

Lecture Notes in Computer Science

1608

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

S. Doaitse Swierstra Pedro R. Henriques
José N. Oliveira (Eds.)

Advanced Functional Programming

Third International School, AFP'98
Braga, Portugal, September 12-19, 1998
Revised Lectures

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

S. Doaitse Swierstra
Utrecht University, Department of Computer Science
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: doaitse@cs.uu.nl

Pedro R. Henriques
José N. Oliveira
University of Minho, Department of Informatics
Campus de Gualtar, 4709 Braga Codex, Portugal
E-mail: {prh,jno}@di.uminho.pt

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Advanced functional programming : third international school ;
revised lectures / AFP'98, Braga, Portugal, September 12 - 19, 1998.
S. Doaitse Swierstra ... (ed.). - Berlin ; Heidelberg ; New York ;
Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo
: Springer, 1999
(Lecture notes in computer science ; Vol. 1608)
ISBN 3-540-66241-3

CR Subject Classification (1998): D.1.1, D.3.2, D.2.2, D.2.10

ISSN 0302-9743

ISBN 3-540-66241-3 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1999
Printed in Germany

Typesetting: Camera-ready by author
SPIN: 10704973 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper

Preface

In this volume you will find the lecture notes corresponding to the presentations given at the 3rd summer school on Advanced Functional Programming, held in Braga, Portugal from September 12–19, 1998.

This school was preceded by earlier ones in Båstad (1995, Sweden, LNCS 925) and Olympia, WA (1996, USA, LNCS 1129). The goal of this series of schools is to bring recent developments in the area of functional programming to a large group of students. The notes are published in order to enable individuals, small study groups, and lecturers to become acquainted with recent work in the fast developing area of functional programming.

What made this school particularly interesting was the fact that all lectures introduced useful software, that was used by the students in the classes to get hands-on experience with the subjects taught. We urge readers of this volume to download the latest version of this software from the Internet and try to do the exercises from the text themselves; the proof of the program is in the typing.

The first lecture, on *Sorting Morphisms*, serves as a gentle introduction to the things to come. If you have always been afraid of the word “morphism”, and you have been wondering what catamorphisms, anamorphisms, hylomorphisms, and paramorphisms were about, this is the paper to read first; you will discover that they are merely names for recursion patterns that occur over and over again when writing functional programs. The algorithms in the paper are all about sorting, and since you are likely to know those algorithms by heart already, seeing them structured and analyzed in a novel way should serve as a motivation to read on to the second lecture.

The second lecture, on *Generic Programming*, is almost a book in a book. The notes can be seen as the culminating point of the STOP-project, sponsored by the Dutch government at the end of the 80’s and the beginning of the 90’s. Its overall goal was the development of a calculational way of deriving programs. The project has provided deeper insight into real functional programming and into the theory behind many things commonly written by functional programmers. One of the main achievements of the project has been to make people aware of the fact that many algorithms can be described in a data-independent way. The PolyP system introduced in these notes is one of the translations to the Haskell-world of this theoretical underpinning.

The third lecture, on *Generic Program Transformation*, can also be seen as an application of the theory introduced in lecture two. Many efficiency-improving program transformations can be performed in a mechanical way, and these would not have been possible without insight into the correctness of such transformations gained in the lecture on Generic Programming.

The fourth lecture, on *Designing and Implementing Combinator Languages*, introduces an easy to write formalism for writing down the catamorphisms introduced in earlier chapters. It is shown how quite complicated catamorphisms, that at first sight seem rather forbidding by making extensive use of higher-order do-

mains, can actually be developed in a step-wise fashion, using an attribute grammar view; it is furthermore shown how to relate this way of programming with concepts from the object-oriented world thus making clear what the strengths and weaknesses of each world are.

The fifth lecture, titled *Using MetaML: A Staged Programming Language*, introduces the concept of partial evaluation. It serves as another instance of the quest for “the most generic of writing programs at the lowest cost”. The staging techniques show how costs that were introduced by adding extra levels of abstraction, may be moved from run-time to compile-time.

It has been common knowledge to users of modern functional languages that the type system can be a great help in shortening programs and reducing errors. In the extreme one might see a type as a predicate capturing the properties of any expression with that type. In the sixth lecture on *Cayenne – Spice up your Programming with Dependent Types* it is shown in what direction functional languages are most likely to develop, and what may be expected of the new type systems to be introduced.

The last lecture, titled *Haskell as an Automation Controller*, shows that writing functional programs does not have to imply that one is bound to remain isolated from the rest of the world. Being able to communicate with software written by others in a uniform way, is probably one of the most interesting new developments in current computer science. It appears that the concept of a monad together with the Haskell typing rules, is quite adequate to describe the interface between Haskell programs and the outer world.

Finally we want to thank everyone who contributed to this school and made it such a successful event: sponsors, local system managers, local organizers, students, and last but not least the lecturers. We are convinced that everyone present at the school enjoyed this event as much as we did, and we all hope that you will feel some of the spirit of this event when studying these lecture notes.

March 1999

Doaitse Swierstra
Pedro Henriques
José Oliveira

Sponsorship

The school has received generous sponsorship from:

FCT - Fundação para a Ciência e Tecnologia, Ministério da Ciência e Tecnologia

Adega Cooperativa de Ponte de Lima

Agência Abreu

CGD - Caixa Geral de Depósitos

CIUM - Centro de Informática da Universidade do Minho

DI - Departamento de Informática da Universidade do Minho

GEPL - Grupo de Especificação e Processamento de Linguagens

LESI - Direcção de Curso de Engenharia de Sistemas e Informática

Enabler

Lactolima

Laticínios das Marinhas, Lda

Novabase Porto - Sistemas de Informação SA

Primavera Software

Projecto Camila - Grupo de Métodos Formais

Sidereus - Sistemas de Informação e Consultoria Informática Lda

SIBS - Sociedade Interbancária de Serviços

Vieira de Castro

Local Committee:

José Almeida, Minho

Luís Barbosa, Minho

José Barros, Minho

M. João Frade, Minho

Pedro Henriques, Minho

F. Mário Martins, Minho

F. Luis Neves, Minho

Carla Oliveira, Minho

Jorge Pinto, Lix

Jorge Rocha, Minho

Cesar Rodrigues, Minho

João Saraiva, Minho

M. João Varanda, Minho

Table of Contents

Sorting Morphisms	1
<i>Lex Augusteijn</i>	
1 Introduction	1
2 Morphisms on Lists	2
2.1 The List Catamorphism	2
2.2 The List Anamorphism	4
2.3 The List Hylomorphism	5
2.4 Insertion Sort	6
2.5 Selection Sorts	7
3 Leaf Trees	9
3.1 The Leaf-Tree Catamorphism	9
3.2 The Leaf-Tree Anamorphism	10
3.3 The Leaf-Tree Hylomorphism	11
3.4 Merge Sort	12
4 Binary Trees	13
4.1 The Tree Catamorphism	13
4.2 The Tree Anamorphism	14
4.3 The Tree Hylomorphism	14
4.4 Quicksort	15
4.5 Heap Sort	16
5 Paramorphisms	18
5.1 The List Paramorphism	18
5.2 Insert As Paramorphism	18
5.3 Remove As Paramorphism	19
6 Generalizing Data Structures	20
6.1 Generalizing Quicksort	20
6.2 Generalizing Heap Sort	21
7 Conclusions	23
Generic Programming – An Introduction –	28
<i>Roland Backhouse, Patrik Jansson, Johan Jeuring, Lambert Meertens</i>	
1 Introduction	28
1.1 The Abstraction-Specialisation Cycle	28
1.2 Genericity in Programming Languages	29
1.3 Path Problems	30
1.4 The Plan	33
1.5 Why Generic Programming?	35
2 Algebras, Functors and Datatypes	36
2.1 Algebras and Homomorphisms	36
2.2 Functors	43

2.3	Polynomial Functors	46
2.4	Datatypes Generically	54
2.5	A Simple Polytypic Program	67
3	PolyP	68
3.1	Regular Functors in PolyP	69
3.2	An Example: <code>psum</code>	70
3.3	Basic Polytypic Functions	72
3.4	Type Checking Polytypic Functions	73
3.5	More Examples of Polytypic Functions	75
3.6	PolyLib: A Library of Polytypic Functions	76
4	Generic Unification	83
4.1	Monads and Terms	85
4.2	Generic Unification	89
5	From Functions to Relations	94
5.1	Why Relations?	94
5.2	Parametric Polymorphism	95
5.3	Relators	99
5.4	Occurs-In	101
6	Solutions to Exercises	104
	Generic Program Transformation	116
	<i>Oege de Moor and Ganesh Sittampalam</i>	
1	Introduction	116
2	Abstraction versus Efficiency	117
2.1	Minimum Depth of a Tree	117
2.2	Decorating a Tree	118
2.3	Partitioning a List	119
3	Automating the Transition: Fusion and Higher Order Rewriting	120
4	The MAG System	125
4.1	Getting Acquainted	125
4.2	Accumulation Parameters	128
4.3	Tupling	131
4.4	Carrying On	133
5	Matching Typed λ -Expressions	134
5.1	Types	134
5.2	Expressions	135
5.3	Substitutions	138
5.4	Matching	138
6	Concluding Remarks	140
7	Answers to Exercises	143
	Designing and Implementing Combinator Languages	150
	<i>S. Doaitse Swierstra, Pablo R. Azero Alcocer, João Saraiva</i>	
1	Introduction	150
1.1	Defining Languages	150
1.2	Extending Languages	151

1.3	Embedding Languages	151
1.4	Overview	152
2	Compositional Programs	153
2.1	The Rep_Min Problem	153
2.2	Table_Formatting	159
2.3	Defining Catamorphisms	170
2.4	Discussion	175
3	Attribute Grammars	177
3.1	The Rep_Min Problem	177
3.2	The Table_Formatting Problem	181
3.3	Comparison with Monadic Approach	184
4	Pretty Printing	185
4.1	The General Approach	187
4.2	Improving Filtering	188
4.3	Loss of Sharing in Computations	193
4.4	Discussion	196
5	Strictification	201
5.1	Introduction	201
5.2	Pretty Printing Combinators Strictified	201
6	Conclusions	203
	Using MetaML: A Staged Programming Language	207
	<i>Tim Sheard</i>	
1	Why Staging?	207
2	Relationship to Other Paradigms	210
3	Introducing MetaML	211
3.1	The Bracket Operator: Building Pieces of Code	212
3.2	The Escape Operator: Composing Pieces of Code	213
3.3	The run Operator: Executing User-Constructed Code	214
3.4	The lift Operator: Another Way to Build Code	215
3.5	Lexical Capture of Free Variables: Constant Pieces of Code	216
4	Pattern Matching Against Code	217
5	A Staged Term Rewriting System	218
6	Safe Reductions under Brackets	222
6.1	Safe-Beta	222
6.2	Safe-Eta	222
6.3	Safe-Let-Hoisting	223
7	Non-standard Extensions	223
7.1	Higher Order Type Constructors	223
7.2	Local Polymorphism	224
7.3	Monads	225
7.4	Monads in MetaML	226
7.5	An Example Monad	226
7.6	Safe Monad-Law-Normalization Inside Brackets	227
8	From Interpreters to Compilers Using Staging	228
8.1	The While-Language	228

8.2	The Structure of the Solution	229
8.3	Step 1: Monadic Interpreter	231
8.4	Step 2: Staged Interpreter	233
9	Typing Staged Programs	236
9.1	Type Questions Still to be Addressed	236
10	Conclusion	238
11	Exercises	238
Cayenne — A Language with Dependent Types		240
<i>Lennart Augustsson</i>		
1	Introduction	240
1.1	The Type of <code>printf</code>	241
1.2	The Set “Package”	242
1.3	The Eq Class	243
2	Core Cayenne	246
2.1	Functions	246
2.2	Data Types	247
2.3	Records	248
2.4	The Type of Types	248
3	Full Cayenne	249
3.1	Hidden Arguments	249
3.2	Syntactic Sugar	250
3.3	Modules	251
4	The Cayenne Type System	252
4.1	Translucent Sums	252
4.2	Typing and Evaluation Rules	253
4.3	Type Checking	254
4.4	Undecidability in Practice	257
5	Cayenne as a Proof System	258
6	Implementation	258
6.1	Erasing Types	258
6.2	Keeping Types	260
6.3	The Current Implementation	260
7	Related Work	260
8	Future Work	261
9	Acknowledgments	261
A	The Eq Class	264
B	The Tautology Function	266
Haskell as an Automation Controller		268
<i>Daan Leijen, Erik Meijer, James Hook</i>		
1	Introduction	268
2	Minuscule Introduction to Haskell	269
3	Using COM Components	270
3.1	MS Agents in Haskell	271
3.2	Exercises	273

4	Essential COM	273
4.1	Interface Types	274
4.2	Inheritance	274
4.3	IDL	275
5	Automation	276
5.1	Using Automation	276
5.2	Methods	277
5.3	Properties	277
5.4	HaskellDirect	278
5.5	Exercises	278
6	Advanced Automation	278
6.1	Variants	279
6.2	Optional Arguments	279
7	Advanced Example	280
7.1	Webster	281
7.2	Exercises	282
8	Interacting with other Languages	282
8.1	The Script Server Interfaces	283
8.2	Exporting Values from Haskell	284
8.3	Visual Basic and Haskell	285
8.4	Importing Values into Haskell	286
8.5	Handling Events	286
8.6	Exercises	288
9	Conclusions	288