Constructing Polymorphic Programs with Quotient Types

Michael Abbott[1], Thorsten Altenkirch[2], Neil Ghani[1], and Conor McBride[3]

[1] Department of Mathematics and Computer Science, University of Leicester
michael@araneidae.co.uk, ng13@mcs.le.ac.uk
[2] School of Computer Science and Information Technology, Nottingham University
txa@cs.nott.ac.uk
[3] Department of Computer Science, University of Durham
c.t.mcbride@durham.ac.uk

Abstract. The efficient representation and manipulation of data is one of the fundamental tasks in the construction of large software systems. Parametric polymorphism has been one of the most successful approaches to date but, as of yet, has not been applicable to programming with quotient datatypes such as unordered pairs, cyclic lists, bags etc. This paper provides the basis for writing polymorphic programs over quotient datatypes by extending our recently developed theory of containers.

1 Introduction

The efficient representation and manipulation of data is one of the fundamental tasks in the construction of large software systems. More precisely, one aims to achieve amongst other properties: i) abstraction so as to hide implementation details and thereby facilitate modular programming; ii) expressivity so as to uniformly capture as wide a class of data types as possible; iii) disciplined recursion principles to provide convenient methods for defining generic operations on data structures; and iv) formal semantics to underpin reasoning about the correctness of programs. The most successful approach to date has been Hindley-Milner polymorphism which provides predefined mechanisms for manipulating data structures providing they are *parametric* in the data. Canonical examples of such parametric polymorphic functions are the `map` and `fold` operations which can be used to define a wide variety of programs in a structured and easy to reason about manner.

However, a number of useful data types and associated operations are not expressible in the Hindley-Milner type system and this has lead to many proposed extensions including, amongst others, generic programming, dependent types (Altenkirch and McBride, 2003), higher order types (Fiore et al., 1999), shapely types (Jay, 1995), imaginary types (Fiore and Leinster, 2004) and type classes. However, one area which has received less attention is that of quotient types such as, for example, unordered pairs, cyclic lists and the bag type. This is because the problem is fundamentally rather difficult — on the one hand one wants to allow as wide a theory as possible so as to encompass as many quotient types as possible while, on the other hand, one wants to restrict one's definition to derive a well-behaved meta-theory which provides support for key programming paradigms such as polymorphic programming etc. Papers such as Hofmann (1995) have tended to consider quotients of specific types rather than quotients of data structures which are independent of the data stored. As a result, this paper is original in giving a detailed analysis of how to program with quotient data structures in a polymorphic fashion. In particular,

- We provide a syntax for declaring quotient datatypes which encompasses a variety of examples. This syntax is structural which we argue is essential for any theory of polymorphism to be applicable.
- We show how the syntax of such a declaration gives rise to a quotient datatype.
- We provide a syntax for writing polymorphic programs between these quotient datatypes and argue that these programs do indeed deserve to be called polymorphic.
- We show that every polymorphic function between our quotient datatypes is represented uniquely by our syntax. That is, our syntax captures all polymorphic programs in a unique manner.

To execute this program of research we extend our work on container datatypes (Abbott, 2003; Abbott et al., 2003a,b). Container types represent types via a set of shapes and locations in each shape where data may be stored. They are therefore like Jay's shapely types (Jay, 1995) but more general as we discuss later. In previous papers cited above, we have shown how these container types are closed under a wide variety of useful constructions and can also be used as a framework for generic programming, eg they support a generic notion of differentiation (Abbott et al., 2003b) which derives a data structure with a hole from a data structure.

This paper extends containers to cover quotient datatypes by saying that certain labellings of locations with data are equivalent to others. We call these structures quotient containers. As such they correspond to the step from normal functors to analytic functors in Joyal (1986). However, our quotient containers are more general than analytic functors as they allow infinite sets of positions to model coinductive datatypes. In addition, our definition of the morphisms between quotient containers is new as is all of their applications to programming. In addition, while pursuing the above program, we also use a series of running examples to aid the reader. We assume only the most basic definitions from category theory like category, functor and natural transformations. The exception is the use of left Kan extensions for which we supply the reader with the two crucial properties in section 2. Not all category theory books contain information on these constructions, so the reader should use Mac Lane (1971); Borceux (1994) as references.

The paper is structured as follows. In section 2 we recall the basic theory of containers, container morphisms and their application to polymorphic programming. We also discuss the relationship between containers and shapely types. In section 3 we discuss how quotient datatypes can be represented in container theoretic terms while in section 4 we discuss how polymorphic programs between quotient types can be represented uniquely as morphisms between quotient containers. We conclude in section 5 with some conclusions and proposals for further work.

2 A Brief Summary of Containers

Notation: We write $\mathbb{N}$ for the set of natural numbers and if $n \in \mathbb{N}$, we write $\underline{n}$ for the set $\{0,\ldots,n-1\}$. We assume the basic definitions of category theory and if $f : X \to Y$ and $g : Y \to Z$ are morphisms in a category, we write their composite $g \circ f : X \to Z$ as is standard categorical practice. We write K_1 for the constantly 1 valued functor from any category to **Sets**. If A is a set and B is an A indexed family of sets, we write $\sum a : A.\ B(a)$ for the set $\{(a,b) \mid a \in A, b \in B(a)\}$. We write ! for the empty map from the empty set to any other set. Injections into the coproduct are written inl and inr.

This paper uses left Kan extensions to extract a universal property of containers which is not immediately visible. We understand that many readers will not be familiar with these structures so we supply all definitions and refer the reader to Mac Lane (1971) for more details. Their use is limited to a couple of places and hence doesn't make the paper inaccessible to the non-cogniscenti. Given a functor $I : \mathscr{A} \to \mathscr{B}$ and a category $\mathscr{C}$, precomposition with I defines a functor $\_ \circ I : [\mathscr{B},\mathscr{C}] \to [\mathscr{A},\mathscr{C}]$. The problem of left Kan extensions is the problem of finding a left adjoint to $\_ \circ I$. More concretely, given a functor $F : \mathscr{A} \to \mathscr{C}$, the left Kan extension of F along I is written $\mathsf{Lan}_I F$ defined via the natural isomorphism

$$[\mathscr{B},\mathscr{C}](\mathsf{Lan}_I F, H) \cong [\mathscr{A},\mathscr{C}](F, H \circ I) \tag{1}$$

One can use the following coend formula to calculate the action of a left Kan extension when $\mathscr{C} = \mathbf{Sets}$ and $\mathscr{A}$ is small

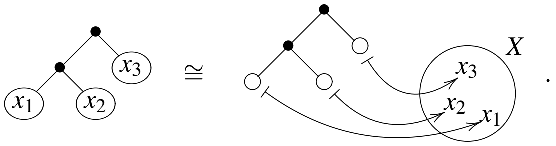
$$(\mathsf{Lan}_I F)X \;=\; \int^{A \in \mathscr{A}} \mathscr{B}(IA, X) \times FA \tag{2}$$

What are Containers? Containers capture the idea that concrete datatypes consist of memory locations where data can be stored. For example, any element of the type of lists $\mathsf{List}(X)$ of X can be uniquely written as a natural number n given by the length of the list, together with a function $\{0,\ldots,n-1\} \to X$ which labels each position within the list with an element from X. Thus we may write

$$\mathsf{List}(X) \;\equiv\; \sum n : \mathbb{N}.\ \{0,\ldots.n-1\} \to X \tag{3}$$

We may think of the set $\{0,\ldots,n-1\}$ as n memory locations while the function f attaches to these memory locations, the data to be stored there. Similarly, any binary tree

tree can be uniquely described by its underlying shape (which is obtained by deleting the data stored at the leaves) and a function mapping the positions in this shape to the data thus:



More generally, we are led to consider datatypes which are given by a set of shapes S and, for each $s \in S$, a set of positions $P(s)$ which we think of as locations in memory where data can be stored. This is precisely a container

Definition 2.1 (Container). *A container $(S \triangleright P)$ consists of a set S and, for each $s \in S$, a set of positions $P(s)$.*

Of course, in general we do not want to restrict ourselves to the category of sets since we want our theory to be applicable to domain theoretic models. Rather, we would develop our theory over locally cartesian closed categories (Hofmann, 1994), certain forms of fibrations such as comprehension categories (Jacobs, 1999) or models of Martin-Löf type theory — see our previous work (Abbott, 2003; Abbott et al., 2003a,b) for such a development. However, part of the motivation for this paper was to make containers accessible to the programming community where we believe they provide a flexible platform for supporting generic forms of programming. Consequently, we have deliberately chosen to work over **Sets** so as to enable us to get our ideas across without an overly mathematical presentation.

As suggested above, lists can be presented as a container

Example 2.2 *The list type is given by the container with shapes given by the natural numbers $\mathbb{N}$ and, for $n \in \mathbb{N}$, define the positions $P(n)$ to be the set $\{0, \ldots, n-1\}$.*

To summarise, containers are our presentations of datatypes in the same way that `data` declarations are presentations of datatypes in Haskell. The semantics of a container is an endofunctor on some category which, in this paper, is **Sets**. This is given by

Definition 2.3 (Extension of a Container). *Let $(S \triangleright P)$ be a container. Its semantics, or extension, is the functor $T_{S \triangleright P}$: **Sets** $\rightarrow$ **Sets** defined by*

$$T_{S \triangleright P}(X) = \sum s : S.\ \ (P(s) \rightarrow X)$$

An element of $T_{S \triangleright P}(X)$ is thus a pair (s, f) where $s \in S$ is a shape and $f : P(s) \rightarrow X$ is a labelling of the positions over s with elements from X. Note that $T_{S \triangleright P}$ really is a functor since its action on a function $g : X \to Y$ sends the element (s, f) to the element $(s, g \circ f)$. Thus for example, the extension of the container for lists is the functor mapping X to

$$\sum n : \mathbb{N}.\ \{0, \ldots, n-1\} \to X \ .$$

As we commented upon in equation 3, this is the list functor.

The theory of containers was developed in a series of recent papers (Abbott, 2003; Abbott et al., 2003a,b) which showed that containers encompass a wide variety of types as they are closed under various type forming operations such as sums, products, constants, fixed exponentiation, (nested) least fixed points and (nested) greatest fixed points. Thus containers encapsulate a large number of datatypes. So far, we have dealt with containers in one variable whose extensions are functors on **Sets**. The extension to n-ary containers, whose extensions are functors $\mathbf{Sets}^n \to \mathbf{Sets}$, is straightforward. Such containers consist of a set of shapes S, and for each $s \in S$ there are n position sets $P_n(s)$. See the above references for details.

We finish this section with a more abstract presentation of containers which will be used to exhibit the crucial universal property that they satisfy. This universal property underlies the key result about containers. First, note that the data in a container $(S \triangleright P)$ can be presented as a functor $P : S \to \mathbf{Sets}$ where here we regard the set S as a discrete category and P maps each s to $P(s)$. In future, we will switch between these two views of a container at will. The semantic functor $T_{S \triangleright P}$ has a universal property given by the following lemma.

Lemma 2.4. *Let $P : S \to \mathbf{Sets}$ be a container. Then $T_{S \triangleright P}$ is the left Kan extension of K_1 along P*

$$\begin{tikzcd} S \arrow[r, "P"] \arrow[d, "K_1"'] & \mathbf{Sets} \arrow[dl, "T_{S \triangleright P} = \mathsf{Lan}_P K_1"] \\ \mathbf{Sets} & \end{tikzcd}$$

Proof We calculate as follows

$$(\mathsf{Lan}_P K_1)X \ = \ \int^{s:S} \mathbf{Sets}(Ps, X) \times K_1 s \ = \ \sum s : S.\ \mathbf{Sets}(Ps, X) \ = \ T_{S \triangleright P}(X)$$

where the first equality is the classic coend formula for left Kan extensions of equation 2, the second equality holds as S is a discrete category and 1 is the unit for the product, and the last equality is the definition of $T_{S \triangleright P}(X)$. $\square$

As we shall see, this universal property will be vital to our representation theorem.

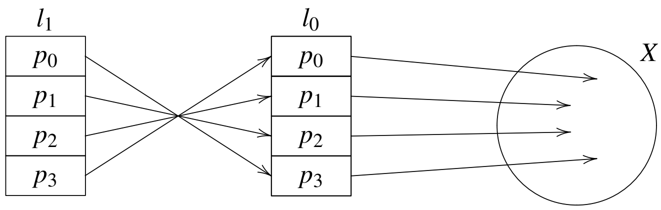
2.1 Container Morphisms

Containers are designed for implementation. Thus, we imagine defining lists in a programming language by writing something like

$$data \;\; List = (n : \mathbb{N} \triangleright \underline{n})$$

although the type dependency means we need a dependently typed language. If we were to make such declarations, how should we program? The usual definitions of lists based upon initial algebras or final coalgebras give rise naturally to recursive forms of programming. As an alternative, we show that all polymorphic functions between containers are captured uniquely by *container morphisms*.

Consider first the reverse function applied to a list written in container form (n, g). Its reversal must be a list and hence of the form (n', g'). In addition, n' should only depend upon n since reverse is polymorphic and hence shouldn't depend upon the actual data in the list given by g. Thus there is a function $\mathbb{N} \to \mathbb{N}$. In the case of reverse, the length of the list doesn't change and hence this is the identity. To define g' which associates to each position in the output a piece of data, we should first associate to each position in the input a position in the input and then look up the data using g. Pictorially we have:



Here we start with a list l_0 which has 4 positions and a labelling function g into X. The result of reversing l_0 is the list l_1 with 4 positions and labelling function given by the above composite. In general, we therefore define

Definition 2.5 (Container Morphisms). *A morphism $(A \triangleright B) \to (C \triangleright D)$ consists of a pair (u, f) where $u : A \to C$ and an A-indexed family of maps $f_a : D(ua) \to B(a)$. The category of containers and container morphisms is written* **Cont**

Example 2.6 *The reverse program is given by the container morphism (Id, f) where the function $f_n : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$ is defined by $f_n(i) = n - i - 1$*

We stress that this would be the actual definition of reverse in a programming language based upon containers. The process of translating other, more abstract and higher level, definitions of reverse into the above form indicates the potential use of containers as an optimisation tool. Note also how f_n says that the data stored at the $i'th$ cell after reversing a list is that data stored at the $n - i - 1$'th cell in the input list.

Consider the tail function tail:: $\mathsf{List(X)} \to 1 + \mathsf{List(X)}$. The shapes of the datatype $1 + \mathsf{List}(X)$ is $1 + \mathbb{N}$ with the shapes above $\mathsf{inl}(*)$ being empty while the shapes above $\mathsf{inr}(n)$ is the set $\{0, \ldots, n-1\}$. We therefore write this container as $(1 + n : \mathbb{N} \triangleright 0 + \underline{n})$.

Example 2.7 *The tail function* (u, f) *is given by the container morphism* $(n : \mathbb{N} \triangleright \underline{n}) \to (1 + n : \mathbb{N} \triangleright 0 + \underline{n})$ *defined by*

$$u(0) = \mathsf{inl}(*) \qquad u(n+1) = \mathsf{inr}(n)$$

and with $f_0 = !$ *and* $f_{n+1} : \underline{n} \to \underline{n+1}$ *defined by* $f_{n+1}(i) = i + 1$.

Thus the i'th cell in the output of a nonempty list comes from the $i+1$'th cell in the input list. Readers may check their understanding at this point by wondering what function is defined by setting $f_{n+1}(i) = i$ in the above example. We finish this section with two final points. First, a categorical re-interpretation of a container morphism analogous to the categorical interpretation of a container $A \triangleright B$ as a presheaf $B : A \to \mathbf{Sets}$ as in lemma 2.4.

Lemma 2.8. *A morphism of containers* $(u, f) : (A \triangleright B) \to (C \triangleright D)$ *is given by a functor* $u : A \to C$ *and a natural transformation* $f : Du \to B$. *Pictorially, this can be represented:*

$$\begin{tikzcd} A \arrow[rr, "u"] \arrow[dr, "B"'] & \arrow[d, phantom, "\leftarrow f"] & C \arrow[dl, "D"] \\ & \mathbf{Set} & \end{tikzcd}$$

Proof Since A and C are discrete categories, a functor is just a function. Since A is discrete, the natural transformation is just a family of maps of the given form. $\square$

Finally, containers are more than just a programming tool — they can also simplify reasoning. For example, consider proving that $\mathsf{reverse} \circ \mathsf{reverse}$ is the identity. Using the usual recursive definition one soon runs into problems and must strengthen the inductive hypothesis to reflect the interaction of reverse and append. Using the container definition, this problem is trivial as we can reason as follows: $(Id, f) \circ (Id, f) = (Id, f \circ f) = (Id, Id)$ since, for each n, the function f_n is clearly idempotent.

2.2 From Container Morphisms to Polymorphic Functions and Back

Given a container morphism $(u, f) : (A \triangleright B) \to (S \triangleright P)$ does (u, f) really define a polymorphic function $T_{A \triangleright B} \to T_{S \triangleright P}$? If so, are all polymorphic functions of this form?. And in a unique manner? To answer these questions we have to describe mathematically what a polymorphic function is. In the theory of program language semantics, covariant datatypes are usually represented as functors while polymorphic functions between covariant datatypes are represented by natural transformations (Bainbridge et al., 1990). Other representations of polymorphic functions are as terms in various polymorphic lambda calculi and via the theory of logical relations. Various theoretical results show that these are equivalent so in the following we take a polymorphic function to be a natural transformation.

Our key theorem is the following which ensures that our syntax for defining polymorphic functions as container morphisms is flexible enough to cover all polymorphic functions.

Theorem 2.9. *Container morphisms* $(A \triangleright B) \to (C \triangleright D)$ *are in bijection with natural transformations* $T_{A \triangleright B} \to T_{C \triangleright D}$*. Formally,* $T : \mathbf{Cont} \to [\mathbf{Sets}, \mathbf{Sets}]$ *is full and faithful.*

Proof The proof is a special case of Theorem 4.3. Alternatively, see Abbott et al. (2003a); Abbott (2003) $\square$

Containers vs Shapely Types: In Jay and Cockett (1994) and Jay (1995) *shapely types* (in one parameter) in **Sets** are pullback preserving functors $F : \mathbf{Sets} \to \mathbf{Sets}$ equipped with a cartesian natural transformation to the list functor. This means there are natural maps $FX \to \mathsf{List}(X)$ which extract the data from an element of FX and place it in a list thereby obtaining a decomposition of data into shapes and positions similar to what occurs in containers.

Note however that the positions in a list have a notion of order and hence a shapely type is also equipped with an order on positions. Typically, when we declare a datatype we do not want to declare such an ordering over it and, indeed, in the course of programming we may wish to traverse a data structure with different orders. At a more theoretical level, by reducing datatypes to lists, a classification theorem such as Theorem 2.9 would reduce polymorphic functions to polymorphic functions between lists but would not be able to classify what these are. Containers do not impose such an order and instead reduce datatypes to the more primitive idea of a family of positions indexed by shapes.

3 Containers and Quotient Types

The purpose of this paper is to extend our previous results on containers to cover quotient datatypes and polymorphic functions between them. In particular we want to

- Generalise the notion of container to cover as many quotient datatypes as possible and thereby derive a syntax for declaring quotient datatypes.

- Define a notion of morphism between such generalised containers and thereby derive a syntax for polymorphic programming with quotient types.
- Prove the representation theorem for these polymorphic programs thereby proving that the quotient container morphisms really are polymorphic functions and that all such polymorphic functions are captured by quotient container morphisms.

3.1 Unordered Pairs and Cyclic Lists

We begin with an example of unordered pairs. Note first that the type $X \times X$ is given by the container $(1 \triangleright 2)$. These are ordered pairs of elements of X. The type of unordered pairs of X is written as $X \otimes X$ and is defined as $X \times X/\sim$ where $\sim$ is the equivalence relation defined by

$$\langle x,y\rangle \sim \langle y,x\rangle$$

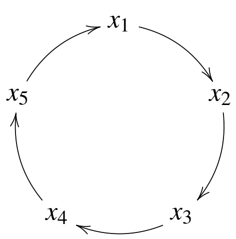
Recall our analysis of the pair type was as one shape, containing two positions $(1 \triangleright 2)$. Lets call these positions p_1 and p_2. An element of the pair type then consists of a labelling for these positions, ie a function $f : \{p_1, p_2\} \to X$ for a set X. To move from ordered pairs to unordered pairs is exactly to note that the labelling f should be regarded the same as the labelling $f \circ \mathsf{swap}$ where $\mathsf{swap} : \{p_1, p_2\} \to \{p_1, p_2\}$ is the function sending p_1 to p_2 and p_2 to p_1. Thus

Example 3.1 *The type of unordered pairs of elements of X is given by*

$$(\{p_1, p_2\} \to X)/\sim$$

where $\sim$ is the equivalence relation on $\{p_1, p_2\} \to X$ obtained by setting $f \circ \mathsf{swap} \sim f$.

Let's cement our intuitions by doing another example. A cyclic list is a list with no starting or ending point. Here is a cyclic list of length 5



Can we represent cyclic lists in the same style as we used for unordered pairs? Recall from equation 3 that lists were given by $\mathsf{List}(X) \equiv \sum n : \mathbb{N}.\ \{0, \ldots ..n-1\} \to X$. Now, in a cyclic list of length n, a labelling $f : \{0, \ldots ..n-1\} \to X$ should be equivalent to the labelling $f \circ \lambda i.(i+k) \bmod n$ where $k \in \{0, \ldots, n-1\}$. Thus we may define

Example 3.2 *The type of cyclic lists of elements of X is given by*

$$\mathsf{CList}(X) \equiv \sum n{:}\mathbb{N}.\ (\{0,\ldots.n-1\} \to X)/\sim_n$$

where $\sim_n$ is the equivalence relation on $\{0,\ldots.n-1\} \to X$ obtained by setting $f \sim_n f \circ \lambda i.(i+k) \bmod n$ where $k \in \{0,\ldots,n-1\}$.

The observant reader will have spotted that, in example 3.2, there is actually an equivalence relation for each shape. Examples 3.1 and 3.2 exemplify the kind of structures we wish to compute with, ie the structures for which we want to find a clean syntax supporting program construction and reasoning. In general they consist of a container as defined before with an equivalence relation on labellings of positions. To ensure the equivalence relation is structural, ie independent of data as one would expect in a polymorphic setting, the equivalence relation is defined by identifying a labelling $f : P(s) \to X$ with the labelling $f \circ \alpha$ where α is one of a given set of isomorphisms on $P(s)$. Hence we define

Definition 3.3 (Quotient Containers). *A quotient container $(S \triangleright P/G)$ is given by a container $(S \triangleright P)$ and, for each shape $s \in S$, a set $G(s)$ of isomorphisms of $P(s)$ closed under composition, inverses and containing the identity.*

Thus a quotient container has an underlying container and, every container as in Definition 2.1 is a quotient container with, for each shape s, the set $G(s)$ containing only the identity isomorphism on $P(s)$. Another way of describing the isomorphisms in a quotient container $(S \triangleright P/G)$ is to say that for each $s \in S$, $G(s)$ is a subgroup of the automorphism, or permutation, group on $P(s)$.

Often we present a quotient container $(S \triangleright P/G)$ by defining, for each shape $s \in S$, the group $G(s)$ to be the smallest group containing a given set. However, the advantage of requiring $G(s)$ to be a group is that if we define $f \sim_G f'$ iff there is a $g \in G(s)$ such that $f = f' \circ g$, then $\sim_G$ is automatically an equivalence relation and so we don't have to consider its closure. A more categorical presentation of quotient containers reflecting the presentation of containers used in lemma 2.4 is the following.

Lemma 3.4. *A quotient container is exactly a functor $P : S \to$ **Sets** where every morphism of S is both an endomorphism and an isomorphism.*

Proof Given a quotient container $(S \triangleright P/G)$, we think of S as the category with objects elements $s \in S$ and, as endomorphisms of s, the set $G(s)$. The functor P is the obvious functor mapping s to $P(s)$. □

Given a quotient container, $(S \triangleright P/G)$, of course we want to calculate the associated datatype or functor $T_{S \triangleright P/G}$: **Sets** → **Sets**. As with the presentation of containers we do this concretely and then more abstractly to uncover a hidden universal property.

Definition 3.5 (Extension of a Quotient Container). *Given a quotient container, say $(S \triangleright P/G)$, its extension is the functor $T_{S \triangleright P/G} : \mathbf{Sets} \to \mathbf{Sets}$ defined by*

$$T_{S \triangleright P/G}(X) = \sum s:S.\ \ (P(s) \to X)/\sim_s$$

where $\sim_s$ is the equivalence relation on the set of functions $P(s) \to X$ defined by $f \sim_s f'$ if there is a $g \in G(s)$ such that $f' = f \circ g$.

So a quotient container is like a container except that in the datatype it gives rise to, a labelling f of positions with data is defined to be the same as the labelling $f \circ g$ obtained by bijectively renaming the positions using $g \in G(s)$ and then performing the labelling f. The more abstract formulation is given by the next lemma which uses lemma 3.4 to regard a quotient container $(S \triangleright P/G)$ as a presheaf $P : S \to \mathbf{Sets}$.

Lemma 3.6. *Let $P : S \to \mathbf{Sets}$ be a quotient container. Then $T_{S \triangleright P/G}$ is the left Kan extension of K_1 along P*

$$\begin{tikzcd}
S \arrow[r, "P"] \arrow[d, "K_1"'] & \mathbf{Set} \arrow[dl, "T_{S \triangleright P/G} = \mathsf{Lan}_P K_1"] \\
\mathbf{Set} &
\end{tikzcd}$$

Proof We can calculate the left Kan extension as follows

$$\begin{aligned}
(\mathsf{Lan}_P K_1)X &\cong \int^{s:S} \mathbf{Sets}(Ps, X) \times K_1 s \\
&\cong \sum s:S.\ \ \mathbf{Sets}(Ps, X)/\sim_s \\
&= T_{S \triangleright P/G}(X)
\end{aligned}$$

where by first equality the classic coend formula of equation 2, the second is the reduction of coends to colimits and the third is the definition of $T_{S \triangleright P/G}$. This is because, the equivalence relation $\sim_s$ in the coend has $f \sim_s f'$ iff there is a $g : s \to s$ such that $f' = f \circ P(g)$ where $f, f' : P(s) \to X$. This is exactly the definition of the extension of a quotient container. $\square$

The theory of containers thus generalises naturally to quotient containers as the same formula of left Kan extension calculates both the semantics of a container and that of a quotient container.

We finish this section on the presentation of quotient types with the example of finite bags (also called multisets) as promised. Bags of other sizes are of course a straightforward generalisation. Given this remark, we henceforth refer to finite bags as simply bags. The key intuition is that a bag is like a set but elements may have multiple occurrences - one may thus define a bag as $\mathsf{Bag}(X) = X \to \mathbb{N}$. By putting all the elements of a bag in a list we get a representation of the bag but of course there are many such representations since there is no order of elements in the bag but there is in a list. Thus we get a bijection between bags and lists quotiented out by all rearrangements of positions. Hence

Example 3.7 *The bag type is the quotient container $(S \triangleright P/G)$ where $(S \triangleright P)$ is the container for lists and, for $n : \mathbb{N}$, we take $G(n)$ to be the set of all isomorphisms on the set $\{0, \ldots, n-1\}$.*

4 Programming with Quotient types

We have identified a class of quotient data structures and axiomatised them as quotient containers. These quotient containers can be seen as datatype declarations in a programming language and so we now ask how we program polymorphically with these quotient containers.

Recall a container morphism $(S \triangleright P) \to (Q \triangleright R)$ consisted of a translation of shapes $u : S \to Q$ and, for each $s \in S$, a map $f_s : R(us) \to P(s)$ sending positions in the output to positions in the input. If we now ask what is a map of quotient containers $(S \triangleright P/G) \to (Q \triangleright R/H)$ its reasonable to require a map $(u, f) : (S \triangleright P) \to (Q \triangleright R)$ of the underlying containers which takes into account the respective quotients. There are two issues:

- Since the maps $f_s : R(us) \to P(s)$ are labellings, the quotient given by H says that a quotient container morphism (u, f) is the same as another quotient container morphism (u, f') if for each $s \in S$, there is an $h_s \in H(us)$ such that $f_s = f'_s \circ h_s$.
- Given a map $f_s : R(us) \to P(s)$ and a $g \in G(s)$ then the labellings f and $g \circ f$ should be regarded as equal as labellings of $R(us)$. Hence there should be an $h_g \in H(us)$ such that $f \circ h_g = g \circ f$.

Hence we define

Definition 4.1 (Quotient Container Morphism). *A pre-morphism of quotient containers $(S \triangleright P/G) \to (Q \triangleright R/H)$ is a morphism of the underlying containers $(u, f) : (S \triangleright P) \to (Q \triangleright R)$ such that for each $s \in S$ and each $g \in G(s)$, there is an $h_g \in H(us)$ such that*

$$\begin{tikzcd} R(us) \arrow[r, "f_s"] \arrow[d, "h_g"'] & P(s) \arrow[d, "g"] \\ R(us) \arrow[r, "f_s"'] & P(s) \end{tikzcd}$$

The morphisms $(S \triangleright P/G) \to (Q \triangleright R/H)$ are the premorphisms quotiented by the equivalence relation

$$(u, f) \sim (u, f') \text{ iff for all } s \in S \text{, there exists } h_s \in H(us) \text{ such that } f_s = f'_s \circ h_s$$

Intuitively, the first condition is precisely the naturality of the quotient container morphism while the second reflects the fact that labellings are defined upto quotient. Is this a good definition of a polymorphic program between quotient containers? We answer this in two ways. On a theoretical level we show that all polymorphic programs can be uniquely captured by such quotient container morphisms while on a practical level we demonstrate a number of examples.

Lemma 4.2. *The quotient container morphisms $(S \triangleright P/G) \to (Q \triangleright R/H)$ are in one-to-one bijection with natural transformations $K_1 \to T_{Q \triangleright R/H} P$ where in the latter we regard $P : S \to$ **Sets** as a presheaf as described in lemma 2.4.*

Proof Such natural transformations are exactly S-indexed maps $K_1(s) \to T_{Q \triangleright R/H} P(s)$ which are natural in S. Thus we have, for each $s \in S$, maps

$$1 \quad \to \quad \sum q : Q.\ \ (R(q) \to P(s)) / \sim_q$$

which are natural in S. Such a family of S-indexed maps is exactly a map $u : S \to Q$ and an S-indexed family of elements of $(R(us) \to P(s)) / \sim_{us}$ natural in S. An element of $(R(us) \to P(s)) / \sim_{us}$ is clearly an equivalence class of functions while the naturality corresponds to the commuting diagram above. □

Theorem 4.3. *Quotient container morphisms are in one-to-one bijection with natural transformations between their extensions. Hence there is a full and faithful embedding T : **QCont** $\to$ [**Sets**, **Sets**].*

Proof Natural transformations $T_{S \triangleright P/G} \to T_{Q \triangleright R/H}$ are by lemma 3.6 exactly natural transformations $\mathsf{Lan}_P K_1 \to \mathsf{Lan}_R K_1$. By the universal property of Kan extensions given in equation 1, these are in one-to-one bijection with natural transformations $K_1 \to T_{Q \triangleright R/H} P$ which by lemma 4.2 are in one-to-one bijection with quotient container morphisms $(S \triangleright P/G) \to (Q \triangleright R/H)$. □

Notice the key role of the left Kan extension here. It identifies a universal property of the extension of a quotient container which is exactly what is required to prove Theorem 4.3. Also note that Theorem 2.9 is a corollary as there is clearly a full and faithful embedding of **Cont** into **QCont**.

We now finish with some examples. Firstly, if $(S \triangleright P/G)$ is a container and for each $s \in S$ the group $G(s)$ is a subgroup of $H(s)$, then there is a morphism of quotient containers $(S \triangleright P/G) \to (S \triangleright P/H)$. Thus

Example 4.4 *The canonical maps* $\mathsf{List}(X) \to \mathsf{CList}(X) \to \mathsf{Bag}(X)$ *are polymorphic as, for a given* $n \in \mathbb{N}$*, they arise as container morphisms from the inclusions of the singleton group into the group of functions* $\{\lambda i.i + k \,\mathsf{mod}\, n | k \in \{0, \ldots, n-1\}\}$ *and of this group into the group of all isomorphisms on* $\{0, \ldots, n-1\}$.

Example 4.5 *Every datatype given by a quotient container* $(S \triangleright P/G)$ *has a map operation which is given by the action of functor* $T_{S \triangleright P/G}$ *on morphisms.*

Example 4.6 *We can extend the operation of reversing a list to a reverse on cyclic lists. One simply needs to check the commutation condition that for each size n and* $k \in \{0, \ldots, n-1\}$*, there is a* k' *such that*

$$\lambda i.(i+k) \,\mathsf{mod}\, n \;\circ\; \lambda i.n - i - 1 \;=\; \lambda i.n - i - 1 \;\circ\; \lambda i.(i+k') \,\mathsf{mod}\, n$$

Of course we simply take $k' = n - k - 1$.

In general this shows how simple our theory is to apply in practice. We simply have one condition to check!

5 Conclusions and Further Work

We have provided a synthesis of polymorphism with quotient datatypes in such a way as to facilitate programming and reasoning with such quotient structures. This work is based upon an extension of our previous work on containers which, going further than shapely types, present datatypes via a collection of shapes and, for each shape, a collection of positions where data may be stored. The treatment of quotient datatypes is structural in that the quotient is determined by a collection of isomorphisms on these position sets which induce a quotient on the labellings of positions with data. On top of this presentation of datatypes, we have provided a means for programming with such structures by defining morphisms of quotient containers. These are essentially morphisms of the underlying containers which respect the relevant quotients. This simple axiomatisation is proven correct in that we show that these morphisms determine exactly the polymorphic programs between these quotient data structures.

As for further work, we believe containers are an excellent platform for generic program and we wish to develop this application of containers. With specific relationship to these quotient containers, we have begun investigating their application to programming in the Theory of Species which was Joyal's motivation for developing analytic functors. From a more practical perspective, we would like to increase the examples of programs covered by both containers and quotient containers to include, for example, searching and sorting algorithms. Note such programs are not strictly polymorphic as their result depends upon inspection of the data. Of course this can already be done concretely by accessing the data in a container via the labellings. However, a greater challenge is to describe the degree to which such algorithms are polymorphic. In Haskell, one uses type classes so we would be looking for a container-theoretic approach to type classes.

Bibliography

M. Abbott. *Categories of Containers*. PhD thesis, University of Leicester, 2003.

M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In A. Gordon, editor, *Proceedings of FOSSACS 2003*, number 2620 in Lecture Notes in Computer Science, pages 23–38. Springer-Verlag, 2003a.

M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. Derivatives of containers. In *Typed Lambda Calculi and Applications, TLCA 2003*, number 2701 in Lecture notes in Computer Science, pages 16–30. Springer, 2003b.

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Generic Programming*, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming.

E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism, preliminary report. In G. Huet, editor, *Logical Foundations of Functional Programming*, chapter 14, pages 315–327. Addison-Wesley, 1990.

F. Borceux. *Handbook of Categorical Algebra*. Encyclopedia of Mathematics. CUP, 1994.

M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding (extended abstract). In *Proc. 14$^{\text{th}}$ LICS Conf.*, pages 193–202. IEEE, Computer Society Press, 1999.

M. P. Fiore and T. Leinster. Objects of categories as complex numbers. *Advances in Mathematics.*, 2004. To appear.

M. Hofmann. On the interpretation of type theory in locally cartesian closed categories. In *CSL*, pages 427–441, 1994.

M. Hofmann. A simple model of quotient types. volume 902 of *Lecture Notes in Computer Science*, pages 216–234. Springer, 1995.

B. Jacobs. *Categorical Logic and Type Theory*. Number 141. Elsevier, 1999.

C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

C. B. Jay and J. R. B. Cockett. Shapely types and shape polymorphism. In *Proceedings of ESOP'94*, Lecture Notes in Computer Science, pages 302–316. Springer, 1994.

A. Joyal. Foncteurs analytiques et espéces de structures. In *Combinatoire énumérative*, number 1234 in LNM, pages 126 – 159. 1986.

S. Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer-Verlag, 1971.