* $\{\{List\}\}$ \neg $[True, False]$ delivers $[False, True]$, *gsize* $\{\{List\}\}$ $(const\ 1)$ $['A', 'B', 'C']$ delivers 3, and *gsize* $\{\{Tree\}\}$ $(const\ 1)$ $(Node\ 7\ Empty\ Empty)$ delivers 1. In the last two, the size of data elements is provided by the constant-function argument.

Traditionally, programmers define functions by induction on the structure of values (pattern matching). In contrast, polytypic programmers define polykinded types by induction on the structure of kinds and polytypic functions by induction on the structure of concrete types—or more precisely, by cases on the building blocks of structure representation types (base types, units, and binary sums and products). Type recursion, type abstraction, type application, and translation from/to structure representation types are taken care automatically by the Generic Haskell compiler [7].

3. The Data Abstraction Conflict

Polytypic programming currently conflicts with data abstraction because the structure of types is obtained from their *concrete* representation. The ‘adaptability of polytypic functions to changing data types’ [9, p5] is no longer a benefit but a hazard: data abstraction has the opposite aim. Indeed, the type(s) implementing an ADT are logically hidden, even physically unavailable in pre-compiled libraries, in order to encapsulate representation changes. Letting polytypic functions access representations wrecks the same havoc as letting non-polytypic ones: results may change with representation changes, may be inaccurate, and may violate implementation invariants.

Consider, for example, FIFO queues whose type-class interface is shown in Figure 3. In the ‘batched’ implementation, two lists are employed for front and rear whereas in the ‘physicists’ implementation a third list is also employed which is a cached prefix of the front list [25]. In the latter implementation, *gsize* counts the prefix list as part of the size, delivering a different (and inaccurate) result than in the ‘batched’ implementation. Consider now ordered sets (interface in Figure 3) implemented as lists. Assume the implementation invariant is ‘the list must be sorted in increasing order and have no duplicates’. Below, ϕ is a ‘repair’ function that re-establishes the invariant and standard function *nub* removes duplicates from a list.

```

class Queue q where
  empty  :: q a
  isEmpty :: q a → Bool
  enq    :: a → q a → q a
  front  :: q a → a
  deq    :: q a → q a

class OrdSet s where
  empty  :: Ord a ⇒ s a
  isEmpty :: Ord a ⇒ s a → Bool
  insert :: Ord a ⇒ a → s a → s a
  min    :: Ord a ⇒ s a → a
  remove :: Ord a ⇒ a → s a → s a
  member :: Ord a ⇒ a → s a → Bool

class Date d where
  mkDate  :: Day → Month → Year → d
  getDay  :: d → Day
  getMonth :: d → Month
  getYear  :: d → Year
  tomorrow :: d → d
  ...

```

Figure 3. FIFO-queue, ordered-set, and date interfaces.

```

data Ord a ⇒ OrdSetList a = S [a]
instance OrdSet OrdSetList where
  empty = S []
  insert x (S xs) = S ( $\phi$  (x : xs))
  where  $\phi$  = sort ∘ nub
  ...
  s :: OrdSetList Int
  s = insert 2 (insert 1 empty)

```

Internally, *s* is represented by $S [1, 2]$. The polytypic application *gmap* $\{\{OrdSetList\}\}$ *negate* *s* would map *negate* over the contents of the list, delivering the value $S [-1, -2]$ which does not represent an increasingly-ordered set. Also, *gmap* $\{\{OrdSetList\}\}$ $(const\ 0)$ *s* would deliver $S [0, 0]$, introducing duplicates. In both cases the implementation invariant has been violated because ϕ has not been applied (see [24] for a characterisation of ϕ and ADT functoriality).

Another part of the conflict is the data access problem affecting non-parametric ADTs. Consider a date type (Figure 3). An instance of the class (date implementation) has kind *. In general, polytypic functions on kind-* types deliver boring results. For instance, *gsize* would deliver 0 and *gmap* would be the identity. It is more convenient and general to have a *gmap* for non-parametric ADTs which is nevertheless parametric on mappings for data elements, and a *gsize* that is parametric on functions that compute the size of data elements, and so on, with specialisations looking somewhat like this:

$$\begin{aligned}
 gsizeDate :: Date\ d \Rightarrow (Day \rightarrow Int) \rightarrow (Month \rightarrow Int) \rightarrow \\
 (Year \rightarrow Int) \rightarrow d \rightarrow Int \\
 gsizeDate\ sd\ sm\ sy\ d = (sd\ (getDay\ d)) + (sm\ (getMonth\ d)) \\
 + (sy\ (getYear\ d))
 \end{aligned}$$

Unfortunately, we cannot redefine kind-* ADTs as parametric for physical (pre-compiled) or logical (*Date a b c* is nonsense) reasons. An alternative mechanism for declaring and abstracting over the types of data elements is needed.

3.1 Non-solutions

3.1.1 Polytypic-function Extension

Most languages supporting polytypic programming also support the modular extension of polytypic functions, e.g. [14, 17]. Using such *polytypic-function extension*, a term borrowed from [14], it is possible to replace or refine the behaviour of polytypic functions at *specific* types. In particular, it is possible to extend polytypic functions with cases for specific ADTs. However, this is not polytypic programming but overloading. In the former a single function (name and body) is defined for all types whereas in the latter a new function body must be provided for every type.

3.1.2 Pattern Matching

Pattern matching is another language feature that conflicts with data abstraction. Several proposals for reconciling the two have been put forth, e.g. [28, 29, 27]. The following are some of the lessons learnt. First, it must be possible for computation to take place at matching time in order to access data. Second, ADTs are not free algebras, construction and observation are not inverses and must be separated. Patterns should be used only for observation, relying on ordinary interface functions for construction. These two points remind us that pattern matching is logically syntactic sugar for observation. We might contrive elaborated pattern-matching mechanisms that let us observe ADTs as sums of products which can then be used by polytypic functions. But this comes with a cost (changes to the type system, possible undecidabilities, etc [27]), and there is the trouble of having to use interface functions in construction.

3.1.3 Generic Views

Generic views [8] is a recently supported Generic Haskell extension that lets programmers pick the encoding into structure representation types from a predefined set of choices (e.g., left-deep or balanced sums of products, etc). However, generic views do not provide a satisfactory solution to polytypic programming with ADTs because Generic Haskell still needs to access representations to generate structure representation types and specialise polytypic functions.

Nevertheless, let us suppose that an ADT provides itself the things needed for the Generic Haskell compiler, i.e., structure representation types and translation functions. The latter's invertibility requirements (Section 2) make the approach impractical. For example, suppose FIFO queues come equipped with the following structure representation type and translation functions:

```

data Queue q ⇒ Queueo q a = Qo (Unit + (a × (q a)))
fromQueue :: Queue q ⇒ q a → Queueo q a
fromQueue q = if isEmpty q then Qo (Inl Unit)
               else Qo (Inr (front q, deq q))

toQueue :: Queue q ⇒ Queueo q a → q a
toQueue (Qo (Inl Unit)) = empty
toQueue (Qo (Inr (x, q))) = reverseQ (enq x (reverseQ q))
where reverseQ = ...

```

Save for the abstraction over variable q , the structure representation type is identical to that of lists. Recall from Section 2 that $toQueue$ must be the left inverse of $fromQueue$. There is no reasonably efficient way of writing $toQueue$ in terms of queue operators and those definable from them, such as $reverseQ$ which reverses the queue and is applied twice in order to invert $fromQueue$. The problem is that $front$ and enq work at opposite ends of the queue, only an inverse of enq would do the trick, but the interface offers no such operator. The inefficiency is aggravated by the fact that translation functions are called at every recursive point [7].

4. Categorical Model of Types

The underlying problem has been pinpointed in the last section: ADTs are not free, construction and observation may not be inverses. Consequently, $from \circ to \subseteq id$ is too strong and $to \circ from = id$ cannot be upheld reasonably efficiently. Formally, the assumed categorical model of types [9, p146] is inappropriate for ADTs. We elaborate this point in the present section.

The assumed category is **CPO**, where objects are complete partial orders (cpo) giving semantics to Haskell monomorphic types ($Int, Bool, [Int]$, etc), and arrows are Scott-continuous functions on cpo giving semantics to Haskell functions on those types. We shall be informal and refer, for example, to objects of **CPO** as types. We use a, b, c , and x for arbitrary types and adhere to Haskell notation in the following categorical definitions, some of which we borrow and adapt from [18]. We only deviate from Haskell in writing some type signatures uncurried and 1 for the unit type. We explain some arrows of **CPO** by showing the Haskell function to which they give semantics, saving the space required to explain the behaviour on \perp .

Products are objects $a \times b$ together with arrows $fst :: a \times b \rightarrow a$ and $snd :: a \times b \rightarrow b$. Coproducts (disjoint sums) are objects $a + b$ together with arrows $Inl :: a \rightarrow a + b$ and $Inr :: b \rightarrow a + b$. We gave Haskell definitions for types \times and $+$ in Section 2. Products and coproducts have unique mediating arrows $(f \Delta g) :: c \rightarrow a \times b$ and $(f \nabla g) :: a + b \rightarrow c$. We show their Haskell definitions below:

$$\begin{aligned}
 (\nabla) &:: (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow a + b \rightarrow c \\
 (f \nabla g) (Inl x) &= f x \\
 (f \nabla g) (Inr y) &= g y \\
 (\Delta) &:: (c \rightarrow a) \rightarrow (c \rightarrow b) \rightarrow c \rightarrow a \times b \\
 (f \Delta g) x &= (f x, g x)
 \end{aligned}$$

Given a category \mathbf{C} , an F -algebra for a functor $F :: \mathbf{C} \rightarrow \mathbf{C}$ is a pair (x, α) where x is an object and $\alpha :: Fx \rightarrow x$ is an arrow. Dually, an F -coalgebra is a pair (x, β) where $\beta :: x \rightarrow Fx$ is an arrow. The triple (x, α, β) is called an F -bialgebra. We follow standard practice and write α, β, γ , etc, for (co)algebra arrows. In our category, x is a type (the algebra's carrier) with value constructors (the algebra's operators) in α , and F is the functor defined by the disjoint sum of the argument types (products) of value constructors. In Haskell, the type x is defined as the least fixed point of F and is an *initial* F -algebra. An initial F -algebra exists when F is *polynomial*, i.e., defined by composing identity, constant, sum, and product functors [2]. For example, Haskell lists are an initial F -algebra. They are defined by the equation $[a] = F [a]$, i.e., as the least fixed point of the functor $Fx = 1 + a \times x$ defined from the sum of argument types of value constructors $[] :: 1 \rightarrow [a]$ and $(:) :: a \times [a] \rightarrow [a]$. Thus, $\alpha :: F [a] \rightarrow [a]$ and $\alpha = [] \nabla (:)$. The *Tree* type of Section 2 is an initial F -algebra where now $Fx = 1 + a \times x \times x$, $\alpha :: F (Tree a) \rightarrow Tree a$, $\alpha = Empty \nabla Node$, $Empty :: 1 \rightarrow Tree a$, and $Node :: a \times (Tree a) \times (Tree a) \rightarrow Tree a$.

There may be several initial F -algebras for a given F , but they are unique up to isomorphism. This means that all observations on concrete types can be adequately captured by defining the types as initial F -algebras. Formally, there is an F -coalgebra (x, α^{op}) characterising observation such that $\alpha^{op} :: x \rightarrow Fx$ is an inverse of α . More precisely, $\alpha \circ \alpha^{op} = id_x$ and $\alpha^{op} \circ \alpha = id_{Fx}$. For example, in the case of lists, $\alpha = (k_1 + (head \Delta tail)) \circ null?$ where k_1 is the arrow part of the constant unit functor (in Haskell, $k_1 = const ()$), $(+)$ is the arrow part of the co-product functor (in Haskell, the map function for sums), and $(?)$ delivers a sum from the value of a *discriminator* predicate, in Haskell:

$$\begin{aligned}
 (?) &:: (a \rightarrow Bool) \rightarrow a \rightarrow a + a \\
 p ? x &= \mathbf{if} p x \mathbf{then} Inl x \mathbf{else} Inr x
 \end{aligned}$$

We obtain a point-wise definition of α^{op} for lists by the following derivation:

$$\begin{aligned}
& \alpha^{op} x \\
= & \{ \text{Definition of } \alpha^{op} \text{ and composition } \} \\
& (k_1 + (\text{head } \Delta \text{ tail})) (\text{null}?x) \\
= & \{ \text{Definition of } (?) \} \\
& (k_1 + (\text{head } \Delta \text{ tail})) (\text{if null } x \text{ then Inl } x \text{ else Inr } x) \\
= & \{ \text{Functor } + \} \\
& \text{if null } x \text{ then Inl } (k_1 x) \text{ else Inr } ((\text{head } \Delta \text{ tail}) x) \\
= & \{ \text{Definition of } k_1 \text{ and } (\Delta) \} \\
& \text{if null } x \text{ then Inl } () \text{ else Inr } (\text{head } x, \text{tail } x)
\end{aligned}$$

4.1 FG-Bialgebras

An alternative, and neater, model of types as bialgebras is discussed in [3] where observation is characterised instead by a G -coalgebra $\beta :: x \rightarrow Gx$ and a natural transformation $\text{cond} :: Gx \rightarrow Fx$ such that α and β are inverses via cond , that is, $\alpha^{op} = \text{cond} \circ \beta$ and $\beta^{op} = \alpha \circ \text{cond}$. For example, in the case of lists we have $Gx = \text{Bool} \times a \times x$, $\beta = \text{null } \Delta \text{ head } \Delta \text{ tail}$, and $\text{cond}(c, t, e) = \text{if } c \text{ then Inl } () \text{ else Inr } (t, e)$. Notice cond can be used to perform what we might call a ‘discriminated selection’, i.e., the selection of product components after discriminating over the sum via a predicate. This is illustrated by the following derivation:

$$\begin{aligned}
& (\text{cond} \circ \beta) x \\
= & \{ \text{Composition plus definition of } \beta \} \\
& \text{cond} ((\text{null } \Delta \text{ head } \Delta \text{ tail}) x) \\
= & \{ \text{Definition of } \Delta \} \\
& \text{cond} (\text{null } x, \text{head } x, \text{tail } x) \\
= & \{ \text{Definition of } \text{cond} \} \\
& \text{if null } x \text{ then Inl } () \text{ else Inr } (\text{head } x, \text{tail } x)
\end{aligned}$$

The G -coalgebra is neater because β only has ‘ Δ -joined’ observers and is therefore dual to α which has ‘ ∇ -joined’ constructors. The triple (x, α, β) is called an FG -bialgebra.

We conclude by noting that G and cond can be obtained from the sums-of-products structure of a polynomial F . Indeed, every sum in F gives rise to a Bool component (the discriminator’s argument) of G , and every proper (non-unit) product of F gives rise to a product component of G (product selection), etc. We do not elaborate on this but rather present in Section 5 a related method for obtaining G and other, more useful functions from the structure of F .

4.2 ADTs and Bialgebras

We explained in Section 4 that all observations on concrete types can be adequately captured by the initial F -algebra concept. However, something goes wrong with ADTs. Let us illustrate what it is using FIFO queues and ordered sets as examples. The FG -bialgebra for FIFO-queues is $(\text{Queue } a, \gamma, \nu)$ where F, G are the same functors as in lists, $\gamma = \text{empty } \nabla \text{ enq}$, and $\nu = \text{isEmpty } \Delta \text{ front } \Delta \text{ deq}$. The FG -bialgebra for ordered sets is $(\text{OrdSet } a, \gamma, \nu)$ where F, G are also the list functors, $\gamma = \text{empty } \nabla \text{ insert}$, and $\nu = \text{isEmpty } \Delta \text{ min } \Delta (\lambda s \rightarrow \text{remove } (\text{min } s) s)$. However, in both cases γ and ν are not inverses via cond . This applies in general to ADTs and is due to non-freeness, i.e., to the presence of equations among constructors [19]. For example, FIFO-queues satisfy, among others, the following two conditional equations. Constructor enq appears on both sides of the second equation:

$$\begin{aligned}
& (\text{isEmpty } q) \Rightarrow \text{deq } (\text{enq } x \ q) = q \\
\lnot & (\text{isEmpty } q) \Rightarrow \text{deq } (\text{enq } x \ q) = \text{enq } x \ (\text{deq } q)
\end{aligned}$$

What is missing in the categorical model is the equations. Indeed, ADTs are initial algebras but in a different category where objects (carriers) are term algebras (algebras giving meaning to terms generated from operator applications) *modulo equivalence relations* on terms introduced by the equations [19]. This is the reason for the inadequacy of the categorical model of types assumed by polytypic programming if assumed also for ADTs.

4.3 Intermission: ADT Classification

Parametric ADTs are often classified according to whether the number and logical position of elements is dependent or independent of properties of element types. For example, the C++ STL [22] classifies containers with iterators into sequences and collections. The Edison framework [26] speaks of sequences, collections, and associative collections. To underline the role of element-type properties we might speak of element-property-dependent/independent ADTs. In [24] we avoided that mouthful and spoke of *sensitive* and *insensitive* ADTs. Insensitive ADTs are stacks, queues, de-queues, arrays, etc, where element position is fixed in the operators (e.g., enq enqueues at the rear). Sensitive ADTs are sets, bags, ordered sets, ordered bags, heaps, priority queues, etc. These ADTs require element types with properties that are imposed by the equations that determine element number and position. Examples are element equality in sets, element comparison in ordered sets, etc. Some of these properties manifest as type-class constraints on element types. Constructors of sensitive ADTs use these properties to uphold the equations. For this reason we call them ‘intelligent’ constructors.

4.4 Extraction and Insertion

It is shown in [24] that given the FG -bialgebra (Aa, γ, ν) of an abstract type Aa , it is possible, under certain conditions explained below, to construct the following. First, FG -bialgebras (Ca, α, β') and (Ca, α', β) where Ca is a concrete type with same functors F and G . Second, an *extraction* function $\varepsilon :: Aa \rightarrow Ca$ and an *insertion* function $\iota :: Ca \rightarrow Aa$ which must only satisfy $\iota \circ \varepsilon = \text{id}$. In general, $\varepsilon \circ \iota \not\subseteq \text{id}$. More concretely:

$$\begin{aligned}
\varepsilon &= \alpha \circ \text{cond} \circ G \varepsilon \circ \nu \\
\iota &= \gamma \circ F \iota \circ \text{cond} \circ \beta
\end{aligned}$$

The insight is to notice that β does not occur in ε ’s definition and α does not occur in ι ’s definition. Thus, we may use (Ca, α, β') for extraction and (Ca, α', β) for insertion where the by-product bialgebra (Ca, α', β') views the concrete type as if behaving like the abstract type. In other words, α' and β' may not be inverses via cond . Contrast with Section 3.1.3 where viewing happens in the opposite direction. For example, in the case of FIFO queues and using lists as the concrete type we have:

$$\begin{aligned}
(\alpha, \beta') &= ([] \nabla (:), \text{null } \Delta \text{ head } \Delta \text{ tail}) \\
(\alpha', \beta) &= ([] \nabla \text{snoc}, \text{null } \Delta \text{ last } \Delta \text{ init}) \\
(\alpha', \beta') &= ([] \nabla \text{snoc}, \text{null } \Delta \text{ head } \Delta \text{ tail}) \\
(\gamma, \nu) &= (\text{empty } \nabla \text{enq}, \text{isEmpty } \Delta \text{ front } \Delta \text{ deq}) \\
\iota x &= \text{if null } x \text{ then empty else enq } (\text{last } x) \ (\iota (\text{init } x)) \\
\varepsilon x &= \text{if isEmpty } x \text{ then } [] \text{ else } (:) \ (\text{front } x) \ (\varepsilon (\text{deq } x))
\end{aligned}$$

Function snoc inserts elements at the rear of the list. Standard functions last and init deliver respectively the last element of a list and the list resulting after removing the last element. Clearly, $([a], \alpha', \beta')$ views the list as a FIFO queue. The bodies of ε and ι have been obtained by expanding their definitions for the given α, β , etc.

In the case of ordered sets, extraction and insertion may use identical list bialgebras because ordered-set constructors in γ are intelligent (i.e., take care of sorting and removing duplicates). More

precisely, $\alpha = \alpha' = [] \nabla (:)$, $\beta = \beta' = \text{null} \triangle \text{head} \triangle \text{tail}$, and ε and ι are obtained by expanding their definitions:

```

 $\iota x = \text{if } \text{null } x \text{ then empty else insert (head } x) (\iota (\text{tail } x))$ 
 $\varepsilon x = \text{if } \text{isEmpty } x \text{ then } []$ 
 $\text{else } (:) (\text{min } x) (\varepsilon ((\text{ls} \rightarrow \text{remove } (\text{min } s) s) x))$ 

```

In general, the concrete-type bialgebras are defined by programming, or picking from a library, a concrete type with the same *interface* functors as the ADT, and by programming, or picking from a library, reasonably efficient concrete-type operators to make $\iota \circ \varepsilon = \text{id}$ hold. The recipe is simple. For insensitive ADTs, the desired concrete-type operators are those satisfying similar equations than ADT operators. For example, *head* and *snoc* satisfy similar equations in lists than *front* and *enq* in queues. Contrast with Section 3.1.3 where what is needed is a counterpart of *last* in queues (an inverse of *enq*). For sensitive ADTs, the choice of concrete-type operators differs for extraction and insertion. Operators for extraction must be those that maintain in the concrete type the logical position of elements in the abstract type. Operators for insertion must be simply those that access the concrete type efficiently, for sensitive ADTs have intelligent constructors.

The ‘certain condition’ mentioned at the beginning of the section is that every constructable value must be observable. Let us explain what this means using a contrasting example: ordered sets versus unordered sets. An ordered set value constructed with *insert* can be observed with *min* and *remove*. However, an unordered set value cannot be observed, only interrogated with *member* (we are referring to unbounded sets as constructed in programs and assume referential transparency which rules out a non-deterministic ‘choice’ operator). Extraction can be defined for the former, not the latter, because unordered sets have no operators $f :: \text{Set } a \rightarrow a$ and $g :: \text{Set } a \rightarrow \text{Set } a$ that let us define $v = \text{isEmpty} \triangle f \triangle g$. The definability of ι and ε thus depends on the operators an ADT provides. Fortunately, both functions can be defined for many well-known ADTs (Section 5 and [24]).

We conclude the section contrasting ε and ι with translation functions (Sections 2 and 3.1.3). The former are recursive, the latter are not although they are invoked by recursive specialised functions at every recursive point. The former translate to/from a concrete type specified by the programmer, the latter to/from a predefined structure representation type. The former must satisfy one invertibility law whereas the latter must satisfy two. The former are flexible about concrete-type operators whereas the latter are not. The question to address is whether ι and ε are polytypic and, if so, on what notion of structure they are parametric.

5. Bialgebra Views

If every constructable value can be observed then G can be obtained from F . We assume $Fx = P_0 + \dots + P_{n-1}$ is an n -ary sum of m -ary products $P_i = E_{i_0} \times \dots \times E_{i_{m-1}}$. When $n = 0$ then $Fx = P$ with P a product, and when $m = 0$ then $P_i = 1_i$ (a unit type, the i subscript is there only to indicate its position). A product component E_{i_j} is either a kind-* type-variable, type name, or type application.

Let X be a type. We define a *bialgebra view* as an equation $X = FX$ together with the following operator declarations: for each sum in FX there are *discriminators* $\text{dsc}_i :: X \rightarrow \text{Bool}$, and for each product P_i there are *constructors* $\text{con}_i :: [X/x] P_i \rightarrow X$ and *selectors* $\text{sel}_{i_j} :: [X/x] P_i \rightarrow [X/x] E_{i_j}$. We write $[A/B]Z$ for ‘substitute A for B in Z ’. For example, for $Fx = 1 + a \times x$ and type X , the equation $X = 1 + a \times X$ and the operators $\text{dsc}_0 :: X \rightarrow \text{Bool}$, $\text{con}_0 :: 1 \rightarrow X$, $\text{con}_1 :: (a \times X) \rightarrow X$, $\text{sel}_{1_0} :: X \rightarrow a$, and $\text{sel}_{1_1} :: X \rightarrow X$ make up a bialgebra view. A functor G can be defined from the target types of discriminators and selectors: $Gx = [x/X](\text{Bool} \times a \times X)$.

There are three important facts about bialgebra views. First, operator names and type signatures have been obtained from the

```

 $\text{gen}[[h]][[x]] X s = [[h x =]] @ (\text{gen}_+ [[h]][[x]] X s)$ 
 $\text{gen}_+ [[h]][[x]] X (P_0 + \dots + P_{n-1}) =$ 
 $[[ \text{if } \text{dsc}_0 x \text{ then } ]] @ (\text{gen}_\times [[h]][[x]] X P_0)$ 
 $@ \dots @ [[ \text{else if } \text{dsc}_{n-1} x \text{ then } ]] @ (\text{gen}_\times [[h]][[x]] X P_{n-2})$ 
 $@ [[ \text{else } ]] @ (\text{gen}_\times [[h]][[x]] X P_{n-1})$ 
 $\text{gen}_\times [[h]][[x]] X 1_i = [[\text{con}_i]]$ 
 $\text{gen}_\times [[h]][[x]] X (E_{i_0} \times \dots \times E_{i_{m-1}}) =$ 
 $[[\text{con}_i]] @ (\text{gen}_E [[h]][[x]] X E_{i_0}) @ \dots @ (\text{gen}_E [[h]][[x]] X E_{i_{m-1}})$ 
 $\text{gen}_E [[h]][[x]] X X_{j_k} = [[h (\text{sel}_{j_k} x)]]$ 
 $\text{gen}_E [[h]][[x]] X - = [[\text{sel}_{j_k} x]]$ 

```

Figure 4. Object-level data is enclosed in semantic brackets and concatenated with $@$. Function gen_+ generates a nested conditional from a sum. Function gen_\times generates constructor calls for every product. Function gen_E generates selections for every product item and recursive calls to h when the item is X . All these functions pattern-match on their third argument which carries the subscript parameters.

sums-of-products structure of FX , so it suffices to specify the latter and assume the operator naming and subscripting convention. Second, a bialgebra view induces an FG -bialgebra which we call a *bialgebra adapter*. For instance, in the previous example the adapter induced is (X, ψ, χ) where $\psi = \text{con}_0 \nabla \text{con}_1$ and $\chi = \text{dsc}_0 \triangle \text{sel}_{1_0} \triangle \text{sel}_{1_1}$. Adapters provide a means of abstracting over operator names. More precisely, we say a bialgebra *instantiates* an adapter when there is a syntactic mapping of type and operator names from the latter to the former which is correct with respect to operator type signatures. For example, FIFO queues instantiate our example adapter via the mappings $X \mapsto \text{Queue } a$, $\text{con}_0 \mapsto \text{empty}$, $\text{con}_1 \mapsto \text{enq}$, $\text{dsc}_0 \mapsto \text{isEmpty}$, $\text{sel}_{1_0} \mapsto \text{front}$, and $\text{sel}_{1_1} \mapsto \text{deq}$. (We have obviated intermediate uncurrying.) Many other concrete and abstract types instantiate this adapter: lists, stacks, catenable lists, priority queues, ordered bags, sortable collections, heaps, double-ended queues (by choosing a particular FIFO behaviour), etc. The reader can find their interfaces in [25]. Finally, the function $h = \psi \circ Fh \circ \text{cond} \circ \chi$ is constructed from the adapter induced by the view. The point-wise definition for the example adapter above is:

```

 $h x = \text{if } \text{dsc}_0 x \text{ then } \text{con}_0$ 
 $\text{else } \text{con}_1 (\text{sel}_{1_0} x) (h (\text{sel}_{1_1} x))$ 

```

Function h is extremely important. First, it is identical to ι and ε save for operator names. Indeed, ι and ε will be defined as instantiations of h using adapters. Second, its point-wise definition can be generated from the sums-of-products structure FX of a bialgebra view (which induces the adapter) as specified in Figure 4. For example, the point-wise definition above is the result of $\text{gen}[[h]][[x]] X (1 + a \times X)$ where gen can be understood as a compiler meta-function that generates the code for an argument function name h with formal parameter x . The following sections explain the relevance and utility of gen . Notice gen is not definable in Generic Haskell because it is defined on n -ary, not binary, sums and products.

6. Polytypic Programming with ADTs

In the following sections we detail how polytypic programming with ADTs can be achieved by means of bialgebra views, adapters, and polytypic definitions of ι and ε . We introduce the language extensions and compile-time mechanisms needed for classic-style Generic Haskell to support them. The language extensions are minimal and backward-compatible, but compiler extensions are re-

quired. We reiterate that our aim is to demonstrate implementability, not to develop a definitive implementation.

We make the following assumptions about ADTs. First, they are specified algebraically [19] and implemented as Haskell types which are instances of type-class interfaces (e.g. Figure 3 and [25]). We assume minimal interfaces, in the spirit of the C++ STL. Second, ADTs are either non-parametric or parametric on a fixed number of first-order arguments. Formally, their kinds conform to the grammar $k ::= * \mid * \rightarrow k$. Most frequently-used and well-known ADTs fall under this category [25]. Parametric ADTs may have type-class constraints on any of their type arguments. For instance, ordered sets have an *Ord* constraint.

Let us anticipate what is to come. First, t and ε are polytypic on the sums-of-products structure of bialgebra views. Their code is generated by the compiler as specified by *gen* (Figure 4). Second, programmers define bialgebra views and declare how ADTs and concrete types instantiate the adapters induced by the views, providing names for such instantiations. Third, programmers define polytypic functions on ADTs which are parametric on these named instantiations. Polytypic function bodies consist merely of compositions of polytypic extraction, polytypic insertion, and calls to vanilla polytypic functions on the concrete types associated with adapters. Polytypic applications are replaced at compile time by calls to generated specialisations. Finally, polytypic-function extension is supported: programmers may provide specialised operators for insertion, extraction, or polytypic functions if desired.

6.1 Defining Bialgebra Views and Adapters

Programmers declare *bialgebra views* after the fashion of data type declarations using the **view** keyword. For example:

```
view Linear t a    = 1 + a × (t a)
view Tuple3 t a b c = a × b × c
view Tuple3'      = Tuple3
where { a = Day; b = Month; c = Year }
```

The *Linear* view defines the equation $t a = F (t a)$ where $Fx = 1 + a \times x$. The *Tuple3* view defines the equation $t a b c = F (t a b c)$ where $Fx = a \times b \times c$. The *Tuple3'* view is a *particularisation* of *Tuple3* where type arguments are instantiated to actual types.

In general, views are of the form '**view** Name $X = FX$ ' (parametric) or '**view** Name₁ = Name₂ **where** *Plist*' (non-parametric). Name is a view name, X has the form ' $t a_1 \dots a_n$ ' for some variables t, a_1, \dots, a_n , with $n > 0$, F is a sums-of-products functor (Section 5), and *Plist* is a particularisation list ' $\{a_1 = T_1; \dots; a_n = T_n\}$ ' in which T_i are kind- $*$ types.

Programmers must understand view declarations as implicit declarations of multi-parameter type classes with subscripted operators as members. In other words, they must understand view declarations as ways of generating type-class encodings of the bialgebra view concept (Section 5). The following are classes for the example views above. We have again obviated intermediate uncurrying:

```
class Tuple3 t a b c where
  con0 :: a → b → c → t a b c
  sel0 :: t a b c → a
  sel1 :: t a b c → b
  sel2 :: t a b c → c

class Tuple3' t where
  con0 :: Day → Month → Year → t
  sel0 :: t → Day
  sel1 :: t → Month
  sel2 :: t → Year

class Linear t a where
  dsc0 :: t a → Bool
  con0 :: t a
  con1 :: a → t a → t a
  sel10 :: t a → a
  sel11 :: t a → t a
```

The compiler will actually generate slightly different class declarations (Section 6.3), but programmers need not be aware of this.

Notice that in class *Tuple3'* type variables are replaced by particularised types and variable t takes no arguments. Classes for particularised views must be generated in similar fashion.

Programmers declare *bialgebra adapters* as *named instances* of the classes. In contrast to ordinary instances, named instances have a name, declared using the **by** keyword, which allows the declaration of what would otherwise be duplicate instances. Named instances can be encoded directly in Haskell as instances of multi-parameter type classes with extra type-variable arguments (Section 6.3). Figure 5 shows adapter examples for FIFO queues, ordered sets, dates, two adapters for Haskell lists, and finally an adapter for a tuple of *Date* data.

6.2 Defining Polytypic Functions on ADTs

6.2.1 Polyaric Types

In general, polykinded types have the following form:

$$\begin{aligned} P\{\{*\}\} \quad t_1 \dots t_n &= \sigma \\ P\{\{k \rightarrow v\}\} t_1 \dots t_n &= \\ \forall a_1 \dots a_n. P\{\{k\}\} a_1 \dots a_n &\rightarrow P\{\{v\}\} (t_1 a_1) \dots (t_n a_n) \end{aligned}$$

In the kind- $*$ case, σ is a type signature where universal quantification is only permitted over type variables of kinds $*$ and $* \rightarrow *$ [7]. As mentioned in Section 6, we are concerned with ADTs that take a fixed number (possibly zero) of first-order arguments. Hence, we define the types of polytypic functions on ADTs more conveniently by induction on their arity instead of by induction on their kind. We define *polyaric types* which have the following form:

$$\begin{aligned} P\{\{0\}\} \quad t_1 \dots t_n &= \sigma \\ P\{\{n+1\}\} t_1 \dots t_n &= \\ \forall a_1 \dots a_n. P\{\{0\}\} a_1 \dots a_n &\rightarrow P\{\{n\}\} (t_1 a_1) \dots (t_n a_n) \end{aligned}$$

For example, polyaric *Map* is defined thus:

```
type Map\{0\} t1 t2 = t1 → t2
type Map\{n+1\} t1 t2 =
  ∀ a1 a2. Map\{0\} a1 a2 → Map\{n\} (t1 a1) (t2 a2)
```

An instantiation example:

```
Map\{1\} q q = ∀ a1 a2. Map\{0\} a1 a2 → Map\{0\} (q a1) (q a2)
              = ∀ a1 a2. (a1 → a2) → Map\{0\} (q a1) (q a2)
              = ∀ a1 a2. (a1 → a2) → (q a1 → q a2)
```

A polyaric type $P\{\{n\}\}$ is trivially translated to a polykinded type $P\{\{toKind(n)\}\}$ where:

$$\begin{aligned} toKind (0) &= * \\ toKind (n+1) &= * \rightarrow toKind (n) \end{aligned}$$

6.2.2 Polytypic Functions

Polytypic functions on ADTs have polyaric types. They are parametric on the names of two bialgebra adapters: an abstract-type adapter and a concrete-type adapter. They are not defined by cases but as compositions of polytypic extraction, polytypic insertion (both parametric on the same bialgebra adapters), and vanilla polytypic functions on the concrete type associated with the concrete-type adapter.

Figure 6 shows three examples: *gsize*, *gmap*, and *geq*, which have, respectively, polyaric types *Size*, *Map*, and *Eq*. These functions compute the size, map, and equality of ADTs for which the programmer has provided adapters. The v argument is an ADT adapter and the w argument is a concrete-type adapter. The functions share names with vanilla versions defined in Section 2, but are clearly differentiated by their arguments: two adapters instead of a type. They also take function arguments of the form $g\{\{arity\}\} v\}$ which we explain shortly. Compile-time functions *arity* and *type*

<p>instance <i>Queue</i> $q \Rightarrow \text{Linear } q \text{ a by } \text{QueueV}$ where <i>dsc</i>₀ = <i>Queue.isEmpty</i> <i>con</i>₀ = <i>Queue.empty</i> <i>con</i>₁ = <i>Queue.enq</i> <i>sel</i>_{1₀} = <i>Queue.front</i> <i>sel</i>_{1₁} = <i>Queue.deq</i></p> <p>instance <i>Linear</i> $[] \text{ a by } \text{ListV1}$ where <i>dsc</i>₀ = <i>null</i> <i>con</i>₀ = $[]$ <i>con</i>₁ = $(:)$ <i>sel</i>_{1₀} = <i>last</i> <i>sel</i>_{1₁} = <i>init</i></p> <p>instance <i>Date</i> $d \Rightarrow \text{Tuple3}' d \text{ by } \text{DateV}$ where <i>con</i>₀ = <i>mkDate</i> <i>sel</i>_{0₀} = <i>getDay</i> <i>sel</i>_{0₁} = <i>getMonth</i> <i>sel</i>_{0₂} = <i>getYear</i></p>	<p>instance $(\text{Ord } a, \text{OrdSet } s) \Rightarrow \text{Linear } s \text{ a by } \text{OrdSetV}$ where <i>dsc</i>₀ = <i>OrdSet.isEmpty</i> <i>con</i>₀ = <i>OrdSet.empty</i> <i>con</i>₁ = <i>OrdSet.insert</i> <i>sel</i>_{1₀} = <i>OrdSet.min</i> <i>sel</i>_{1₁} = $(\lambda s \rightarrow \text{OrdSet.remove } (\text{OrdSet.min } s) s)$</p> <p>instance <i>Ord</i> $a \Rightarrow \text{Linear } [] \text{ a by } \text{ListV2}$ where <i>dsc</i>₀ = <i>null</i> <i>con</i>₀ = $[]$ <i>con</i>₁ = $(:)$ <i>sel</i>_{1₀} = <i>head</i> <i>sel</i>_{1₁} = <i>tail</i></p> <p>instance <i>Tuple3'</i> $(\text{Day}, \text{Month}, \text{Year}) \text{ by } \text{Tuple3}'V$ where <i>con</i>₀ = $(, ,)$ <i>sel</i>_{0₀} $(x, -, -) = x$ <i>sel</i>_{0₁} $(-, y, -) = y$ <i>sel</i>_{0₂} $(-, -, z) = z$</p>
--	--

Figure 5. Bialgebra adapters are named instances.

<p>type <i>Size</i>$\{0\}$ t = $t \rightarrow \text{Int}$ type <i>Size</i>$\{n+1\}$ t = $\forall a. \text{Size}\{0\} a \rightarrow \text{Size}\{n\} (t a)$ <i>gsize</i>$\{v, w\}$:: <i>Size</i>$\{\text{arity } v\}$ (type v) <i>gsize</i>$\{v, w\}$ $g\{\text{arity } v\}$ = <i>gsize</i>$\{\text{type } w\}$ $g\{\text{arity } v\} \circ \varepsilon\{v, w\}$ type <i>Map</i>$\{0\}$ $t_1 t_2$ = $t_1 \rightarrow t_2$ type <i>Map</i>$\{n+1\}$ $t_1 t_2$ = $\forall a_1 a_2. \text{Map}\{0\} a_1 a_2 \rightarrow \text{Map}\{n\} (t_1 a_1) (t_2 a_2)$ <i>gmap</i>$\{v, w\}$:: <i>Map</i>$\{\text{arity } v\}$ (type v) (type v) <i>gmap</i>$\{v, w\}$ $g\{\text{arity } v\}$ = $t\{v, w\} \circ \text{gmap}\{\text{type } w\} g\{\text{arity } v\} \circ \varepsilon\{v, w\}$ type <i>Eq</i>$\{0\}$ t = $t \rightarrow t \rightarrow \text{Bool}$ type <i>Eq</i>$\{n+1\}$ t = $\forall a. \text{Eq}\{0\} a \rightarrow \text{Eq}\{n\} (t a)$ <i>geq</i>$\{v, w\}$:: <i>Eq</i>$\{\text{arity } v\}$ (type v) <i>geq</i>$\{v, w\}$ $g\{\text{arity } v\} x y = \text{geq}\{\text{type } w\} g\{\text{arity } v\} (\varepsilon\{v, w\} x) (\varepsilon\{v, w\} y)$</p>

Figure 6. Polytypic functions *gsize*, *gmap*, and *geq* and their polyaric types *Size*, *Map*, and *Eq*.

respectively return the arity and data type name associated with the adapter argument (Section 6.3). They have been included in the figure for readability but programmers could omit them and assume them introduced behind the scenes by the compiler.

In words, the size of an ADT of arbitrary arity n (has data of n types) is computed by extracting all data into the concrete type (of the same arity) and then applying vanilla size on the latter. The size of element data is computed by n argument functions that will be passed to the specialisation. This is what $g\{\text{arity } v\}$ means (Section 6.3). The map of an ADT is computed by extracting the elements into the concrete type, applying vanilla map over the latter, and then inserting the elements back into the ADT. Finally, equality for two ADTs is computed by extracting the data into two concrete-type values and applying vanilla equality on the latter.

We conclude with examples of usage. Let q be a queue value with three elements and let s be the ordered-set value defined in Section 3. The application $\text{gsize}\{\text{QueueV}, \text{ListV1}\} (\text{const } 1) q$ delivers the value 3 independently of q 's implementation. The application $\text{gsize}\{\text{OrdSetV}, \text{ListV2}\} (\text{const } 1) s$ delivers 2. Finally, after the applications $\text{gmap}\{\text{OrdSetV}, \text{ListV2}\} (\text{const } 0) s$ and $\text{gmap}\{\text{OrdSetV}, \text{ListV2}\} \text{negate } s$, internally s is respectively rep-

resented by $S [0]$ and $S [-2, -1]$ in the list implementation of Section 3.

6.3 Specialisation

Specialisation requires two steps. First, the compiler must process views and adapters to book-keep the information they provide. Second, it must replace polytypic applications on actual adapter arguments by calls to statically-generated specialised functions.

6.3.1 Processing views and adapters.

The compiler must generate multi-parameter type classes from view declarations. Figure 7(1) shows the actual classes that must be generated for *Linear* and *Tuple3'* views. They differ from the ones shown in Section 5 in an extra type variable n which will be a type representing adapter names. All class members take this name as a first argument and we explain why in the next paragraph.

The compiler must also generate, using *gen* (Figure 4), the specialisation of h (Section 5) for the views. Figure 7(1) shows specialisations $h\text{Linear}$ and $h\text{Tuple3}'$. The former's code is generated by $\text{gen}\llbracket h\text{Linear} \rrbracket\llbracket x \rrbracket (t a) (1 + a \times (t a))$ and the latter's by $\text{gen}\llbracket h\text{Tuple3}' \rrbracket\llbracket x \rrbracket (t a b c) (a \times b \times c)$ (the choice of formal parameter name x is arbitrary), with the addition of two parameters n_1

class *Linear* *t a n* **where**

*dsc*₀ :: *n* → *t a* → *Bool*
*con*₀ :: *n* → *t a*
*con*₁ :: *n* → *a* → *t a* → *t a*
*sel*₁₀ :: *n* → *t a* → *a*
*sel*₁₁ :: *n* → *t a* → *t a*

hLinear :: (*Linear* *t*₁ *a* *n*₁, *Linear* *t*₂ *a* *n*₂) ⇒ *n*₁ → *n*₂ → *t*₁ *a* → *t*₂ *a*
hLinear *n*₁ *n*₂ *x* = **if** *dsc*₀ *n*₁ *x* **then** *con*₀ *n*₂
else *con*₁ *n*₂ (*sel*₁₀ *n*₁ *x*) (*hLinear* *n*₁ *n*₂ (*sel*₁₁ *n*₁ *x*))

class *Tuple3'* *t n* **where**

*con*₀ :: *n* → *Day* → *Month* → *Year* → *t*
*sel*₀₀ :: *n* → *t* → *Day*
*sel*₀₁ :: *n* → *t* → *Month*
*sel*₀₂ :: *n* → *t* → *Year*

hTuple3' :: (*Tuple3'* *t*₁ *n*₁, *Tuple3'* *t*₂ *n*₂) ⇒ *n*₁ → *n*₂ → *t*₁ → *t*₂
hTuple3' *n*₁ *n*₂ *x* = *con*₀ *n*₂ (*sel*₀₀ *n*₁ *x*) (*sel*₀₁ *n*₁ *x*) (*sel*₀₂ *n*₁ *x*)

data *QueueV* = *QueueV*

instance *Queue* *q* ⇒ *Linear* *q a* *QueueV* **where**

*dsc*₀ = *const* *Queue.isEmpty*
*con*₀ = *const* *Queue.empty*
*con*₁ = *const* *Queue.enq*
*sel*₁₀ = *const* *Queue.front*
*sel*₁₁ = *const* *Queue.deq*

data *ListV1* = *ListV1*

instance *Linear* [] *a* *ListV1* **where**

*dsc*₀ = *const* *null*
*con*₀ = *const* []
*con*₁ = *const* (:)
*sel*₁₀ = *const* *last*
*sel*₁₁ = *const* *init*

data *DateV* = *DateV*

instance *Date* *d* ⇒ *Tuple3'* *d* *DateV* **where**

*con*₀ = *const* *mkDate*
*sel*₀₀ = *const* *getDay*
*sel*₀₁ = *const* *getMonth*
*sel*₀₂ = *const* *getYear*

data *OrdSetV* = *OrdSetV*

instance (*Ord* *a*, *OrdSet* *s*) ⇒ *Linear* *s a* *OrdSetV* **where**

*dsc*₀ = *const* *OrdSet.isEmpty*
*con*₀ = *const* *OrdSet.empty*
*con*₁ = *const* *OrdSet.insert*
*sel*₁₀ = *const* *OrdSet.min*
*sel*₁₁ = *const* (λ*s* → *OrdSet.remove* (*OrdSet.min* *s*) *s*)

data *ListV2* = *ListV2*

instance *Ord* *a* ⇒ *Linear* [] *a* *ListV2* **where**

*dsc*₀ = *const* *null*
*con*₀ = *const* []
*con*₁ = *const* (:)
*sel*₁₀ = *const* *head*
*sel*₁₁ = *const* *tail*

data *Tuple3'V* = *Tuple3'V*

instance *Tuple3'* (*Day*, *Month*, *Year*) *Tuple3'V* **where**

*con*₀ _ = (, ,)
*sel*₀₀ _ (x, -, -) = *x*
*sel*₀₁ _ (-, y, -) = *y*
*sel*₀₂ _ (-, -, z) = *z*

QueueV = ⟨*Linear*, *Queue*, *q*, 1, 1 + *a* × (*q a*),
 ⟨*dsc*₀ ↦ *Queue.isEmpty*,
 ...
*sel*₁₁ ↦ *Queue.deq*⟩,
 ⟨⟩⟩

ListV1 = ⟨*Linear*, -, [], 1, 1 + *a* × [*a*],
 ⟨*dsc*₀ ↦ *null*, ..., *sel*₁₁ ↦ *init*⟩,
 ⟨⟩⟩

DateV = ⟨*Tuple3'*, *Date*, *d*, 3, *a* × *b* × *c*,
 ⟨*con*₀ ↦ *mkDate*, ..., *sel*₀₂ ↦ *getYear*⟩,
 ⟨*a* ↦ *Day*, *b* ↦ *Month*, *c* ↦ *Year*⟩⟩

OrdSetV = ⟨*Linear*, *OrdSet*, *s*, 1, 1 + *a* × (*s a*),
 ⟨*dsc*₀ ↦ *OrdSet.isEmpty*,
 ...
*sel*₁₁ ↦ (λ*s* → *OrdSet.remove* ...)⟩,
 ⟨⟩⟩

ListV2 = ⟨*Linear*, -, [], 1, 1 + *a* × [*a*],
 ⟨*dsc*₀ ↦ *null*, ..., *sel*₁₁ ↦ *tail*⟩,
 ⟨⟩⟩

Tuple3'V = ⟨*Tuple3'*, -, (, ,), 3, *a* × *b* × *c*,
 ⟨*con*₀ ↦ (, ,), ..., *sel*₀₂ ↦ (λ(-, -, z) → *z*)⟩,
 ⟨*a* ↦ *Day*, *b* ↦ *Month*, *c* ↦ *Year*⟩⟩

vname ⟨*n*, -, -, -, -, -⟩ = *n*

class ⟨-, *c*, -, -, -, -⟩ = *c*

type ⟨-, -, *t*, -, -, -⟩ = *t*

arity ⟨-, -, -, *a*, -, -, -⟩ = *a*

vshape ⟨-, -, -, *v*, -, -⟩ = *v*

ops ⟨-, -, -, *o*, -⟩ = *o*

plist ⟨-, -, -, -, *p*⟩ = *p*

Figure 7. Implementation details: (1) Type classes and specialisations generated for *Linear* and *Tuple3'* views. (2) Vanilla instance declarations generated from named instance declarations. (3) Information captured by the compiler from view and adapter declarations. (4) Some compile-time functions for accessing the information.

and n_2 which stand for types representing adapter names associated with types t_1 and t_2 . These parameters must be passed in operator calls to help the Haskell type checker select the appropriate instance of the class. Thus, $dsc_0 n_1$ is a discriminator acting on a value of type t_1 whereas $con_0 n_2$ is a constructor creating a value of type t_2 .

The compiler must process adapter declarations (named instances) and encode them as vanilla instances of the type classes generated for the views. The names of named instances are encoded by newly-generated data types with a single value constructor. Figure 7(2) shows the processed code for the adapter declarations of Figure 5. All operators are preceded either by a *const* call or by a dummy underscore pattern (if the operator was defined by pattern matching in the named instance) in order to ignore the name parameter, for it only plays a role during type checking.

The information provided by view and adapter declarations is to be book-kept by the compiler in an internal structure which captures the name of the bialgebra view, the name of the type-class ADT interface (or an empty entry in the case of concrete types), the type variable or *type constructor* used in the instance, the arity of the view, the sum-of-products of the view, the syntactic mapping of operators, and the particularisation information (if applicable). Figure 7(3) shows the structures for the adapters of Figure 5. We assume a set of trivial compile-time functions, shown in Figure 7(4) and written in sans-serif, for accessing each field in the structure. Notice that the sums-of-products structure of particularised views is that of the view they particularise, hence, *vshape* $Tuple^3 V$ delivers $a \times b \times c$. Notice also that type $Tuple^3 V$ delivers $(, ,)$, that is, the *type constructor* used in the named instance.

6.3.2 Specialising Polytypic Applications.

Polytypic applications can now take the form $g\{V, W\}$, where V, W are *actual* adapter-name parameters. These polytypic applications are replaced at compile time by calls to their generated specialisations gVW which have the following form:

$$\begin{aligned} gVW &:: C t \Rightarrow (\text{inst } ((P' \{[k]\} t \dots t) \diamond \Delta) L) \\ gVW g_1 \dots g_n &= B \end{aligned}$$

In the type signature, $C = (\text{class } V)$, $t = (\text{type } V)$, $n = (\text{arity } V)$, $L = (\text{plist } V)$, and $P' \{[k]\} t \dots t$ is the *context-parametric* polykinded type [23] obtained at compile time from the polyaric type $P \{[n]\} t \dots t$ of $g\{v, w\}$, with $k = \text{toKind } (n)$. Finally, Δ is the list of type-class constraints associated with the abstract type. The context-parametric polykinded type is expanded, with \diamond pushing the constraints in Δ to appropriate universally-quantified variables (see Appendix A for technical details). Compile-time function *inst* takes the expanded type signature and particularises it if applicable. More precisely, if L is not empty, it particularises universally-quantified type variables to particular types as indicated by L , and also applies the substitution $[t / (t a_1 \dots a_n)]$ to the type signature. If L is empty, *inst* behaves as an identity.

The body B is the specialisation of the polytypic function body. It is obtained as follows. First, calls to $\varepsilon\{V, W\}$ and $\iota\{V, W\}$ are replaced by calls to generated specialisations εVW and ιVW which are simply calls to h , the function obtained by:

$$\text{gen}[h][x] ((\text{type } V) a_1 \dots a_n) (\text{vshape } V)$$

Recall (type V) is a type constructor so for particularised views the result delivered must be applied to T_i instead of a_i . Extraction invokes h passing the types encoding adapter names V and W . Insertion invokes h passing W and V . Second, the concrete type-constructor name (type W) is placed in the call to the vanilla polytypic function.

Let us illustrate the process for the adapter examples of Figure 7(2).

The polytypic application $gmap\{QueueV, ListV1\}$ is replaced by a call to the specialised function $gmapQueueVListV1$ whose type signature is:

$$gmapQueueVListV1 :: Queue\ q \Rightarrow Map\{[1]\} q\ q$$

We have omitted *inst* from the signature because the particularisation list is empty and *inst* is the identity. We have written the polyaric type because there are no type-class constraints on the element type and the expansion of the context-parametric type that would be manufactured from the polyaric type by the compiler would deliver the same result, shown in Figure 8(1). The figure also shows the specialised body. The specialised versions of ι and ε , namely $\iota QueueVListV1$ and $\varepsilon QueueVListV1$, simply invoke $hLinear$ on the types encoding the adapter names. Their type signatures have been included for readability and can be inferred by the compiler.

The polytypic application $gmap\{OrdSetV, ListV2\}$ is replaced by a call to the specialised function $gmapOrdSetVListV2$ whose type signature is:

$$gmapOrdSetVListV2 :: OrdSet\ s \Rightarrow ((Map' \{[* \rightarrow *]\} s\ s) \diamond [Ord])$$

Again, the particularisation list is empty and *inst* is the identity. The expanded type and body are also shown in Figure 8(1), along with the specialised versions of ι and ε . The *Ord* constraint is placed when expanding the context-parametric polykinded type $Map' \{[* \rightarrow *]\}$ together with a constraint list consisting only of *Ord*. This type is manufactured by the compiler from the polyaric type $Map\{[1]\}$ (Appendix A).

The polytypic application $gsize\{DateV, Tuple^3 V\}$ is replaced by a call to the specialised function $gsizeDateVTuple^3 V$ whose type is given by:

$$Date\ d \Rightarrow (\text{inst } (Size\{[3]\} d) \langle (a, Day), (b, Month), (c, Year) \rangle)$$

We have written the polyaric type because there are also no type-class constraints on the element type. Figure 8(2) shows the expansion of $(Size\{[3]\} d)$ and the result after particularisation. The expanded type and body are shown in Figure 8(1). The reader should compare $gsizeDateVTuple^3 V$ with the size function for dates shown at the end of Section 3.

6.4 Polytypic-function Extension

Polytypic-function extension can be accommodated after the fashion of template specialisation in C++: definitions for ι , ε , or polytypic functions are simply provided at specific types:

$$\begin{aligned} \varepsilon\{OrdSet, []\} &= \text{enumerate} \\ \varepsilon\{Queue, []\} &= \text{toList} \\ \iota\{Queue, []\} &= \text{fromList} \\ gsize\{OrdSet, -\} g &= \text{cardinality} \end{aligned}$$

In general, these definitions have the form $g\{C, T\} g_1 \dots g_n = o$ and instruct the compiler to bypass the generation process and replace polytypic applications $g\{V, W\}$ by the right-hand-side operator o when $C = (\text{class } V)$ and $T = (\text{type } W)$. The compiler has to check that types match, i.e., that *toList* has type $Queue\ q \Rightarrow q\ a \rightarrow [a]$, etc. Notice *gsize* is specialised to *cardinality*:: $(OrdSet\ s, Ord\ a) \Rightarrow s\ a \rightarrow Int$. The concrete type is irrelevant (it does not appear in the type signature) and can be omitted using underscore. Notice also that the size of elements is hardwired in *cardinality* and, therefore, argument function g is ignored.

7. Discussion and Appraisal

With regard to programming effort, in our proposal programmers must define new versions of polytypic functions for ADTs, but these are defined once. Polyaric types are sugar and not strictly nec-

$gmapQueueVListVI$	$:: Queue\ q \Rightarrow (\forall a_1 a_2. (a_1 \rightarrow a_2) \rightarrow (q\ a_1 \rightarrow q\ a_2))$
$gmapQueueVListVI\ g_1$	$= \iota QueueVListVI \circ gmap\{\{\}\}\ g_1 \circ \varepsilon QueueVListVI$
$\varepsilon QueueVListVI$	$:: Queue\ q \Rightarrow q\ a \rightarrow [a]$
$\varepsilon QueueVListVI$	$= hLinear\ QueueV\ ListVI$
$\iota QueueVListVI$	$:: Queue\ q \Rightarrow [a] \rightarrow q\ a$
$\iota QueueVListVI$	$= hLinear\ ListVI\ QueueV$
$gmapOrdSetVListV2$	$:: OrdSet\ s \Rightarrow (\forall a_1 a_2. (Ord\ a_1, Ord\ a_2) \Rightarrow (a_1 \rightarrow a_2) \rightarrow (s\ a_1 \rightarrow s\ a_2))$
$gmapOrdSetVListV2\ g_1$	$= \iota OrdSetVListV2 \circ gmap\{\{\}\}\ g_1 \circ \varepsilon OrdSetVListV2$
$\varepsilon OrdSetVListV2$	$:: (OrdSet\ s, Ord\ a) \Rightarrow s\ a \rightarrow [a]$
$\varepsilon OrdSetVListV2$	$= hLinear\ OrdSetV\ ListV2$
$\iota OrdSetVListV2$	$:: (OrdSet\ s, Ord\ a) \Rightarrow [a] \rightarrow s\ a$
$\iota OrdSetVListV2$	$= hLinear\ ListV2\ OrdSetV$
$gsizeDateVTuple3'V$	$:: Date\ d \Rightarrow (Day \rightarrow Int) \rightarrow (Month \rightarrow Int) \rightarrow (Year \rightarrow Int) \rightarrow d \rightarrow Int$
$gsizeDateVTuple3'V\ g_1\ g_2\ g_3$	$= gsize\{\{(\cdot, \cdot)\}\}\ g_1\ g_2\ g_3 \circ \varepsilon DateVTuple3'V$
$\varepsilon DateVTuple3'V$	$:: Date\ d \Rightarrow d \rightarrow (Day, Month, Year)$
$\varepsilon DateVTuple3'V$	$= hTuple3'\ DateV\ Tuple3'V$
$\iota DateVTuple3'V$	$:: Date\ d \Rightarrow (Day, Month, Year) \rightarrow d$
$\iota DateVTuple3'V$	$= hTuple3'\ Tuple3'V\ DateV$

$Size\{\{3\}\}\ d = \forall a. Size\{\{0\}\}\ a \rightarrow Size\{\{2\}\}\ (d\ a)$
$= \forall a. (a \rightarrow Int) \rightarrow (\forall b. Size\{\{0\}\}\ b \rightarrow Size\{\{1\}\}\ (d\ a\ b))$
$= \forall a. (a \rightarrow Int) \rightarrow (\forall b. (b \rightarrow Int) \rightarrow Size\{\{1\}\}\ (d\ a\ b))$
$= \forall a. (a \rightarrow Int) \rightarrow (\forall b. (b \rightarrow Int) \rightarrow (\forall c. (c \rightarrow Int) \rightarrow Size\{\{0\}\}\ (d\ a\ b\ c)))$
$= \forall a. (a \rightarrow Int) \rightarrow (\forall b. (b \rightarrow Int) \rightarrow (\forall c. (c \rightarrow Int) \rightarrow (d\ a\ b\ c \rightarrow Int)))$
$inst\ (Size\{\{3\}\}\ d)\ \langle (a, Day), (b, Month), (c, Year) \rangle = (Day \rightarrow Int) \rightarrow (Month \rightarrow Int) \rightarrow (Year \rightarrow Int) \rightarrow d \rightarrow Int$

Figure 8. Implementation details (cont’d): (1) Specialised function definitions for polytypic applications $gmap\{QueueV, ListVI\}$, $gmap\{OrdSetV, ListV2\}$, and $gsize\{DateV, Tuple3'V\}$. (2) Expansion of polyaric $Size$ and result after $inst$.

essary, polykinded types could have been used instead. Programmers must also define bialgebra views, but these are defined once and can be re-used with various adapters. A set of typical view declarations could be provided in a library. Finally, programmers must define adapters per ADT, but concrete adapters can be re-used with several ADT adapters. For example, we could have used $ListVI$ instead of $ListV2$ in conjunction with $OrdSetV$. Certainly, some work is required from programmers, but not an unreasonable amount. In fact, we argue it is less than the amount required by the C++ STL (Section 8).

The language extensions proposed (view declarations, polytypic definitions) are reasonable and backward-compatible. As mentioned above, polyaric types are not strictly necessary. However, substantial compiler extensions are needed: support for context-parametric polykinded types (Appendix A), compile-time code generation process, and book-keeping of adapter structures.

With regard to efficiency, there are certainly trade-offs. Functions on abstract types are more efficiently computed when programmed by the ADT implementor and offered as part of the interface. For instance, the size of an unbounded ordered set can be computed in $O(n)$ whereas $gsize$ takes $O(n^2)$ in extracting and computing the list size. Mapping a function over an ordered set, taking duplicate removal and re-ordering into account (Section 3), could take $O(n^2 \log n)$ or less whereas $gmap$ takes $O(n^3)$. Nevertheless, such efficiency criticisms are applicable to all generic programming techniques that rely on to-and-fro translations to common representations, in our case a concrete type. The point is that genericity, flexibility, and re-use are gained instead. There is no generic programming in making ADTs supply the functionality (fat interfaces), and

this forces ADT designers to anticipate such functionality. Finally, polytypic-function extension is available if needed in particular situations, and compiler optimisations might be possible, e.g., fusing extraction, computation, and insertion for sensitive ADTs [24].

With regard to safety, at present nothing stops programmers from invoking polytypic functions on erroneous concrete-type adapters. For example, they may write $gmap\{QueueV, ListV2\}$ and mess up the queue. Requiring programmers to provide a list of valid associations of concrete-type and abstract-type adapters would only guarantee that polytypic applications conform to the list, but would not solve the problem, which is that no standard Haskell compiler will prove or validate $\iota \circ \varepsilon = id$. But neither will it prove or validate that an implementation of queues satisfies the queue’s semantics. Proof obligations require compilers with theorem proving or validating capabilities. These capabilities are not imposed by views or adapters, but by the presence of an axiomatic semantics: conditional equations are proof obligations.

There are type-class-based ADT-library proposals in Haskell. One example is the Edison framework [26]. It classifies ADTs into three categories (sequences, collections, and associative collections) and provides type-class interfaces for them. The interfaces are fat, they try to capture all the useful operators an instance of the class should implement. The library does not deploy generic programming beyond that offered by overloading. The work presented here fits nicely with Edison and similar libraries because types can be instances of several type classes each belonging to orthogonal hierarchies. For example, a type implementing queues can be simultaneously an instance of Edison’s *Sequence* class and an instance of *Linear*.

Finally, the bialgebra view and adapter concepts can be carried out to other polytypic languages and frameworks. To give an idea, in the Scrap Your Boilerplate approach [13], generic traversals are programmed in terms of a generic *gfoldl* function applicable to types in the *Data* class. Instances of *gfoldl* could be generated automatically by the compiler using specialisations of insertion and extraction. For example:

```
instance (Data a, Ord a) => Data (OrdSet a) where
  gfoldl k z s = k (z \OrdSetVListV2) (\eOrdSetVListV2 s)
```

8. Related Work

We are not aware of any other work addressing the conflict and cohabitation of polytypic programming with data abstraction.

F-algebras provide a neat categorical characterisation of free algebras. A categorical characterisation of non-free algebras is given in [3] where equations are replaced by ‘transformers’, i.e., mappings from *dialgebras* to *dialgebras* that satisfy certain properties. An *FG*-dialgebra is a mapping $\varphi :: Fx \rightarrow Gx$. However, homomorphisms in the category of such dialgebras cannot map to dialgebras with less equations or would not be uniquely defined.

The C++ STL is a library of generic algorithms implemented using C++ Templates. Genericity is attained by means of so-called iterators which abstract over container structure and enable their linear traversal. Iterators may be seen as the common representation of container objects. For example, a map function for queues can be implemented using a left-to-right iterator. However, due to linearity, a map for trees cannot be defined in this fashion. Conceptually, and side-effects aside, we might view an iterator as an adapter which specifies how to flatten a container into a list. (See [5] for a more precise characterisation of iterators.) Similarly, we might see bialgebra views and adapters as a means of specifying general iterators which do not flatten but copy the data into a concrete type (with the same interface structure) and back. In the STL, iterator instances are programmed per container. In our approach, bialgebra adapters are written per ADT. In the STL, algorithms are defined once using iterators. In our approach, polytypic functions are defined once and rely on automatically generated ι and ε . It requires more effort for a container’s *implementor* to write an iterator than for an ADT’s *user* to write a view and an adapter. Container *users* could write ‘external’ iterators using views and adapters, where the use of an intermediate concrete type facilitates the definition of bidirectional iterators. Finally, views and concrete adapters can be reused, but iterators are not reusable from one container to another. Our approach also supports polytypic-function extension and non-parametric ADTs.

Named instances are not new [12]. We have used a simple version of named instances for a particular form of constructor classes. Our encoding into vanilla instances is also simple and we believe it to be common lore.

9. Conclusions

Polytypic programming is founded on a regularity: function structure follows data structure. Data abstraction destroys this regularity. Polytypic functions must either adapt or be insensitive to ADT equations.

Generic programming techniques rely on computation via a typed translation to and from a common representation (e.g., structure representation types). Generic functions are parametric on type structure, and the choice of common representation for that structure should also be a parameter. We have proposed bialgebra views and adapters as a means of defining structure according to interfaces, and of introducing common representations (concrete types) that conform to interfaces. We say that ‘(vanilla) polytypic func-

tions are parametric on the structure of (concrete) types’. We have proposed ‘polytypic functions on ADTs which are parametric on the structure provided by bialgebra views and on a set of operator names that instantiate (adapt) the views’. We have proposed compiler and language extensions to support these concepts in Generic Haskell. We have also demonstrated their implementability.

References

- [1] Artem Alimarine and Marinus Plasmeijer. A generic programming extension for Clean. In *Implementation of Functional Languages*, 2001.
- [2] Michael Barr and Charles Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.
- [3] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, The Netherlands, 1992.
- [4] Jeremy Gibbons. Patterns in datatype-generic programming. In *Declarative Programming in the Context of Object-Oriented Languages*, 2003.
- [5] Jeremy Gibbons and Bruno C. d. S. Oliveira. The essence of the Iterator pattern. In *Mathematically-Structured Functional Programming*, 2006.
- [6] Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3), 2002.
- [7] Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and Theory. In *Generic Programming – Advanced Lectures*, volume 2793 of *LNCS*. Springer, 2003.
- [8] Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In *Mathematics of Program Construction*, 2006.
- [9] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University of Technology, 2000.
- [10] C. B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6), 1998.
- [11] Johan Jeuring and Patrik Jansson. Polytypic programming. In *Advanced Functional Programming*, volume 1129 of *LNCS*. Springer, 1996.
- [12] Wolfram Kahl and Jan Scheffczyk. Named instances for Haskell type classes. In *Haskell Workshop*, 2001.
- [13] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Workshop on Types in Language Design and Implementation*, 2003.
- [14] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *International Conference on Functional Programming*, 2005.
- [15] Ralf Lämmel, Eelco Visser, and Joost Visser. The essence of strategic programming. Tutorial, 2002.
- [16] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
- [17] Andres Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.
- [18] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, 1991.
- [19] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
- [20] A. Moors, F. Piessens, and W. Joosen. An object-oriented approach to datatype-generic programming. In *Workshop on Generic Programming*, 2006.
- [21] G. Munkby, A. Priesnitz, S. Schupp, and M. Zalewski. Scrap++: Scrap your boilerplate in C++. In *Workshop on Generic Programming*, 2006.

- [22] David Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.
- [23] Pablo Nogueira. Context-parametric polykinded types. In *Workshop on Generic Programming*, 2006.
- [24] Pablo Nogueira. When is an abstract data type a functor? In *Trends in Functional Programming*, volume 7. Intellect, 2006.
- [25] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [26] Chris Okasaki. An overview of Edison. In *Haskell Workshop*, 2000.
- [27] Pedro Palao Gostanza, Ricardo Peña, and Manuel Núñez. A new look at pattern matching in abstract data types. In *International Conference on Functional Programming*, 1996.
- [28] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*, 1987.
- [29] F. Warren Burton and Robert Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2), 1993.

A. Constraint Parametrisation

Context-parametric polykinded types were introduced in [23] because Generic Haskell does not support constrained types. Constrained types are parametric *concrete* types with at least one type-variable argument constrained by type-class membership. For example:

data $Ord\ a \Rightarrow Tree\ a = Empty \mid Node\ a\ (Tree\ a)\ (Tree\ a)$

The Generic Haskell compiler cannot specialise polytypic functions for constrained types because polykinded types are oblivious to constraints. Context-parametric polykinded types are polykinded types that are parametric on a (possibly empty) list of class constraints. More precisely, a polykinded type:

$$\begin{aligned} P\{\{*\}\} \quad t_1 \dots t_n &= \sigma \\ P\{\{k \rightarrow v\}\} \quad t_1 \dots t_n &= \\ &\forall a_1 \dots a_n. P\{\{k\}\} \quad a_1 \dots a_n \rightarrow P\{\{v\}\} \quad (t_1\ a_1) \dots (t_n\ a_n) \end{aligned}$$

must be transformed at compile time to its context-parametric version:

$$\begin{aligned} P'\{\{*\}\} \quad t_1 \dots t_n &= \Lambda C. \sigma \\ P'\{\{k \rightarrow v\}\} \quad t_1 \dots t_n &= \Lambda C. \forall a_1 \dots a_n. (C\ a_1, \dots, C\ a_n) \Rightarrow \\ &((P'\{\{k\}\} \quad a_1 \dots a_n) \diamond []) \rightarrow ((P'\{\{v\}\} \quad (t_1\ a_1) \dots (t_n\ a_n)) \diamond C) \end{aligned}$$

Constraint parametrisation is denoted by Λ -abstraction (with formal parameter C) and constraint application by \diamond .

For example, polyadic type *Size* (Section 6.2.2) is sugar for the following polykinded type:

type $Size\{\{*\}\}\ t \quad = t \rightarrow Int$
type $Size\{\{* \rightarrow v\}\}\ t \quad = \forall a. Size\{\{*\}\}\ a \rightarrow Size\{\{v\}\}\ (t\ a)$

and this is its context-parametric translation:

type $Size'\{\{*\}\}\ t \quad = \Lambda C. t \rightarrow Int$
type $Size'\{\{* \rightarrow v\}\}\ t \quad = \\ \Lambda C. \forall a. (C\ a) \Rightarrow ((Size'\{\{*\}\}\ a) \diamond []) \rightarrow ((Size'\{\{v\}\}\ (t\ a)) \diamond C)$

An instantiation $P\{\{k \rightarrow v\}\}\ T \dots T$ for an actual constrained-type argument T is transformed at compile time to $(P\{\{k \rightarrow v\}\}\ T \dots T) \diamond \Delta$, where Δ is T 's constraint list. The constraints in the list are pushed down to appropriate type variables during expansion, as specified by the rules in Figure 9.

The following derivation shows the rules in action. It involves $Size'$, a type-variable $s :: * \rightarrow *$ and a singleton constraint list with an *Ord* constraint:

$$\begin{aligned} &Size'\{\{* \rightarrow *\}\}\ s \diamond [Ord] \\ &= \{ \text{Definition of } Size' \} \end{aligned}$$

Empty Constraint List:

$$\begin{aligned} &(\Lambda C. \forall a_1 \dots a_n. (C\ a_1, \dots, C\ a_n) \Rightarrow \sigma) \diamond [] \\ &= \\ &\forall a_1 \dots a_n. [] / C \sigma \end{aligned}$$

Empty Constraint:

$$\begin{aligned} &(\Lambda C. \forall a_1 \dots a_n. (C\ a_1, \dots, C\ a_n) \Rightarrow \sigma) \diamond (\emptyset : cs) \\ &= \\ &\forall a_1 \dots a_n. [cs / C] \sigma \end{aligned}$$

Push Constraint:

$$\begin{aligned} &(\Lambda C. \forall a_1 \dots a_n. (C\ a_1, \dots, C\ a_n) \Rightarrow \sigma) \diamond (C : cs) \\ &= \\ &\forall a_1 \dots a_n. (C\ a_1, \dots, C\ a_n) \Rightarrow [cs / C] \sigma \end{aligned}$$

Drop Constraint:

$$(\Lambda C. \tau) \diamond cs = \tau \quad \text{-- if } C \text{ fresh in } \tau$$

Figure 9. Constraint-application rules applied when expanding context-parametric polykinded types.

$$\begin{aligned} &(\Lambda C. \forall a. (C\ a) \Rightarrow \\ &\quad ((Size'\{\{*\}\}\ a) \diamond []) \rightarrow ((Size'\{\{*\}\}\ (s\ a)) \diamond C)) \diamond [Ord] \\ &= \{ \text{Push Constraint} \} \\ &\quad \forall a. Ord\ a \Rightarrow ((Size'\{\{*\}\}\ a) \diamond []) \rightarrow ((Size'\{\{*\}\}\ (s\ a)) \diamond []) \\ &= \{ \text{Definition of } Size' \} \\ &\quad \forall a. Ord\ a \Rightarrow ((\Lambda C. a \rightarrow Int) \diamond []) \rightarrow ((Size'\{\{*\}\}\ (s\ a)) \diamond []) \\ &= \{ \text{Drop Constraint} \} \\ &\quad \forall a. Ord\ a \Rightarrow (a \rightarrow Int) \rightarrow ((Size'\{\{*\}\}\ (s\ a)) \diamond []) \\ &= \{ \text{Definition of } Size' \} \\ &\quad \forall a. Ord\ a \Rightarrow (a \rightarrow Int) \rightarrow ((\Lambda C. s\ a \rightarrow Int) \diamond []) \\ &= \{ \text{Drop Constraint} \} \\ &\quad \forall a. Ord\ a \Rightarrow (a \rightarrow Int) \rightarrow (s\ a \rightarrow Int) \end{aligned}$$

Context-parametric polykinded types can be used with ADTs that have constraints on their parameters, as is the case with ordered sets. Instantiations are simply $(P\{\{k \rightarrow v\}\}\ t \dots t) \diamond \Delta$ where t is a type variable and Δ is the list of constraints associated with the abstract type.