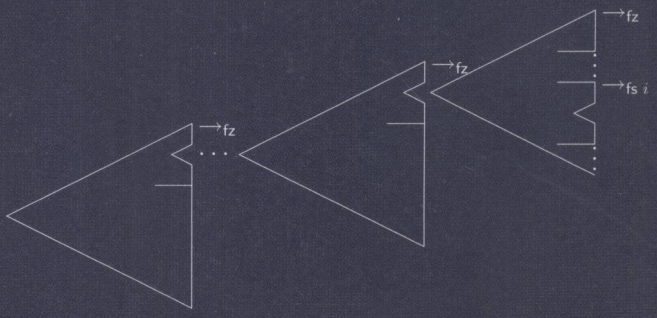
Roland Backhouse
Jeremy Gibbons
Ralf Hinze
Johan Jeuring (Eds.)

Tutorial

LNCS 4719

Datatype-Generic Programming

International Spring School, SSDGP 2006
Nottingham, UK, April 2006
Revised Lectures



Springer

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

Datatype-Generic Programming

International Spring School, SSDGP 2006
Nottingham, UK, April 24-27, 2006
Revised Lectures

Springer

Volume Editors

Roland Backhouse
University of Nottingham
School of Computer Science
Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK
E-mail: rcb@cs.nott.ac.uk

Jeremy Gibbons
Oxford University, Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1, 3QD, UK
E-mail: Jeremy.Gibbons@comlab.ox.ac.uk

Ralf Hinze
Universität Bonn
Institut für Informatik III
Römerstraße 164, 53117 Bonn, Germany
E-mail: ralf@informatik.uni-bonn.de

Johan Jeuring
Utrecht University
Institute of Information and Computing Science
3508 TB Utrecht, The Netherlands
E-mail: johanj@cs.uu.nl

Library of Congress Control Number: Applied for

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

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-540-76785-1 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-76785-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12191963 06/3180 5 4 3 2 1 0

Lecture Notes in Computer Science 4719

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

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 Fs," 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 organized 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 Universtiy 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 programmming 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 a recent Publication [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 chosen. 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 *Ωmega* language introduced by Sheard ("Generic Programming in *Ωmega*"). 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 organization. Thanks also to Avril Rathbone and Pablo Nogueira for their organizational 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.

June 2007

Roland Backhouse
Jeremy Gibbons
Ralf Hinze
Johan Jeuring

References

1. Backhouse, R.C., Doornbos, H., Hoogendijk, P.: A class of commuting relators (September 1992), Available via World-Wide Web at `http://www.cs.nott.ac.uk/~rcb/MPC/papers`
2. Backhouse, R., Gibbons, J. (eds.): Generic Programming. LNCS, vol. 2793. Springer, Heidelberg (2003)
3. Backhouse, R., Jansson, P., Jeuring, J., Meertens, L.: Generic programming. An introduction. In: Swierstra, et al., pp. 28–115, `Braga98`
4. Backhouse, R., Sheard, T. (eds.): Workshop on Generic Programming, Informal proceedings (1998), available at `http://www.win.tue.nl/cs/wp/papers.html`
5. Bird, R.S.: An introduction to the theory of lists. In: Broy, M. (ed.) Logic of Programming and Calculi of Discrete Design. NATO ASI Series, vol. F36, Springer, Heidelberg (1987)
6. de Moor, O., Sittampalam, G.: Generic program transformation. In: Swierstra, et al., pp. 116–149, `Braga98`
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
8. Gibbons, J.: Patterns in datatype-generic programming. In: Striegnitz, J., Davis, K. (eds.) Multiparadigm Programming, John von Neumann Institute for Computing (NIC), First International Workshop on Declarative Programming in the Context of Object-Oriented Languages (DPCOOL), vol. 27 (2003)
9. Hinze, R., Jeuring, J.: Generic Haskell: Applications. In: Backhouse and Gibbons, pp. 57–96, `gp03`
10. Hinze, R., Jeuring, J.: Generic Haskell: Practice and theory. In: Backhouse and Gibbons, pp. 1–56, `gp03`
11. Hoogendijk, P., Backhouse, R.: When do datatypes commute? In: Moggi, E., Rosolini, G. (eds.) CTCS 1997. LNCS, vol. 1290, pp. 242–260. Springer, Heidelberg (1997)
12. Jansson, P., Jeuring, J.: PolyP - a polytypic programming language extension. In: POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 470–482. ACM Press, New York (1997)
13. Jazayeri, M., Musser, D.R., Loos, R.G.K. (eds.): Generic Programming. LNCS, vol. 1766. Springer, Heidelberg (2000), at `http://www.cs.rpi.edu/~musser/gp/dagstuhl/`
14. Jeuring, J., Jansson, P.: Polytypic programming. In: Launchbury, J., Sheard, T., Meijer, E. (eds.) Advanced Functional Programming. LNCS, vol. 1129, pp. 68–114. Springer, Heidelberg (1996)
15. Sheard, T.: Using MetaML: a staged programming language. In: Swierstra, et al., pp. 207–239, `Braga98`
16. Swierstra, S.D., Oliveira, J.N. (eds.): AFP 1998. LNCS, vol. 1608. Springer, Heidelberg (1999)
17. Wadler, P.: Theorems for free. In: 4'th Symposium on Functional Programming Languages and Computer Architecture, pp. 347–359. ACM, London (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
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/

Lecture Notes in Computer Science

Sublibrary 1: Theoretical Computer Science and General Issues

For information about Vols. 1– 4525
please contact your bookseller or Springer

Vol. 4878: E. Tovar, P. Tsigas, H. Fouchal (Eds.), Principles of Distributed Systems. XIII, 457 pages. 2007.

Vol. 4873: S. Aluru, M. Parashar, R. Badrinath, V.K. Prasanna (Eds.), High Performance Computing – HiPC 2007. XXIV, 663 pages. 2007.

Vol. 4863: A. Bonato, F.R.K. Chung (Eds.), Algorithms and Models for the Web-Graph. X, 217 pages. 2007.

Vol. 4855: V. Arvind, S. Prasad (Eds.), FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science. XIV, 558 pages. 2007.

Vol. 4851: S. Boztaş, H.-F.(F.) Lu (Eds.), Applied Algebra, Algebraic Algorithms and Error-Correcting Codes. XII, 368 pages. 2007.

Vol. 4847: M. Xu, Y. Zhan, J. Cao, Y. Liu (Eds.), Advanced Parallel Processing Technologies. XIX, 767 pages. 2007.

Vol. 4846: I. Cervesato (Ed.), Advances in Computer Science – ASIAN 2007. XI, 313 pages. 2007.

Vol. 4838: T. Masuzawa, S. Tixeuil (Eds.), Stabilization, Safety, and Security of Distributed Systems. XIII, 409 pages. 2007.

Vol. 4835: T. Tokuyama (Ed.), Algorithms and Computation. XVII, 929 pages. 2007.

Vol. 4783: J. Holub, J. Žďárek (Eds.), Implementation and Application of Automata. XIII, 324 pages. 2007.

Vol. 4782: R. Perrott, B.M. Chapman, J. Subhlok, R.F. de Mello, L.T. Yang (Eds.), High Performance Computing and Communications. XIX, 823 pages. 2007.

Vol. 4771: T. Bartz-Beielstein, M.J. Blesa Aguilera, C. Blum, B. Naujoks, A. Roli, G. Rudolph, M. Sampels (Eds.), Hybrid Metaheuristics. X, 202 pages. 2007.

Vol. 4770: V.G. Ganzha, E.W. Mayr, E.V. Vorozhtsov (Eds.), Computer Algebra in Scientific Computing. XIII, 460 pages. 2007.

Vol. 4763: J.-F. Raskin, P.S. Thiagarajan (Eds.), Formal Modeling and Analysis of Timed Systems. X, 369 pages. 2007.

Vol. 4746: A. Bondavalli, F. Brasileiro, S. Rajsbaum (Eds.), Dependable Computing. XV, 239 pages. 2007.

Vol. 4743: P. Thulasiraman, X. He, T.L. Xu, M.K. Denko, R.K. Thulasiram, L.T. Yang (Eds.), Frontiers of High Performance Computing and Networking ISPA 2007 Workshops. XXIX, 536 pages. 2007.

Vol. 4742: I. Stojmenovic, R.K. Thulasiram, L.T. Yang, W. Jia, M. Guo, R.F. de Mello (Eds.), Parallel and Distributed Processing and Applications. XX, 995 pages. 2007.

Vol. 4739: R. Moreno Díaz, F. Pichler, A. Quesada Arencibia (Eds.), Computer Aided Systems Theory – EUROCAST 2007. XIX, 1233 pages. 2007.

Vol. 4736: S. Winter, M. Duckham, L. Kulik, B. Kuipers (Eds.), Spatial Information Theory. XV, 455 pages. 2007.

Vol. 4732: K. Schneider, J. Brandt (Eds.), Theorem Proving in Higher Order Logics. IX, 401 pages. 2007.

Vol. 4731: A. Pelc (Ed.), Distributed Computing. XVI, 510 pages. 2007.

Vol. 4726: N. Ziviani, R. Baeza-Yates (Eds.), String Processing and Information Retrieval. XII, 311 pages. 2007.

Vol. 4719: R. Backhouse, J. Gibbons, R. Hinze, J. Jeuring (Eds.), Datatype-Generic Programming. XI, 369 pages. 2007.

Vol. 4711: C.B. Jones, Z. Liu, J. Woodcock (Eds.), Theoretical Aspects of Computing – ICTAC 2007. XI, 483 pages. 2007.

Vol. 4710: C.W. George, Z. Liu, J. Woodcock (Eds.), Domain Modeling and the Duration Calculus. XI, 237 pages. 2007.

Vol. 4708: L. Kučera, A. Kučera (Eds.), Mathematical Foundations of Computer Science 2007. XVIII, 764 pages. 2007.

Vol. 4707: O. Gervasi, M.L. Gavrilova (Eds.), Computational Science and Its Applications – ICCSA 2007, Part III. XXIV, 1205 pages. 2007.

Vol. 4706: O. Gervasi, M.L. Gavrilova (Eds.), Computational Science and Its Applications – ICCSA 2007, Part II. XXIII, 1129 pages. 2007.

Vol. 4705: O. Gervasi, M.L. Gavrilova (Eds.), Computational Science and Its Applications – ICCSA 2007, Part I. XLIV, 1169 pages. 2007.

Vol. 4703: L. Caires, V.T. Vasconcelos (Eds.), CONCUR 2007 – Concurrency Theory. XIII, 507 pages. 2007.

Vol. 4700: C.B. Jones, Z. Liu, J. Woodcock (Eds.), Formal Methods and Hybrid Real-Time Systems. XVI, 539 pages. 2007.

Vol. 4699: B. Kågström, E. Elmroth, J. Dongarra, J. Waśniewski (Eds.), Applied Parallel Computing. XXIX, 1192 pages. 2007.

Vol. 4698: L. Arge, M. Hoffmann, E. Welzl (Eds.), Algorithms – ESA 2007. XV, 769 pages. 2007.

Vol. 4697: L. Choi, Y. Paek, S. Cho (Eds.), Advances in Computer Systems Architecture. XIII, 400 pages. 2007.

Vol. 4688: K. Li, M. Fei, G.W. Irwin, S. Ma (Eds.), Bio-Inspired Computational Intelligence and Applications. XIX, 805 pages. 2007.

Vol. 4684: L. Kang, Y. Liu, S. Zeng (Eds.), Evolvable Systems: From Biology to Hardware. XIV, 446 pages. 2007.

Vol. 4683: L. Kang, Y. Liu, S. Zeng (Eds.), Advances in Computation and Intelligence. XVII, 663 pages. 2007.

Vol. 4681: D.-S. Huang, L. Heutte, M. Loog (Eds.), Advanced Intelligent Computing Theories and Applications. XXVI, 1379 pages. 2007.

Vol. 4672: K. Li, C. Jesshope, H. Jin, J.-L. Gaudiot (Eds.), Network and Parallel Computing. XVIII, 558 pages. 2007.

Vol. 4671: V.E. Malyshkin (Ed.), Parallel Computing Technologies. XIV, 635 pages. 2007.

Vol. 4669: J.M. de Sá, L.A. Alexandre, W. Duch, D. Mandic (Eds.), Artificial Neural Networks – ICANN 2007, Part II. XXXI, 990 pages. 2007.

Vol. 4668: J.M. de Sá, L.A. Alexandre, W. Duch, D. Mandic (Eds.), Artificial Neural Networks – ICANN 2007, Part I. XXXI, 978 pages. 2007.

Vol. 4666: M.E. Davies, C.J. James, S.A. Abdallah, M.D. Plumbley (Eds.), Independent Component Analysis and Blind Signal Separation. XIX, 847 pages. 2007.

Vol. 4665: J. Hromkovič, R. Královič, M. Nunkesser, P. Widmayer (Eds.), Stochastic Algorithms: Foundations and Applications. X, 167 pages. 2007.

Vol. 4664: J. Durand-Lose, M. Margenstern (Eds.), Machines, Computations, and Universality. X, 325 pages. 2007.

Vol. 4661: U. Montanari, D. Sannella, R. Bruni (Eds.), Trustworthy Global Computing. X, 339 pages. 2007.

Vol. 4649: V. Diekert, M.V. Volkov, A. Voronkov (Eds.), Computer Science – Theory and Applications. XIII, 420 pages. 2007.

Vol. 4647: R. Martin, M.A. Sabin, J.R. Winkler (Eds.), Mathematics of Surfaces XII. IX, 509 pages. 2007.

Vol. 4646: J. Duparc, T.A. Henzinger (Eds.), Computer Science Logic. XIV, 600 pages. 2007.

Vol. 4644: N. Azémard, L. Svensson (Eds.), Integrated Circuit and System Design. XIV, 583 pages. 2007.

Vol. 4641: A.-M. Kermarrec, L. Bougé, T. Priol (Eds.), Euro-Par 2007 Parallel Processing. XXVII, 974 pages. 2007.

Vol. 4639: E. Csuhaj-Varjú, Z. Ésik (Eds.), Fundamentals of Computation Theory. XIV, 508 pages. 2007.

Vol. 4638: T. Stützle, M. Birattari, H. H. Hoos (Eds.), Engineering Stochastic Local Search Algorithms. X, 223 pages. 2007.

Vol. 4630: H.J. van den Herik, P. Ciancarini, H.H.L.M.(J.) Donkers (Eds.), Computers and Games. XII, 283 pages. 2007.

Vol. 4628: L.N. de Castro, F.J. Von Zuben, H. Knidel (Eds.), Artificial Immune Systems. XII, 438 pages. 2007.

Vol. 4627: M. Charikar, K. Jansen, O. Reingold, J.D.P. Rolim (Eds.), Approximation, Randomization, and Combinatorial Optimization. XII, 626 pages. 2007.

Vol. 4624: T. Mossakowski, U. Montanari, M. Haveraaen (Eds.), Algebra and Coalgebra in Computer Science. XI, 463 pages. 2007.

Vol. 4623: M. Collard (Ed.), Ontologies-Based Databases and Information Systems. X, 153 pages. 2007.

Vol. 4621: D. Wagner, R. Wattenhofer (Eds.), Algorithms for Sensor and Ad Hoc Networks. XIII, 415 pages. 2007.

Vol. 4619: F. Dehne, J.-R. Sack, N. Zeh (Eds.), Algorithms and Data Structures. XVI, 662 pages. 2007.

Vol. 4618: S.G. Akl, C.S. Calude, M.J. Dinneen, G. Rozenberg, H.T. Wareham (Eds.), Unconventional Computation. X, 243 pages. 2007.

Vol. 4616: A.W.M. Dress, Y. Xu, B. Zhu (Eds.), Combinatorial Optimization and Applications. XI, 390 pages. 2007.

Vol. 4614: B. Chen, M. Paterson, G. Zhang (Eds.), Combinatorics, Algorithms, Probabilistic and Experimental Methodologies. XII, 530 pages. 2007.

Vol. 4613: F.P. Preparata, Q. Fang (Eds.), Frontiers in Algorithmics. XI, 348 pages. 2007.

Vol. 4600: H. Comon-Lundh, C. Kirchner, H. Kirchner (Eds.), Rewriting, Computation and Proof. XVI, 273 pages. 2007.

Vol. 4599: S. Vassiliadis, M. Bereković, T.D. Hämäläinen (Eds.), Embedded Computer Systems: Architectures, Modeling, and Simulation. XVIII, 466 pages. 2007.

Vol. 4598: G. Lin (Ed.), Computing and Combinatorics. XII, 570 pages. 2007.

Vol. 4596: L. Arge, C. Cachin, T. Jurdziński, A. Tarlecki (Eds.), Automata, Languages and Programming. XVII, 953 pages. 2007.

Vol. 4595: D. Bošnački, S. Edelkamp (Eds.), Model Checking Software. X, 285 pages. 2007.

Vol. 4590: W. Damm, H. Hermanns (Eds.), Computer Aided Verification. XV, 562 pages. 2007.

Vol. 4588: T. Harju, J. Karhumäki, A. Lepistö (Eds.), Developments in Language Theory. XI, 423 pages. 2007.

Vol. 4583: S.R. Della Rocca (Ed.), Typed Lambda Calculi and Applications. X, 397 pages. 2007.

Vol. 4580: B. Ma, K. Zhang (Eds.), Combinatorial Pattern Matching. XII, 366 pages. 2007.

Vol. 4576: D. Leivant, R. de Queiroz (Eds.), Logic, Language, Information and Computation. X, 363 pages. 2007.

Vol. 4547: C. Carlet, B. Sunar (Eds.), Arithmetic of Finite Fields. XI, 355 pages. 2007.

Vol. 4546: J. Kleijn, A. Yakovlev (Eds.), Petri Nets and Other Models of Concurrency – ICATPN 2007. XI, 515 pages. 2007.

Vol. 4545: H. Anai, K. Horimoto, T. Kutsia (Eds.), Algebraic Biology. XIII, 379 pages. 2007.

Vol. 4533: F. Baader (Ed.), Term Rewriting and Applications. XII, 419 pages. 2007.

Vol. 4528: J. Mira, J.R. Álvarez (Eds.), Nature Inspired Problem-Solving Methods in Knowledge Engineering, Part II. XXII, 650 pages. 2007.

Vol. 4527: J. Mira, J.R. Álvarez (Eds.), Bio-inspired Modeling of Cognitive Tasks, Part I. XXII, 630 pages. 2007.

¥406.00元

Table of Contents

Datatype-Generic Programming 1
Jeremy Gibbons

Comparing Approaches to Generic Programming in Haskell 72
Ralf Hinze, Johan Jeuring, and Andres Löh

Generic Programming, Now! 150
Ralf Hinze and Andres Löh

Generic Programming with Dependent Types 209
Thorsten Altenkirch, Conor McBride, and Peter Morris

Generic Programming in Ωmega 258
Tim Sheard

Revealing the X/O Impedance Mismatch 285
Ralf Lämmel and Erik Meijer

Author Index 369

Datatype-Generic Programming

Jeremy Gibbons

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, United Kingdom
http://www.comlab.ox.ac.uk/jeremy.gibbons/

Abstract. *Generic programming* aims to increase the flexibility of programming languages, by expanding the possibilities for parametrization — ideally, without also expanding the possibilities for uncaught errors. The term means different things to different people: *parametric polymorphism*, *data abstraction*, *meta-programming*, and so on. We use it to mean polytypism, that is, parametrization by the *shape* of data structures rather than their contents. To avoid confusion with other uses, we have coined the qualified term *datatype-generic programming* for this purpose. In these lecture notes, we expand on the definition of datatype-generic programming, and present some examples of datatype-generic programs. We also explore the connection with *design patterns* in object-oriented programming; in particular, we argue that certain design patterns are just higher-order datatype-generic programs.

1 Introduction

Generic programming is about making programming languages more flexible without compromising safety. Both sides of this equation are important, and becoming more so as we seek to do more and more with computer systems, while becoming ever more dependent on their reliability.

The term 'generic programming' means different things to different people, because they have different ideas about how to achieve the common goal of combining flexibility and safety. To some people, it means *parametric polymorphism*; to others, it means libraries of *algorithms and data structures*; to another group, it means *reflection and meta-programming*; to us, it means *polytypism*, that is, type-safe parametrization by a datatype. Rather than trying to impose our meaning on the other users of the term, or risk confusion by ignoring the other uses, we have chosen to coin the more specific term *datatype-generic programming*. We look in more detail at what we mean by 'datatype-generic programming', and how it relates to what others mean by 'generic programming', in Section 2.

Among the various approaches to datatype-generic programming, one is what we have called elsewhere *origami programming* [38], and what others have variously called *constructive algorithmics* [12,123], *Squiggol* [93], *bananas and lenses* [101], the *Bird-Meertens Formalism* [122,52], and the *algebra of programming* [9], among other names. This is a style of functional (or relational) programming

R. Backhouse et al. (Eds.): Datatype-Generic Programming 2006, LNCS 4719, pp. 1–71, 2007.

based on maps, folds, unfolds and other such higher-order structured recursion operators. Malcolm [92], building on earlier theoretical work by Hagino [56], showed how the existing ad-hoc datatype-specific recursion operators (maps and folds on lists, on binary trees, and so on) could be unified datatype-generically. We explain this school of programming in Section 3.

The origami approach to datatype-generic programming offers a number of benefits, not least of which is the support it provides for reasoning about recursive programs. But one of the reasons for our interest in the approach is that it seems to offer a good way of capturing precisely the essence of a number of the so-called Gang of Four *design patterns*, or reusable abstractions in object-oriented software [35]. This is appealing, because without some kind of datatype-generic constructs, these patterns can only be expressed extra-linguistically, 'as prose, pictures, and prototypes', rather than captured in a library, analysed and reused. We argue this case in Section 4, by presenting higher-order datatype-generic programs capturing ORIGAMI, a small suite of patterns for recursive data structures.

A declarative style of origami programming seems to capture well the computational structure of at least some of these patterns. But because they are usually applied in an imperative setting, they generally involve impure aspects too; a declarative approach does not capture those aspects well. The standard approach the functional programming community now takes to incorporating impure features in a pure setting is by way of *monads* [105,135], which elegantly model all sorts of impure effects such as state, I/O, exceptions and non-determinism. More recently, McBride and Paterson have introduced the notion of *idiom* or *applicative functor* [95], a slight generalization of monads with better compositional properties. One consequence of their definitions is a datatype-generic means of *traversing* collections 'idiomatically', incorporating both pure accumulations and impure effects. In Section 5, we explore the extent to which this construction offers a more faithful higher-order datatype-generic model of the ITERATOR design pattern specifically.

These lecture notes synthesize ideas and results from earlier publications, rather than presenting much that is new. In particular, Section 3 is a summary of two earlier sets of lectures [37,38]; Section 4 recaps the content of a tutorial presented at ECOOP [39] and OOPSLA [40], and subsequently published in a short paper [41]; Section 5 reports on some more recent joint work with Bruno Oliveira [44]. Much of this work took place within the EPSRC-funded *Datatype-Generic Programming* project at Oxford and Nottingham, of which this Spring School marks the final milestone.

2 Generic Programming

Generic programming usually manifests itself as a kind of parametrization. By abstracting from the differences in what would otherwise be separate but similar specific programs, one can make a single unified generic program. Instantiating the parameter in various ways retrieves the various specific programs one started with. Ideally, the abstraction increases expressivity, when some instantiations of

the parameter yield new programs in addition to the original ones; otherwise, all one has gained is some elimination of duplication and a warm fuzzy feeling. The different interpretations of the term 'generic programming' arise from different notions of what constitutes a 'parameter'.

Moreover, a parametrization is usually only called 'generic' programming if it is of a 'non-traditional' kind; by definition, traditional kinds of parametrization give rise only to traditional programming, not generic programming. (This is analogous to the so-called *AI effect*: Rodney Brooks, director of MIT's Artificial Intelligence Laboratory, quoted in [79], observes that 'Every time we figure out a piece of [AI], it stops being magical; we say, "Oh, that's just a computation" '.) Therefore, 'genericity' is in the eye of the beholder, with beholders from different programming traditions having different interpretations of the term. No doubt by-value and by-reference parameter-passing mechanisms for arguments to procedures, as found in Pascal [74], look like 'generic programming' to an assembly-language programmer with no such tools at their disposal.

In this section, we review a number of interpretations of 'genericity' in terms of the kind of parametrization they support. Parametrization by *value* is the kind of parameter-passing mechanism familiar from most programming languages, and while (as argued above) this would not normally be considered 'generic programming', we include it for completeness; parametrization by *type* is what is normally known as polymorphism; parametrization by *function* is sometimes called 'higher-order programming', and is really just parametrization by value where the values are functions; parametrization by *structure* involves passing 'modules' with a varying private implementation of a fixed public signature or interface; parametrization by *property* is a refinement of parametrization by structure, whereby operations of the signature are required to satisfy some laws; parametrization by *stage* allows programs to be partitioned, with meta-programs that generate object programs; and parametrization by *shape* is to parametrization by type as 'by function' is to 'by value'.

2.1 Genericity by Value

One of the first and most fundamental techniques that any programmer learns is how to parametrize computations by values. Those old enough to have been brought up on structured programming are likely to have been given exercises to write programs to draw simple ASCII art: Whatever the scenario, students soon realise the futility of hard-wiring fixed behaviour into programs:

```
procedure Triangle4;
begin
  WriteString ("*"); WriteLn;
  WriteString ("**"); WriteLn;
  WriteString ("***"); WriteLn;
  WriteString ("****"); WriteLn
end;
```

and the benefits of abstracting that behaviour into parameters:

```
procedure Triangle (Side : cardinal);
begin
  var Row, Col : cardinal;
  for Row := 1 to Side do begin
    for Col := 1 to Row do WriteChar ('*');
    WriteLn
  end
end
```

Instead of a parameterless program that always performs the same computation, one ends up with a program with formal parameters, performing different but related computations depending on the actual parameters passed: a *function.*

2.2 Genericity by Type

Suppose that one wants a datatype of lists of integers, and a function to append two such lists. These are written in Haskell [112] as follows:

```
data ListI = NilI | ConsI Integer ListI
appendI :: ListI → ListI → ListI
appendI NilI          ys = ys
appendI (ConsI x xs) ys = ConsI x (appendI xs ys)
```

Suppose in addition that one wanted a datatype and an append function for lists of characters:

```
data ListC = NilC | ConsC Char ListC
appendC :: ListC → ListC → ListC
appendC NilC          ys = ys
appendC (ConsC x xs) ys = ConsC x (appendC xs ys)
```

It is tedious to repeat similar definitions in this way, and it doesn't take much vision to realise that the repetition is unnecessary: the definitions of the datatypes *ListI* and *ListC* are essentially identical, as are the definitions of the functions *appendI* and *appendC*. Apart from the necessity in Haskell to choose distinct names, the only difference in the two datatype definitions is the type of list elements, *Integer* or *Char*. Abstracting from this hard-wired constant leads to a single *polymorphic* datatype parametrized by another type, the type of list elements:

```
data List a = Nil | Cons a (List a)
```

(The term 'parametric datatype' would probably be more precise, but 'polymorphic datatype' is well established.) Unifying the two list datatypes in this way unifies the two programs too, into a single polymorphic program:

```
append :: List a → List a → List a
append Nil          ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```