Michael Hanus (Ed.)

Practical Aspects of Declarative Languages

LNCS 4354

9th International Symposium, PADL 2007
Nice, France, January 2007
Proceedings

Springer

Michael Hanus (Ed.)

Practical Aspects of Declarative Languages

9th International Symposium, PADL 2007
Nice, France, January 14-15, 2007
Proceedings

Springer

Volume Editor

Michael Hanus
Christian-Albrechts-Universität Kiel
Institut für Informatik
24098 Kiel, Germany
E-mail: mh@informatik.uni-kiel.de

Library of Congress Control Number: 2006939136

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-69608-3 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-69608-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11968177 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 4354

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

Editorial Board

Preface

This volume contains the papers presented at the Ninth International Symposium on Practical Aspects of Declarative Languages (PADL 2007) held on January 14–15, 2007 in Nice, France. Information about the conference can be found at `http://www.informatik.uni-kiel.de/~mh/padl07`. Following the tradition of previous events, PADL 2007 was co-located with the 34th Annual Symposium on Principles of Programming Languages (POPL 2007) that was held on January 17–19, 2007.

The PADL conference series is a forum for researchers and practioners to present original work emphasizing novel applications and implementation techniques for all forms of declarative concepts, including functional, logic, constraints, etc. Topics of interest include:

- Innovative applications of declarative languages
- Declarative domain-specific languages and applications
- Practical applications of theoretical results
- New language developments and their impact on applications
- Evaluation of implementation techniques on practical applications
- Novel implementation techniques relevant to applications
- Novel uses of declarative languages in the classroom
- Practical experiences

In response to the call for papers, 65 abstracts were initially received. Finally, 58 full papers were submitted. Each submission was reviewed by at least three Program Committee members. The committee decided to accept 19 papers. In addition, the program also included two invited talks by John Hughes (Chalmers University of Technology) and Pedro Barahona (Universidade Nova de Lisboa).

I would like to thank the Program Committee members who worked hard to produce high-quality reviews for the papers with a tight schedule, as well as all the external reviewers involved in the paper selection. I also would like to thank Gopal Gupta for his expert advice in many aspects of the conference and his publicity efforts. Many thanks also to the organizers of POPL 2007 for hosting PADL 2007 as an affiliated event and to Andrei Voronkov for his continuous help with the EasyChair system that automates many of the tasks involved in chairing a conference. Finally, I thank the University of Kiel, the University of Texas at Dallas, and Compulog Americas for supporting PADL 2007.

October 2006 Michael Hanus

Conference Organization

Program Chair

Michael Hanus
Institut für Informatik
Christian-Albrechts-Universität Kiel
24098 Kiel, Germany
E-mail: mh@informatik.uni-kiel.de

General Chair

Gopal Gupta
Department of Computer Science
University of Texas at Dallas
Dallas, Texas, USA
E-mail: gupta@utdallas.edu

Program Committee

Matthias Blume	Toyota Technological Institute at Chicago, USA
Manuel Chakravarty	University of New South Wales, Australia
Marc Feeley	University of Montreal, Canada
Hai-Feng Guo	University of Nebraska at Omaha, USA
Gopal Gupta	University of Texas at Dallas, USA
Michael Hanus	University of Kiel, Germany (Chair)
Michael Leuschel	University of Düsseldorf, Germany
Simon Peyton Jones	Microsoft Research, Cambridge, UK
Enrico Pontelli	New Mexico State University, USA
Germán Puebla	Technical University of Madrid, Spain
Francesca Rossi	University of Padova, Italy
Michel Rueher	University of Nice, France
Christian Schulte	Royal Institute of Technology, Sweden
Zoltan Somogyi	University of Melbourne, Australia
Peter Stuckey	University of Melbourne, Australia
Doaitse Swierstra	Utrecht University, The Netherlands
Simon Thompson	University of Kent, UK
Pascal Van Hentenryck	Brown University, USA
Germán Vidal	Technical University of Valencia, Spain

External Reviewers

Slim Abdennadher
Alex Aiken
Beatriz Alarcon
Jesus Almendros
Puri Arenas
Ajay Bansal
Jens Bendisposto
Gilles Bernot
Gavin Bierman
Stefano Bistarelli
Mireille Blay-Fornarino
Dan Licata
Suhabe Bugrara
Daniel Cabeza
Manuel Carro
John Clements
Jesus Correas
John Dias
Frank Dignum
Greg Duck
Martin Erwig
Marc Feeley
Amy Felty
Matthew Flatt
Matthew Fluet
Marc Fontaine
Arnaud Gotlieb
Dan Grossman
Raul Gutierrez
David Haguenauer
Kevin Hammond
Stefan Holdermans
Jose Iborra
Johan Jeuring
Andrew Kennedy
Andy King
Karl Klose
Srividya Kona
Marco Kuhlmann
Mikael Z. Lagerkvist
Mario Latendresse
Roman Leshchinksiy
Rainer Leupers
Olivier Lhomme
Sylvain Lippi
Andres Loeh
Michael Maher
Ajay Mallya
Massimo Marchiori
Stefan Monnier
Jose Morales
Claudio Ochoa
Ross Paterson
Inna Pivkina
Bernie Pope
Fred Popowich
Norman Ramsey
Francesca Rossi
Michel Rueher
Claudio Russo
Jean-Charles Régin
Kostis Sagonas
Jaime Sanchez-Hernandez
Dietmar Seipel
Manuel Serrano
Luke Simon
Harald Sondergaard
Don Stewart
Martin Sulzmann
Don Syme
Guido Tack
Peter Thiemann
Son Cao Tran
Alicia Villanueva
Qian Wang
Roland Yap
Damiano Zanardini
Neng-Fa Zhou

Lecture Notes in Computer Science

For information about Vols. 1–4271

please contact your bookseller or Springer

Vol. 4377: M. Abe (Ed.), Topics in Cryptology – CT-RSA 2007. XI, 403 pages. 2006.

Vol. 4369: M. Umeda, A. Wolf, O. Bartenstein, U. Geske, D. Seipel, O. Takata (Eds.), Declarative Programming for Knowledge Management. X, 229 pages. 2006. (Sublibrary LNAI).

Vol. 4367: K. De Bosschere, D. Kaeli, P. Stenström, D. Whalley, T. Ungerer (Eds.), High Performance Embedded Architectures and Compilers. XI, 307 pages. 2006.

Vol. 4361: H.J. Hoogeboom, G. Păun, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing. IX, 555 pages. 2006.

Vol. 4357: L. Buttyán, V. Gligor, D. Westhoff (Eds.), Security and Privacy in Ad-hoc and Sensor Neworks. X, 193 pages. 2006.

Vol. 4355: J. Julliand, O. Kouchnarenko (Eds.), B 2007: Formal Specification and Development in B. XIII, 293 pages. 2006.

Vol. 4354: M. Hanus (Ed.), Practical Aspects of Declarative Languages. X, 335 pages. 2006.

Vol. 4353: T. Schwentick, D. Suciu (Eds.), Database Theory – ICDT 2007. XI, 419 pages. 2006.

Vol. 4352: T.-J. Cham, J. Cai, D. Rajan, T.-S. Chua, L.-T. Chia (Eds.), Advances in Multimedia Modeling, Part II. XVIII, 743 pages. 2007.

Vol. 4351: T.-J. Cham, J. Cai, C. Dorai, D. Rajan, T.-S. Chua, L.-T. Chia (Eds.), Advances in Multimedia Modeling, Part I. XIX, 797 pages. 2007.

Vol. 4348: S.T. Taft, R.A. Duff, R.L. Brukardt, E. Ploedereder, P. Leroy (Eds.), Ada 2005 Reference Manual. XXII, 765 pages. 2006.

Vol. 4347: J. Lopez (Ed.), Critical Information Infrastructures Security. X, 286 pages. 2006.

Vol. 4345: N. Maglaveras, I. Chouvarda, V. Koutkias, R. Brause (Eds.), Biological and Medical Data Analysis. XIII, 496 pages. 2006. (Sublibrary LNBI).

Vol. 4344: V. Gruhn, F. Oquendo (Eds.), Software Architecture. X, 245 pages. 2006.

Vol. 4342: H. de Swart, E. Orłowska, G. Schmidt, M. Roubens (Eds.), Theory and Applications of Relational Structures as Knowledge Instruments II. X, 373 pages. 2006. (Sublibrary LNAI).

Vol. 4341: P.Q. Nguyen (Ed.), Progress in Cryptology - VIETCRYPT 2006. XI, 385 pages. 2006.

Vol. 4340: R. Prodan, T. Fahringer, Grid Computing. XXIII, 317 pages. 2006.

Vol. 4339: E. Ayguadé, G. Baumgartner, J. Ramanujam, P. Sadayappan (Eds.), Languages and Compilers for Parallel Computing. XI, 476 pages. 2006.

Vol. 4338: P. Kalra, S. Peleg (Eds.), Computer Vision, Graphics and Image Processing. XV, 965 pages. 2006.

Vol. 4337: S. Arun-Kumar, N. Garg (Eds.), FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science. XIII, 430 pages. 2006.

Vol. 4334: B. Beckert, R. Hähnle, P.H. Schmitt (Eds.), Verification of Object-Oriented Software. XXIX, 658 pages. 2006. (Sublibrary LNAI).

Vol. 4333: U. Reimer, D. Karagiannis (Eds.), Practical Aspects of Knowledge Management. XII, 338 pages. 2006. (Sublibrary LNAI).

Vol. 4332: A. Bagchi, V. Atluri (Eds.), Information Systems Security. XV, 382 pages. 2006.

Vol. 4331: G. Min, B. Di Martino, L.T. Yang, M. Guo, G. Ruenger (Eds.), Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops. XXXVII, 1141 pages. 2006.

Vol. 4330: M. Guo, L.T. Yang, B. Di Martino, H.P. Zima, J. Dongarra, F. Tang (Eds.), Parallel and Distributed Processing and Applications. XVIII, 953 pages. 2006.

Vol. 4329: R. Barua, T. Lange (Eds.), Progress in Cryptology - INDOCRYPT 2006. X, 454 pages. 2006.

Vol. 4328: D. Penkler, M. Reitenspiess, F. Tam (Eds.), Service Availability. X, 289 pages. 2006.

Vol. 4327: M. Baldoni, U. Endriss (Eds.), Declarative Agent Languages and Technologies IV. VIII, 257 pages. 2006. (Sublibrary LNAI).

Vol. 4326: S. Göbel, R. Malkewitz, I. Iurgel (Eds.), Technologies for Interactive Digital Storytelling and Entertainment. X, 384 pages. 2006.

Vol. 4325: J. Cao, I. Stojmenovic, X. Jia, S.K. Das (Eds.), Mobile Ad-hoc and Sensor Networks. XIX, 887 pages. 2006.

Vol. 4320: R. Gotzhein, R. Reed (Eds.), System Analysis and Modeling: Language Profiles. X, 229 pages. 2006.

Vol. 4319: L.-W. Chang, W.-N. Lie (Eds.), Advances in Image and Video Technology. XXVI, 1347 pages. 2006.

Vol. 4318: H. Lipmaa, M. Yung, D. Lin (Eds.), Information Security and Cryptology. XI, 305 pages. 2006.

Vol. 4317: S.K. Madria, K.T. Claypool, R. Kannan, P. Uppuluri, M.M. Gore (Eds.), Distributed Computing and Internet Technology. XIX, 466 pages. 2006.

Vol. 4316: M.M. Dalkilic, S. Kim, J. Yang (Eds.), Data Mining and Bioinformatics. VIII, 197 pages. 2006. (Sublibrary LNBI).

Vol. 4313: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods. IX, 197 pages. 2006.

Vol. 4312: S. Sugimoto, J. Hunter, A. Rauber, A. Morishima (Eds.), Digital Libraries: Achievements, Challenges and Opportunities. XVIII, 571 pages. 2006.

Vol. 4311: K. Cho, P. Jacquet (Eds.), Technologies for Advanced Heterogeneous Networks II. XI, 253 pages. 2006.

Vol. 4309: P. Inverardi, M. Jazayeri (Eds.), Software Engineering Education in the Modern Age. VIII, 207 pages. 2006.

Vol. 4308: S. Chaudhuri, S.R. Das, H.S. Paul, S. Tirthapura (Eds.), Distributed Computing and Networking. XIX, 608 pages. 2006.

Vol. 4307: P. Ning, S. Qing, N. Li (Eds.), Information and Communications Security. XIV, 558 pages. 2006.

Vol. 4306: Y. Avrithis, Y. Kompatsiaris, S. Staab, N.E. O'Connor (Eds.), Semantic Multimedia. XII, 241 pages. 2006.

Vol. 4305: A.A. Shvartsman (Ed.), Principles of Distributed Systems. XIII, 441 pages. 2006.

Vol. 4304: A. Sattar, B.-h. Kang (Eds.), AI 2006: Advances in Artificial Intelligence. XXVII, 1303 pages. 2006. (Sublibrary LNAI).

Vol. 4303: A. Hoffmann, B.-h. Kang, D. Richards, S. Tsumoto (Eds.), Advances in Knowledge Acquisition and Management. XI, 259 pages. 2006. (Sublibrary LNAI).

Vol. 4302: J. Domingo-Ferrer, L. Franconi (Eds.), Privacy in Statistical Databases. XI, 383 pages. 2006.

Vol. 4301: D. Pointcheval, Y. Mu, K. Chen (Eds.), Cryptology and Network Security. XIII, 381 pages. 2006.

Vol. 4300: Y.Q. Shi (Ed.), Transactions on Data Hiding and Multimedia Security I. IX, 139 pages. 2006.

Vol. 4299: S. Renals, S. Bengio, J.G. Fiscus (Eds.), Machine Learning for Multimodal Interaction. XII, 470 pages. 2006.

Vol. 4297: Y. Robert, M. Parashar, R. Badrinath, V.K. Prasanna (Eds.), High Performance Computing - HiPC 2006. XXIV, 642 pages. 2006.

Vol. 4296: M.S. Rhee, B. Lee (Eds.), Information Security and Cryptology – ICISC 2006. XIII, 358 pages. 2006.

Vol. 4295: J.D. Carswell, T. Tezuka (Eds.), Web and Wireless Geographical Information Systems. XI, 269 pages. 2006.

Vol. 4294: A. Dan, W. Lamersdorf (Eds.), Service-Oriented Computing – ICSOC 2006. XIX, 653 pages. 2006.

Vol. 4293: A. Gelbukh, C.A. Reyes-Garcia (Eds.), MICAI 2006: Advances in Artificial Intelligence. XXVIII, 1232 pages. 2006. (Sublibrary LNAI).

Vol. 4292: G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, A. Nefian, G. Meenakshisundaram, V. Pascucci, J. Zara, J. Molineros, H. Theisel, T. Malzbender (Eds.), Advances in Visual Computing, Part II. XXXII, 906 pages. 2006.

Vol. 4291: G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, A. Nefian, G. Meenakshisundaram, V. Pascucci, J. Zara, J. Molineros, H. Theisel, T. Malzbender (Eds.), Advances in Visual Computing, Part I. XXXI, 916 pages. 2006.

Vol. 4290: M. van Steen, M. Henning (Eds.), Middleware 2006. XIII, 425 pages. 2006.

Vol. 4289: M. Ackermann, B. Berendt, M. Grobelnik, A. Hotho, D. Mladenič, G. Semeraro, M. Spiliopoulou, G. Stumme, V. Svatek, M. van Someren (Eds.), Semantics, Web and Mining. X, 197 pages. 2006. (Sublibrary LNAI).

Vol. 4288: T. Asano (Ed.), Algorithms and Computation. XX, 766 pages. 2006.

Vol. 4287: C. Mao, T. Yokomori (Eds.), DNA Computing. XII, 440 pages. 2006.

Vol. 4286: P. Spirakis, M. Mavronicolas, S. Kontogiannis (Eds.), Internet and Network Economics. XI, 401 pages. 2006.

Vol. 4285: Y. Matsumoto, R. Sproat, K.-F. Wong, M. Zhang (Eds.), Computer Processing of Oriental Languages. XVII, 544 pages. 2006. (Sublibrary LNAI).

Vol. 4284: X. Lai, K. Chen (Eds.), Advances in Cryptology – ASIACRYPT 2006. XIV, 468 pages. 2006.

Vol. 4283: Y.Q. Shi, B. Jeon (Eds.), Digital Watermarking. XII, 474 pages. 2006.

Vol. 4282: Z. Pan, A. Cheok, M. Haller, R.W.H. Lau, H. Saito, R. Liang (Eds.), Advances in Artificial Reality and Tele-Existence. XXIII, 1347 pages. 2006.

Vol. 4281: K. Barkaoui, A. Cavalcanti, A. Cerone (Eds.), Theoretical Aspects of Computing - ICTAC 2006. XV, 371 pages. 2006.

Vol. 4280: A.K. Datta, M. Gradinariu (Eds.), Stabilization, Safety, and Security of Distributed Systems. XVII, 590 pages. 2006.

Vol. 4279: N. Kobayashi (Ed.), Programming Languages and Systems. XI, 423 pages. 2006.

Vol. 4278: R. Meersman, Z. Tari, P. Herrero (Eds.), On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, Part II. XLV, 1004 pages. 2006.

Vol. 4277: R. Meersman, Z. Tari, P. Herrero (Eds.), On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops, Part I. XLV, 1009 pages. 2006.

Vol. 4276: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, Part II. XXXII, 752 pages. 2006.

Vol. 4275: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, Part I. XXXI, 1115 pages. 2006.

Vol. 4274: Q. Huo, B. Ma, E.-S. Chng, H. Li (Eds.), Chinese Spoken Language Processing. XXIV, 805 pages. 2006. (Sublibrary LNAI).

Vol. 4273: I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, L. Aroyo (Eds.), The Semantic Web - ISWC 2006. XXIV, 1001 pages. 2006.

Vol. 4272: P. Havinga, M. Lijding, N. Meratnia, M. Wegdam (Eds.), Smart Sensing and Context. XI, 267 pages. 2006.

Table of Contents

QuickCheck Testing for Fun and Profit

John Hughes

Chalmers University of Technology,
S-41296 Gothenburg,
Sweden

1 Introduction

One of the nice things about purely functional languages is that functions often satisfy simple properties, and enjoy simple algebraic relationships. Indeed, if the functions of an API satisfy elegant laws, that in itself is a sign of a good design—the laws not only indicate conceptual simplicity, but are useful in practice for simplifying programs that use the API, by equational reasoning or otherwise. It is a comfort to us all, for example, to know that in Haskell the following law holds:

```
reverse (xs++ys) == reverse xs++reverse ys
```

where `reverse` is the list reversal function, and `++` is list append.

It is productive to formulate such laws about one's code, but there is always the risk of formulating them incorrectly. A stated law which is untrue is worse than no law at all! Ideally, of course, one should prove them, but at the very least, one should try out the law in a few cases—just to avoid stupid mistakes. We can ease that task a little bit by defining a function to test the law, given values for its free variables:

```
prop_revApp xs ys =
  reverse (xs++ys) == reverse xs++reverse ys
```

Now we can test the law just by applying `prop_revApp` to suitable pairs of lists.

Inventing such pairs of lists, and running the tests, is tedious, however. Wouldn't it be fun to have a tool that would perform that task for us? Then we could simply write laws in our programs and automatically check that they are reasonable hypotheses, at least. In 1999, Koen Claessen and I built just such a tool for Haskell, called "QuickCheck" [4,5,7,6]. Given the definition above, we need only pass `prop_revApp` to `quickCheck` to test the property in 100 random cases:

```
> quickCheck prop_revApp
Falsifiable, after 2 tests:
[1,-1]
[0]
```

Doing so exposes at once that the property is not true! The values printed are a counter-example to the claim, `[1,-1]` being the value of `xs`, and `[0]` the value of `ys`. Indeed, inspecting the property more closely, we see that `xs` and `ys` are the

M. Hanus (Ed.): PADL 2007, LNCS 4354, pp. 1–32, 2007.

wrong way round in the right hand side of the law. After correcting the mistake, quickChecking the property succeeds:

```
> quickCheck prop_revApp
OK, passed 100 tests.
```

While there is no *guarantee* that the property now holds, we can be very much more confident that we did not make a stupid mistake... particularly after running another few thousand tests, which is the work of a few more seconds.

We wrote QuickCheck for fun, but it has turned out to be much more useful and important than we imagined at the time. This paper will describe some of the uses to which it has since been put.

2 A Simple Example: Skew Heaps

To illustrate the use of QuickCheck in program development, we shall implement *skew heaps* (a representation of priority queues), following Chris Okasaki [15]. A heap is a binary tree with labels in the nodes,

```
data Tree a = Null | Fork a (Tree a) (Tree a)
  deriving (Eq, Show)
empty = Null
```

such that the value in each node is less than any value in its subtrees:

```
invariant Null = True
invariant (Fork x l r) = smaller x l && smaller x r
smaller x Null = True
smaller x (Fork y l r) = x <= y && invariant (Fork y l r)
```

Thanks to the invariant, we can extract the minimum element (i.e. the first element in the queue) very cheaply:

```
minElem (Fork x _ _) = x
```

To make other operations on the heap cheap, we aim to keep it roughly balanced—then the cost of traversing a branch will be logarithmic in the number of elements. This is achieved in a skew heap by inserting elements into the two subtrees alternately. No extra information is needed in nodes to keep track of where to insert next: we *always* insert into the left subtree, but swap the subtrees after each insertion—*skewing* the heap—so that the next insertion chooses the other subtree.

```
insert x Null = Fork x Null Null
insert x (Fork y l r) = Fork (min x y) r (insert (max x y) l)
```

We expect that the two subtrees of a node should be "roughly balanced", but what does this mean precisely? A moment's thought suggests that the left and right subtrees should contain precisely the same number of elements after an *odd*

number of insertions, but the right subtree may be one element larger than the left one after an *even* number of insertions. We conjecture that skew heaps are balanced in the following sense:

```
balanced Null = True
balanced (Fork _ l r) = (d==0 || d==1) && balanced l && balanced r
  where d = weight r - weight l

weight Null = 0
weight (Fork _ l r) = 1 + weight l + weight r
```

Now we can use QuickCheck to test our conjecture. To do so we need to generate random skew heaps. Since the only function so far that constructs skew heaps is `insert`, we can construct any reachable skew heap by choosing a random list of elements, and inserting them into the empty heap:

```
make :: [Integer] -> Tree Integer
make ns = foldl (\h n -> insert n h) empty ns
```

We can now formulate the two properties we are interested in as follows:

```
prop_invariant ns = invariant (make ns)
prop_balanced ns = balanced (make ns)
```

We gave `make` a specific type to control the generation of test data: QuickCheck generates property arguments based on the type expected, and constraining the type of `make` is a convenient way to constrain the argument types of both properties at the same time. (If we forget this, then QuickCheck cannot tell what kind of test data to generate, and an "ambiguous overloading" error is reported). Now we can invoke QuickCheck to confirm our conjecture:

```
Skew> quickCheck prop_invariant
OK, passed 100 tests.
Skew> quickCheck prop_balanced
OK, passed 100 tests.
```

We also need an operation to *delete the minimum element* from a heap. Although finding the element is easy (it is always at the root), deleting it is not, because we have to *merge* the two subtrees into one single heap.

```
deleteMin (Fork x l r) = merge l r
```

(In fact, `merge` is usually presented as part of the interface of skew heaps, even if its utility for priority queues is less obvious). If either argument is `Null`, then merge is easy to define, but how should we merge two non-empty heaps? Clearly, the root of the merged heap must contain the lesser of the root elements of `l` and `r`, but that leaves us with *three* heaps to fit into the two subtrees of the new `Fork`—`l`, `r` and `h` below—so two must be merged recursively... *but which two?*

```
merge l Null = l
merge Null r = r
merge l r | minElem l <= minElem r = join l r
          | otherwise              = join r l

join (Fork x l r) h = Fork x ...
```

The trick is to realize that the two subtrees of a node are not created equal: we ensured during insertion that the left subtree is never larger than the right one. So any recursion should be on the *left* subtree, guaranteeing that the size of the recursive argument at least halves at each call, and that the total number of calls is logarithmic in the size of the heaps. Thus we should merge `l` with `h` above, not `r`, and because merging increases the size of the heap, skew the subtrees again, so that the next merge will choose `r` instead.

```
join (Fork x l r) h = Fork x r (merge l h)
```

Is this really right? Let us test our properties again! Of course, now skew heaps can be constructed by a combination of insertions and deletions, so our method of generating random reachable heaps is no longer complete. Now we must generate heaps from a random sequence of insertions *and deletions*:

```
data Op = Insert Integer | DeleteMin
  deriving Show

make ops = foldl op Null ops
  where op h (Insert n)   = insert n h
        op Null DeleteMin = Null
        op h DeleteMin    = deleteMin h
```

One difficulty is that a *random* sequence of insertions and deletions may attempt to delete an element from an empty heap, provoking an error. There are various ways to avoid this: we could arrange not to generate such sequences in the first place, we could generate arbitrary sequences but discard the erroneous ones, or we can simply ignore any deletions that are applied to an empty heap. In the code above we chose the last alternative, because it is the simplest to implement.

Note that `make` now has a different type—it expects a list of `Ops` as its argument—and thus so do our two properties. To test them, QuickCheck needs to be able to generate values of the `Op` type, and to make that possible, we must specify a *generator* for this type.

QuickCheck generators are an abstract data type, with a rich collection of operations for constructing them. Indeed, provision of *first-class generators* is one of the main innovations in QuickCheck. We use the Haskell class system to associate generators with types, by defining instances of

```
class Arbitrary a where
  arbitrary :: Gen a
```

The `Gen` type is also a *monad*, making available the monad operations

```
return :: a -> Gen a
```

to construct a constant generator, and

```
(>>=) :: Gen a -> (a -> Gen b) -> Gen b
```

to sequence two generators—although we usually use the latter via Haskell's syntactic sugar, the do-notation.

So, we specify how `Op` values should be generated as follows:

```
instance Arbitrary Op where
  arbitrary =
    frequency [(2,do n <- arbitrary; return (Insert n)),
               (1,return DeleteMin)]
```

The `frequency` function combines weighted alternatives—here we generate an insertion twice as often as a deletion, since otherwise the resulting heaps would often be very small. In the first alternative, we choose an arbitrary `Integer` and generate an `Insert` containing it; in the second alternative we generate a `DeleteMin` directly.

Now we can check that any sequence of insertions and deletions preserves the heap invariant

```
Skew> quickCheck prop_invariant
OK, passed 100 tests.
```

and that skew heaps remain balanced:

```
Skew> quickCheck prop_balanced
Falsifiable, after 37 tests:
[DeleteMin,Insert (-9),Insert (-18),Insert (-14),Insert 5,
Insert (-13),Insert (-8),Insert 13,DeleteMin,DeleteMin]
```

Oh dear! Clearly, deletion does *not* preserve the balance condition. But maybe the balance condition is too strong? All we really needed above was that the *left subtree is no larger than the right*—so let's call a node "good" if that is the case.

```
good (Fork _ l r) = weight l <= weight r
```

Now, if all the nodes in a heap are good, then `insert` and `merge` will still run in logarithmic time. We can define and test the property that all nodes are good:

```
Skew> quickCheck prop_AllGood
Falsifiable, after 55 tests:
[Insert (-7),DeleteMin,Insert (-16),Insert (-14),DeleteMin,
DeleteMin,DeleteMin,Insert (-21),Insert (-8),Insert 3,
Insert (-1),Insert 1,DeleteMin,DeleteMin,Insert (-12),
Insert 17,Insert 13]
```

Oh dear dear! Evidently, skew heaps contain a mixture of good and bad nodes.

Consulting Okasaki, we find the key insight behind the efficiency of skew heaps: *although bad nodes are more costly to process, they are cheaper to construct!* Whenever we construct a bad node with a large left subtree, then *at the same time* we recurse to create an unusually *small* right subtree—so this recursion is cheaper than expected. What we lose on the swings, we regain on the roundabouts, making for logarithmic *amortized* complexity.

To formalise this argument, Okasaki introduces the notion of "credits"—each bad node carries one credit, which must be supplied when it is created, and can be consumed when it is processed.

```
credits Null = 0
credits h@(Fork _ l r) =
  credits l + credits r + if good h then 0 else 1
```

Since we cannot directly observe the cost of insertion and deletion, we define a function `cost_insert h` that returns the number of recursive calls of `insert` made when inserting into `h`, and `cost_deleteMin h`, which returns the number of calls of `join` made when deleting from `h` (definitions omitted). Now, we claim that *on average* each insertion or deletion in a heap of `n` nodes traverses only `log2 n` nodes, and creates equally many new, possibly bad nodes, so `2*log2 n` credits should suffice for each call. (The first `log2 n` credits pay for the recursion in this call, and the second `log2 n` credits pay for bad nodes in the result).

If we now specify

```
prop_cost_insert n ops =
  cost_insert h <= 2*log2 (weight h) + 1
  where h = make ops
```

then QuickCheck finds a counterexample[1], because this property only holds on average, but when we take credits into account

```
prop_cost_insert n ops =
  cost_insert h + credits (insert n h)
  <=
  2*log2 (weight h) + 1 + credits h
  where h = make ops
```

then the property passes hundreds of thousands of tests. Likewise, the property

```
prop_cost_deleteMin ops =
  h/=Null ==>
    cost_deleteMin h + credits (deleteMin h)
    <=
    2*log2 (weight h) + credits h
 where h = make ops
```

[1] Only one test case in around 3,000 is a counterexample. This is because the method we use to generate heaps produces rather few bad nodes. Counterexamples can be found more quickly by generating heaps directly, rather than via `insert` and `deleteMin`, so that the proportion of bad nodes can be increased.