

Lecture Notes in Computer Science 4719

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

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Tokyo

Roland Backhouse, Jeremy Gibbons, Ralf Hinze and
Johan Jeuring (Eds.)

Datatype-Generic Programming

Spring School
Nottingham, UK, April 24–27, 2006
Revised Lectures

Springer

Preface

A leitmotif in the evolution of programming paradigms has been the level and extent of parametrisation that is facilitated — the so-called *genericity* of the paradigm. The sorts of parameters that can be envisaged in a programming language range from simple values, like integers and floating-point numbers, through structured values, types and classes, to kinds (the type of types and/or classes). *Datatype-generic programming* is about parametrising programs by the structure of the data that they manipulate.

To appreciate the importance of datatype genericity, one need look no further than the internet. The internet is a massive repository of structured data, but the structure is rarely exploited. For example, compression of data can be much more effective if its structure is known, but most compression algorithms regard the input data as simply a string of bits, and take no account of its internal organisation.

Datatype-generic programming is about exploiting the structure of data when it is relevant and ignoring it when it is not. Programming languages most commonly used at the present time do not provide effective mechanisms for documenting and implementing datatype genericity. This volume is a contribution towards improving the state of the art.

The emergence of datatype genericity can be traced back to the late 1980s. A particularly influential contribution was made by the Dutch STOP (Specification and Transformation of Programs) project, led by Lambert Meertens and Doaitse Swierstra. The idea that was “in the air” at the time was the commonality in ways of reasoning about different datatypes. Reynolds’ parametricity theorem, popularised by Wadler [17] as “theorems for free”, and so-called “deforestation” techniques came together in the datatype-generic notions of “catamorphism”, “anamorphism” and “hylomorphism”, and the theorem that every hylomorphism can be expressed as the composition of a catamorphism after an anamorphism. The “theory of lists” [5] became a “theory of F s”, where F is an arbitrary datatype, and the “zip” operation on a *pair* of equal-length *lists* became a generic transformation from an F structure of same-shape G structures to a G structure of same-shape F structures [1, 11].

In response to these largely theoretical results, efforts got underway in the mid-to-late 1990s to properly reflect the developments in programming language design. The extension of functional programming to “polytypic” programming [14, 12] was begun, and, in 1998, the “generic programming” workshop was organised by Roland Backhouse and Tim Sheard at Marstrand in Sweden [4], shortly after a Dagstuhl Seminar on the same topic [13]. The advances that had been made played a prominent part in the Advanced Functional Programming summer school [16, 3, 15, 6], which was held in 1998.

Since the year 2000, the emphasis has shifted yet more towards making datatype-generic programming more practical. Research projects with this goal

have been the Generic Haskell project led by Johan Jeuring at Utrecht University (see, for example, [10, 9]), the DFG-funded Generic Programming project led by Ralf Hinze at the University of Bonn, and the EPSRC-supported Datatype-Generic Programming project at the Universities of Nottingham and Oxford, which sponsored the Spring School reported in this volume. (Note that, although the summer school held in Oxford in August 2002 [2] was entitled “Generic Programming”, the need to distinguish “datatype” generic programming from other notions of “generic” programming had become evident; the paper “Patterns in Datatype-Generic Programming” [8] is the first published occurrence of the term “datatype-generic programming”.)

This volume comprises revisions of the lectures presented at the Spring School on Datatype-Generic Programming held at the University of Nottingham in April 2006. All the lectures have been subjected to thorough internal review by the editors and contributors, supported by independent external reviews.

Gibbons (“Datatype-Generic Programming”) opens the volume with a comprehensive review of different sorts of parametrisation mechanisms in programming languages, including how they are implemented, leading up to the notion of datatype genericity. In common with the majority of the contributors, Gibbons chooses the functional programming language Haskell to make the notions concrete. This is because functional programming languages provide the best test-bed for experimental ideas, free from the administrative noise and clutter inherent in large-scale programming in mainstream languages. In this way, Gibbons relates the so-called “design patterns” introduced by Gamma, Helm, Johnson and Vlissides [7] to datatype-generic programming constructs (the different types of morphism mentioned earlier). The advantage is that the patterns are made concrete, rather than being expressed in prose by example as in [7].

Hinze, Jeuring and Löh (“Comparing Approaches to Generic Programming in Haskell”) compare a variety of ways that datatype-generic programming techniques have been incorporated into functional programming languages, in particular (but not exclusively) Haskell. They base their comparison on a collection of standard examples: encoding and decoding values of a given datatype, comparing values for equality, and mapping a function over, “showing”, and performing incremental updates on the values stored in a datatype. The comparison is based on a number of criteria, including elements like integration into a programming language and tool support.

The goal of Hinze and Löh’s paper (“Generic Programming Now”) is to show how datatype-generic programming can be enabled in present-day Haskell. They identify three key ingredients essential to the task: a *type reflection* mechanism, a *type representation* and a generic *view* on data. Their contribution is to show how these ingredients can be furnished using generalised algebraic datatypes.

The theme of type reflection and type representation is central to Altenkirch, McBride and Morris’s contribution (“Generic Programming with Dependent Types”). Their paper is about defining different universes of types in the *Epigram* system, an experimental programming system based on dependent types. They argue that the level of genericity is dictated by the universe that is cho-

sen. Simpler universes allow greater levels of genericity, whilst more complex universes cause the genericity to be more restricted.

Dependent types, and the Curry-Howard isomorphism between proofs and programs, also play a central role in the Ω mega language introduced by Sheard (“Generic Programming in Ω mega”). Sheard argues for a type system that is more general than Haskell’s, allowing a richer set of programming patterns, whilst still maintaining a sound balance between computations that are performed at run-time and computations performed at compile-time.

Finally, Lämmel and Meijer (“Revealing the X/O Impedance Mismatch”) explore the actual problem of datatype-generic programming in the context of present-day implementations of object-oriented languages and XML data models. The X/O impedance mismatch refers to the incompatibilities between XML and object-oriented models of data. They provide a very comprehensive and up-to-date account of the issues faced by programmers, and how these issues can be resolved.

It remains for us to express our thanks to those who have contributed to the success of the School. First and foremost, we thank Fermín Reig, who was responsible for much of the preparations for the School and its day-to-day organisation. Thanks also to Avril Rathbone and Pablo Nogueira for their organisational support, and to the EPSRC (under grant numbers GR/S27085/01 and GR/D502632/1) and the School of Computer Science and IT of the University of Nottingham for financial support. Finally, we would like to thank the (anonymous) external referees for their efforts towards ensuring the quality of these lecture notes.

Roland Backhouse
 Jeremy Gibbons
 Ralf Hinze
 Johan Jeuring
 June, 2007

References

1. R.C. Backhouse, H. Doornbos, and P. Hoogendijk. A class of commuting relators. Available via World-Wide Web at <http://www.cs.nott.ac.uk/~rcb/MPC/papers>, September 1992.
2. Roland Backhouse and Jeremy Gibbons, editors. *Generic Programming*, volume 2793 of *LNCS Tutorial Series*. Springer, 2003.
3. Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming. An introduction. In Swierstra et al. [16], pages 28–115.
4. Roland Backhouse and Tim Sheard, editors. *Workshop on Generic Programming*, 1998. Informal proceedings available at <http://www.win.tue.nl/cs/wp/papers.html>.
5. R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*. Springer-Verlag, 1987. NATO ASI Series, vol. F36.

6. Oege de Moor and Ganesh Sittampalam. Generic program transformation. In Swierstra et al. [16], pages 116–149.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
8. Jeremy Gibbons. Patterns in datatype-generic programming. In Jörg Striegnitz and Kei Davis, editors, *Multiparadigm Programming*, volume 27. John von Neumann Institute for Computing (NIC), 2003. First International Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL).
9. Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. In Backhouse and Gibbons [2], pages 57–96.
10. Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Backhouse and Gibbons [2], pages 1–56.
11. Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In Eugenio Moggi and Giuseppe Rosolini, editors, *Category Theory and Computer Science, 7th International Conference*, volume 1290 of *LNCS*, pages 242–260. Springer-Verlag, September 1997.
12. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
13. Mehdi Jazayeri, Rüdiger Loos, and David Musser, editors. *Generic Programming: International Seminar, Dagstuhl Castle, Germany, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, Germany, 2000. Follow-up materials at <http://www.cs.rpi.edu/~musser/gp/dagstuhl/>.
14. J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Proceedings of the Second International Summer School on Advanced Functional Programming Techniques*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.
15. Tim Sheard. Using MetaML: a staged programming language. In Swierstra et al. [16], pages 207–239.
16. S. Doaitse Swierstra, Pedro R. Henriques, and José N. Oliveira, editors. *3rd International Summer School on Advanced Functional Programming, Braga, Portugal, 12th-19th September, 1998*, volume 1608 of *LNCS*. Springer, 1999.
17. P. Wadler. Theorems for free! In *4th Symposium on Functional Programming Languages and Computer Architecture, ACM, London*, pages 347–359, September 1989.

Contributors

Thorsten Altenkirch:

School of Computer Science and Information Technology,
University of Nottingham, Nottingham, NG8 1BB, UK
txa@cs.nott.ac.uk,
<http://www.cs.nott.ac.uk/~txa/>.

Roland Backhouse:

School of Computer Science and Information Technology,
University of Nottingham, Nottingham, NG8 1BB, UK
rcb@cs.nott.ac.uk,
<http://www.cs.nott.ac.uk/~rcb/>.

Jeremy Gibbons:

Computing Laboratory, University of Oxford, Oxford, OX1 3QD, UK
jeremy.gibbons@comlab.ox.ac.uk.
<http://www.comlab.ox.ac.uk/jeremy.gibbons/>.

Ralf Hinze:

Institut für Informatik III, Universität Bonn, Römerstraße 164,
53117 Bonn, Germany.
ralf@informatik.uni-bonn.de,
<http://www.informatik.uni-bonn.de/~ralf/>.

Johan Jeuring:

Institute of Information and Computing Sciences, Utrecht University,
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands,
and
Open University, Heerlen, The Netherlands.
johanj@cs.uu.nl
<http://www.cs.uu.nl/~johanj/>

Ralf Lämmel:

Data Programmability Team, Microsoft Corporation, Redmond WA USA
ralf.lammel@microsoft.com,
<http://homepages.cwi.nl/~ralf/>.

Andres Löh:

Institut für Informatik III, Universität Bonn, Römerstraße 164,
53117 Bonn, Germany.
loeh@informatik.uni-bonn.de,
<http://www.informatik.uni-bonn.de/~loeh/>.

Conor McBride:

School of Computer Science and Information Technology,
University of Nottingham, Nottingham, NG8 1BB, UK
ctm@cs.nott.ac.uk,

X

<http://www.cs.nott.ac.uk/~ctm/>.

Erik Meijer:

SQL Server, Microsoft Corporation, Redmond WA, USA
emeijer@microsoft.com,
<http://research.microsoft.com/~emeijer/>.

Peter Morris:

School of Computer Science and Information Technology,
University of Nottingham, Nottingham, NG8 1BB, UK
pwm@cs.nott.ac.uk,
<http://www.cs.nott.ac.uk/~pwm/>.

Tim Sheard:

Computer Science Department, Maseeh College of Engineering and
Computer Science, Portland State University, Portland OR, USA
sheard@cs.pdx.edu
<http://web.cecs.pdx.edu/~sheard/>

Table of Contents

Chapter 1. Datatype-Generic Programming	1
<i>Jeremy Gibbons</i>	
1 Introduction	1
2 Generic programming	2
2.1 Genericity by value	3
2.2 Genericity by type	4
2.3 Genericity by function	6
2.4 Genericity by structure	8
2.5 Genericity by property	10
2.6 Genericity by stage	13
2.7 Genericity by shape	14
2.8 Universal vs ad-hoc genericity	17
2.9 Another dimension of classification	18
3 Origami programming	18
3.1 Maps and folds on lists	19
3.2 Unfolds on lists	19
3.3 Origami for binary trees	20
3.4 Hylomorphisms	21
3.5 Short-cut fusion	22
3.6 Datatype genericity	23
3.7 Bifunctors	24
3.8 Datatype-generic recursion patterns	25
4 The Origami patterns	26
4.1 The Origami family of patterns	27
4.2 An application of Origami	32
4.3 Patterns as HODGPs	34
4.4 The example, revisited	37
5 The essence of the Iterator pattern	39
5.1 Functional iteration	40
5.2 Idioms	42
5.3 Idiomatic traversal	46
5.4 Examples of traversal: shape and contents	47
5.5 Collection and dispersal	49
5.6 Backwards traversal	50
5.7 Laws of traverse	50
5.8 Example	54
6 Conclusions	57
7 Appendix: Java programs	65
7.1 Component	65
7.2 Section	65
7.3 Paragraph	66

7.4	Iterator	66
7.5	SectionIterator	66
7.6	ParagraphIterator	67
7.7	Action	67
7.8	SpellCorrector	67
7.9	Visitor	68
7.10	PrintVisitor	68
7.11	Builder	68
7.12	InvalidBuilderId	69
7.13	ComponentBuilder	69
7.14	PrintBuilder	70
7.15	Main	72
Chapter 2. Comparing Approaches to Generic Programming in Haskell . .		75
<i>Ralf Hinze, Johan Jeuring, and Andres Löh</i>		
1	Introduction	75
2	Why generic programming matters	78
2.1	Data types in Haskell	79
2.2	Structure-representation types	81
2.3	Encoding and decoding	82
2.4	Equality	86
2.5	Map	87
2.6	Show	90
2.7	Update salaries	91
3	Criteria for comparison	92
3.1	Structure in programming languages	92
3.2	The Type Completeness Principle	94
3.3	Well-typed expressions do not go wrong	96
3.4	Information in types	96
3.5	Integration with the underlying programming language	97
3.6	Tools	98
3.7	Other criteria	99
4	Comparing approaches to generic programming	99
4.1	Clean	100
4.2	PolyP	103
4.3	Scrap Your Boilerplate	108
4.4	Approaches based on reflection	118
4.5	Lightweight approaches to generic programming	128
5	Conclusions and future work	144
5.1	Suitability for generic programming concepts	144
5.2	Why would I use this approach?	146
5.3	Future work	148

Chapter 3. Generic Programming, Now!	155
<i>Ralf Hinze and Andres Löh</i>	
1 Introduction	155
2 Preliminaries	156
2.1 Values, types and kinds	156
2.2 Generalised algebraic datatypes	158
2.3 Open datatypes and open functions	159
3 A guided tour	161
3.1 Type-indexed functions	161
3.2 Introducing new datatypes	166
3.3 Generic functions	167
3.4 Dynamic values	170
3.5 Stocktaking	174
4 Type representations	175
4.1 Representation types for types of a fixed kind	175
4.2 Kind-indexed families of representation types	180
4.3 Representations of open type terms	185
5 Views	189
5.1 Spine view	191
5.2 The type-spine view	193
5.3 Lifted spine view	197
5.4 Sum of products	202
5.5 Lifted sums of products	206
6 Related work	207
A Library	209
A.1 Binary trees	209
A.2 Text with indentation	210
A.3 Parsing	211
Chapter 4. Generic Programming with Dependent Types	215
<i>Thorsten Altenkirch, Conor McBride and Peter Morris</i>	
1 Introduction	215
2 Programming with dependent types in Epigram	216
3 The universe of finite types	225
4 Universes for generic programming	228
4.1 Enumerating finite types	230
4.2 Elements of context-free types	233
4.3 Strictly positive types	237
4.4 Generic map	238
4.5 Relating universes	240
4.6 Universes and representation types	240
5 Containers	242
5.1 Unary containers	242
5.2 n -ary containers	246
5.3 Coproducts and products	247
5.4 Structural operations	248

5.5	Inductive types (μ)	250
5.6	Interpreting universes	251
5.7	Small containers	252
6	Derivatives	253
6.1	Derivatives of context-free types	254
6.2	Generic plugging	257
6.3	Derivatives of containers	258
7	Conclusions and further work	260
Chapter 5. Generic Programming in Ω mega		265
<i>Tim Sheard</i>		
1	Introduction	265
1.1	Genericity and the Curry-Howard Isomorphism	265
2	The structure of Ω mega	266
3	A simple example	266
3.1	Overview	268
3.2	Relation to other systems	269
4	Introduction to Ω mega	270
5	Generic programming	277
5.1	Datatype generic programs	277
5.2	The n-sum example	279
5.3	Generic n-way zip	281
5.4	Using Ω mega's hierarchy of levels	286
6	Ω mega's approach to dependent types	288
6.1	Conclusion	289
A	Inductively sequential functions	291
Chapter 6. Revealing the X/O Impedance Mismatch		293
<i>Ralf Lämmel and Erik Meijer</i>		
1	Introduction	293
1.1	What is the X/O impedance mismatch anyway?	293
1.2	An illustrative X-to-O mapping sample	294
1.3	Dimensions of the X/O impedance mismatch	296
1.4	The ambition: survey X/O differences	297
1.5	The setup: map XSD to C#	297
2	Background	298
2.1	Reconciliation of the X/O impedance mismatch	298
2.2	OO programming on XML data with DOM & Co.	300
2.3	OO programming on XML data with 'schema first'	302
2.4	Plain objects vs. XML objects	305
2.5	Object serialization based on 'code first'	306
2.6	Properties of X-to-O mappings	309
3	The X/O data models	313
3.1	Trees vs. graphs	313
3.2	Accessible vs. unavailable parents	315
3.3	Ambiguous vs. unique nominal selectors	316

3.4	Queryable trees vs. dotted objects	317
3.5	Node labels vs. edge labels	319
3.6	Ordered vs. unordered edges	323
3.7	Qualified vs. local selectors	324
3.8	Semi-structured vs. structured content	326
3.9	Tree literals vs. object initialization	328
4	The X/O type systems	331
4.1	Occurrence constraints	332
4.2	Choice types	335
4.3	Nested composites	338
4.4	Local elements	346
4.5	Element templates	347
4.6	Type extension	351
4.7	Element substitution	354
4.8	Type restriction	357
4.9	Simple types	359
5	Concluding remarks	363
A	Extreme mapping options	367
A.1	Type-driven member access	367
A.2	Compile-time validation for construction	369
A.3	Haskell-like maybies	372
A.4	Generics for compositors	375

