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 a fundamental task 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 a fundamental task in the construction of large software systems. More precisely, one aims to achieve: 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 (4), higher order types (5) shapely types (8), imaginary types (11) 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 ones definition to derive a well behave meta-theory which provides support for key programming paradigms such as polymorphic programming etc.

This paper is, as far as we know, the first paper to give a detailed analysis of how to program with quotient types in a polymorphic fashion. In particular we i) 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; ii) show how the syntax of such a declaration gives rise to a quotient datatype; iii) provide a syntax for writing polymorphic programs between these quotient datatypes and argue that these programs do indeed deserve to be called polymorphic; and iv) 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 (1; 2; 3). 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 (8) but, as commented upon in the above references, are more primitive in not relying on lists. 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. (3).

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's work (9). 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 of category, functor and natural transformations. The exception is the use of left Kan extensions for which we supply the reader with the required information.

The paper is structured as follows. In section 2 we briefly recall the basic theory of containers, container morphisms and their application to polymorphic programming. 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 History of Containers

Notation: We write $\mathbb{N}$ or 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 $\Sigma a : A.B(a)$ 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.