Dexter Kozen (Ed.)

LNCS 3125

Mathematics of Program Construction

7th International Conference, MPC 2004
Stirling, Scotland, UK, July 2004
Proceedings

Springer

Dexter Kozen (Ed.)

Mathematics of Program Construction

7th International Conference, MPC 2004
Stirling, Scotland, UK, July 12-14, 2004
Proceedings

Springer

Volume Editor

Dexter Kozen
Cornell University
Department of Computer Science
Ithaca, NY 14853-7501, USA
E-mail: kozen@cs.cornell.edu

Library of Congress Control Number: 2004108032

CR Subject Classification (1998): F.3, F.4, D.2, F.1, D.3

ISSN 0302-9743
ISBN 3-540-22380-0 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11019305 06/3142 5 4 3 2 1 0

Preface

This volume contains the proceedings of MPC 2004, the Seventh International Conference on the Mathematics of Program Construction. This series of conferences aims to promote the development of mathematical principles and techniques that are demonstrably useful in the process of constructing computer programs, whether implemented in hardware or software. The focus is on techniques that combine precision with conciseness, enabling programs to be constructed by formal calculation. Within this theme, the scope of the series is very diverse, including programming methodology, program specification and transformation, programming paradigms, programming calculi, and programming language semantics.

The quality of the papers submitted to the conference was in general very high, and the number of submissions was comparable to that for the previous conference. Each paper was refereed by at least four, and often more, committee members.

This volume contains 19 papers selected for presentation by the program committee from 37 submissions, as well as the abstract of one invited talk: *Extended Static Checking for Java* by Greg Nelson, Imaging Systems Department, HP Labs, Palo Alto, California.

The conference took place in Stirling, Scotland. The previous six conferences were held in 1989 in Twente, The Netherlands; in 1992 in Oxford, UK; in 1995 in Kloster Irsee, Germany; in 1998 in Marstrand near Göteborg, Sweden; in 2000 in Ponte de Lima, Portugal; and in 2002 in Dagstuhl, Germany. The proceedings of these conferences were published as LNCS 375, 669, 947, 1422, 1837, and 2386, respectively.

Three other international events were co-located with the conference: the Tenth International Conference on Algebraic Methodology And Software Technology (AMAST 2004), the Sixth AMAST Workshop on Real-Time Systems (ARTS 2004), and the Fourth International Workshop on Constructive Methods for Parallel Programming (CMPP 2004). We thank the organizers of these events for their interest in sharing the atmosphere of the conference.

May 2004 Dexter Kozen

Acknowledgments

We are grateful to the members of the program committee and their referees for their care and diligence in reviewing the submitted papers. We are also grateful to the sponsoring institutions and to the FACS Specialist Group. Finally, we would like to extend a special thanks to Kelly Patwell for all her hard work with the organization of the conference.

Program Committee

Roland Backhouse (UK)	K. Rustan M. Leino (USA)
Stephen Bloom (USA)	Hans Leiss (Germany)
Eerke Boiten (UK)	Christian Lengauer (Germany)
Jules Desharnais (Canada)	Lambert Meertens (USA)
Thorsten Ehm (Germany)	Bernhard Moeller (Germany)
Jeremy Gibbons (UK)	David Naumann (USA)
Ian Hayes (Australia)	Alberto Pardo (Uruguay)
Eric Hehner (Canada)	Georg Struth (Germany)
Johan Jeuring (The Netherlands)	Jerzy Tiuryn (Poland)
Dexter Kozen (USA, chair)	Mark Utting (NZ)

Sponsoring Institutions

The generous support of the following institutions is gratefully acknowledged.

Cornell University
University of Stirling
Formal Aspects of Computing Science (FACS) Specialist Group

External Referees

All submitted papers were carefully reviewed by members of the program committee and the following external referees, who produced extensive review reports that were transmitted to the authors. We apologize for any omissions or inaccuracies.

Viviana Bono	Wim H. Hesselink	Steve Reeves
Ana Bove	Marcin Jurdzinski	Andreas Schäfer
David Carrington	Stefan Kahrs	Luis Sierra
Maximiliano Cristiá	Zhiming Liu	Michael Anthony Smith
Sharon Curtis	Clare Martin	Jerzy Tyszkiewicz
Colin Fidge	Diethard Michaelis	Pawel Urzyczyn
Marcelo Frias	Ernst-Rüdiger Olderog	Geoffrey Watson
Christoph Herrmann	Bruno Oliveira	Paolo Zuliani

Best Paper Award

Modelling Nondeterminism
Clare E. Martin, Sharon A. Curtis, and Ingrid Rewitzky

Table of Contents

Extended Static Checking for Java

Greg Nelson

Imaging Systems Department
HP Labs
Mail Stop 1203
1501 Page Mill Road
Palo Alto, CA 94304, USA
gnelson@hp.com

Abstract. The talk provides an overview and demonstration of an Extended Static Checker for the Java programming language, a program checker that finds errors statically but has a much more accurate semantic model than existing static checkers like type checkers and data flow analysers. For example, ESC/Java uses an automatic theorem-prover and reasons about the semantics of assignments and tests in the same way that a program verifier does. But the checker is fully automatic, and feels to the programmer more like a type checker than like a program verifier. A more detailed account of ESC/Java is contained in a recent PLDI paper [1]. The checker described in the talk and in the PLDI paper is a research prototype on which work ceased several years ago, but Joe Kiniry and David Cok have recently produced a more up-to-date checker, ESC/Java 2 [2].

References

1. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *Proc. PLDI'02.* ACM. Berlin, Germany, 2002.
2. David Cok and Joe Kiniry. ESC/Java 2 project page. `http://www.cs.kun.nl/sos/research/escjava/main.html`.

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 $\{0,\dots,n-1\}$. 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

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

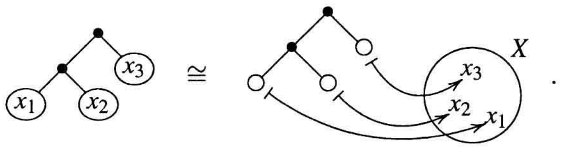
One can use the following coend formula to calculate the action of a left Kan extension when $\mathscr{C} = \mathbf{Sets}$ and $\mathscr{A}$ is small

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

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 $\{0,\dots,n-1\} \to X$ which labels each position within the list with an element from X. Thus we may write

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

We may think of the set $\{0,\dots,n-1\}$ 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 $(S \rhd P)$ 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 $\{0, \dots, n-1\}$.*

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 $(S \rhd P)$ be a container. Its semantics, or extension, is the functor $T_{S\rhd P}$:* **Sets** $\longrightarrow$ **Sets** *defined by*

$$T_{S\rhd P}(X) = \textstyle\sum s:S.\ (P(s) \to X)$$

An element of $T_{S\rhd P}(X)$ is thus a pair (s, f) where $s \in S$ is a shape and $f : P(s) \to X$ is a labelling of the positions over s with elements from X. Note that $T_{S\rhd P}$ 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

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

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 **Sets**^{n} $\longrightarrow$ **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 $(S \rhd P)$

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 $T_{S \triangleright P}(X)$. □

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

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

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 output a position in the input and then look up the data using g.