Phil Trinder
Greg Michaelson
Ricardo Peña (Eds.)

LNCS 3145

Implementation of Functional Languages

15th International Workshop, IFL 2003
Edinburgh, UK, September 2003
Revised Papers

Implementation of Functional Languages

15th International Workshop, IFL 2003
Edinburgh, UK, September 8-11, 2003
Revised Papers

Volume Editors

Phil Trinder
Greg Michaelson
Heriot-Watt University
School of Mathematical and Computer Sciences
Riccarton, EH14 4AS, UK
E-mail: {trinder, greg}@macs.hw.ac.uk

Ricardo Peña
Universidad Complutense de Madrid
Facultad de Informática
Departamento Sistemas Informáticos y Programación
C/ Juan del Rosal, 8, 28040 Madrid, Spain
E-mail: ricardo@sip.ucm.es

Library of Congress Control Number: 2004114139

CR Subject Classification (1998): D.3, D.1.1, F.3

ISSN 0302-9743
ISBN 3-540-23727-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11342205 06/3142 5 4 3 2 1 0

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

Editorial Board

Preface

Functional programming has a long history, reaching back through early realisations in languages like LISP to foundational theories of Computing, in particular λ-calculus and recursive function theory. In turn, functional programming has had wide influence in Computing, both through developments within the discipline, such as formal semantics, polymorphic type checking, lazy evaluation and structural proof, and as a practical embodiment of formalised approaches, such as specification, transformation and partial application.

One of the engaging features of functional programming is precisely the crossover between theory and practice. In particular, it is regarded as essential that all aspects of functional programming are appropriately formalised, especially the specification and implementation of functional languages. Thus, specialist functional programming events like the International Workshop on the Implementation of Functional Languages (IFL) attract contributions where strong use is made of syntactic, semantic and meta-mathematical formalisms to motivate, justify and underpin very practical software systems.

IFL grew out of smaller workshops aimed at practitioners wrestling with the nuts and bolts of making concrete implementations of highly abstract languages. Functional programming has always been bedeviled by an unwarranted reputation for slow and inefficient implementations. IFL is one venue where such problems are tackled head on, always using formal techniques to justify practical implementations.

The 15th International Workshop on the Implementation of Functional Languages (IFL'03) was held in Edinburgh, Scotland from the 8th to the 11th of September, 2003. 42 people attended the Workshop, with participants from Australia, Germany, Holland, Hungary, Ireland, Russia, Spain, Sweden and the USA, as well as from the UK.

There were 32 presentations at IFL'03, in streams on Testing, Compilation and Implementation, Applications, Language Constructs and Programming, Types and Program Analysis, Concurrency and Parallelism, and Language Interfacing. 28 papers were submitted for the draft proceedings. After refereeing, 12 papers were selected for publication in these proceedings, an acceptance rate of 42%.

The Programme Committee was pleased to award the 2nd Peter Landin Prize for the best IFL paper to Pedro Vasconcelos, first author of *Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs*[1].

The 16th International Workshop on the Implementation and Application of Functional Languages will be held in Lübeck, Germany in September 2004. For further details, please see: `http://www.isp.uni-luebeck.de/ifl04/index.htm`.

[1] Co-author Kevin Hammond honourably declined to share the prize as he was associated with its establishment.

Acknowledgments

IFL'03 was organised by the Department of Computer Science, School of Mathematical and Computer Sciences, Heriot-Watt University.

We would like to thank June Maxwell and Christine Mackenzie for their most able Workshop administration and financial arrangements. We would also like to thank Andre Rauber Du Bois for wrangling the WWW site, and Abyd Al Zain and Jann Nystrom for Workshop gophering.

We are pleased to acknowledge the sponsorship of the British Computer Society Formal Aspects of Computing Special Interest Group.

May 2004 Phil Trinder, Greg Michaelson and Ricardo Peña
Editors

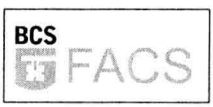
Programme Committee

Thomas Arts	IT-University in Gothenburg, Sweden
Clemens Grelck	University Lübeck, Germany
Stephen Gilmore	University of Edinburgh, UK
Kevin Hammond	University of St Andrews, UK
Frank Huch	Christian-Albrechts-University of Kiel, Germany
Barry Jay	University of Technology Sydney, Australia
Greg Michaelson (Chair)	Heriot-Watt University, UK
Yolanda Ortega Mallen	Universidad Complutense de Madrid, Spain
Ricardo Peña	Universidad Complutense de Madrid, Spain
Simon Peyton Jones	Microsoft Research, UK
Rinus Plasmeijer	University of Nijmegen, The Netherlands
Jocelyn Serot	Blaise Pascal University, France
Phil Trinder (Chair)	Heriot-Watt University, UK
David S. Wise	Indiana University, USA

Referees

Abdallah Al Zain	Ralf Laemmel	Clara Segura
Artem Alimarine	Hans-Wolfgang Loidl	Sjaak Smetsers
Bernd Braßel	Rita Loogen	Jonathan Sobel
Olaf Chitil	Jan Henry Nystrom	Don Syme
Koen Claessen	Enno Ohlebusch	John van Groningen
Walter Dosch	Lars Pareto	Arjen van Weelden
Andre Rauber Du Bois	Robert Pointon	Pedro Vasconcelos
David de Frutos Escrig	Fernando Rubio	
Michael Hanus	Sven-Bodo Scholz	

Sponsors



Lecture Notes in Computer Science

For information about Vols. 1–3198

please contact your bookseller or Springer

Vol. 3305: P.M.A. Sloot, B. Chopard, A.G. Hoekstra (Eds.), Cellular Automata. XV, 883 pages. 2004.

Vol. 3302: W.-N. Chin (Ed.), Programming Languages and Systems. XIII, 453 pages. 2004.

Vol. 3299: F. Wang (Ed.), Automated Technology for Verification and Analysis. XII, 506 pages. 2004.

Vol. 3294: C.N. Dean, R.T. Boute (Eds.), Teaching Formal Methods. X, 249 pages. 2004.

Vol. 3293: C.-H. Chi, M. van Steen, C. Wills (Eds.), Web Content Caching and Distribution. IX, 283 pages. 2004.

Vol. 3292: R. Meersman, Z. Tari, A. Corsaro (Eds.), On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops. XXIII, 885 pages. 2004.

Vol. 3291: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. XXV, 824 pages. 2004.

Vol. 3290: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. XXV, 823 pages. 2004.

Vol. 3289: S. Wang, K. Tanaka, S. Zhou, T.W. Ling, J. Guan, D. Yang, F. Grandi, E. Mangina, I.-Y. Song, H.C. Mayr (Eds.), Conceptual Modeling for Advanced Application Domains. XXII, 692 pages. 2004.

Vol. 3288: P. Atzeni, W. Chu, H. Lu, S. Zhou, T.W. Ling (Eds.), Conceptual Modeling – ER 2004. XXI, 869 pages. 2004.

Vol. 3287: A. Sanfeliu, J.F.M. Trinidad, J.A. Carrasco Ochoa (Eds.), Progress in Pattern Recognition, Image Analysis and Applications. XVII, 703 pages. 2004.

Vol. 3286: G. Karsai, E. Visser (Eds.), Generative Programming and Component Engineering. XIII, 491 pages. 2004.

Vol. 3284: A. Karmouch, L. Korba, E.R.M. Madeira (Eds.), Mobility Aware Technologies and Applications. XII, 382 pages. 2004.

Vol. 3281: T. Dingsøyr (Ed.), Software Process Improvement. X, 207 pages. 2004.

Vol. 3280: C. Aykanat, T. Dayar, İ. Körpeoğlu (Eds.), Computer and Information Sciences - ISCIS 2004. XVIII, 1009 pages. 2004.

Vol. 3278: A. Sahai, F. Wu (Eds.), Utility Computing. XI, 272 pages. 2004.

Vol. 3274: R. Guerraoui (Ed.), Distributed Computing. XIII, 465 pages. 2004.

Vol. 3273: T. Baar, A. Strohmeier, A. Moreira, S.J. Mellor (Eds.), <<UML>> 2004 - The Unified Modelling Language. XIII, 454 pages. 2004.

Vol. 3271: J. Vicente, D. Hutchison (Eds.), Management of Multimedia Networks and Services. XIII, 335 pages. 2004.

Vol. 3270: M. Jeckle, R. Kowalczyk, P. Braun (Eds.), Grid Services Engineering and Management. X, 165 pages. 2004.

Vol. 3269: J. Lopez, S. Qing, E. Okamoto (Eds.), Information and Communications Security. XI, 564 pages. 2004.

Vol. 3266: J. Solé-Pareta, M. Smirnov, P.V. Mieghem, J. Domingo-Pascual, E. Monteiro, P. Reichl, B. Stiller, R.J. Gibbens (Eds.), Quality of Service in the Emerging Networking Panorama. XVI, 390 pages. 2004.

Vol. 3265: R.E. Frederking, K.B. Taylor (Eds.), Machine Translation: From Real Users to Research. XI, 392 pages. 2004. (Subseries LNAI).

Vol. 3264: G. Paliouras, Y. Sakakibara (Eds.), Grammatical Inference: Algorithms and Applications. XI, 291 pages. 2004. (Subseries LNAI).

Vol. 3263: M. Weske, P. Liggesmeyer (Eds.), Object-Oriented and Internet-Based Technologies. XII, 239 pages. 2004.

Vol. 3262: M.M. Freire, P. Chemouil, P. Lorenz, A. Gravey (Eds.), Universal Multiservice Networks. XIII, 556 pages. 2004.

Vol. 3261: T. Yakhno (Ed.), Advances in Information Systems. XIV, 617 pages. 2004.

Vol. 3260: I.G.M.M. Niemegeers, S.H. de Groot (Eds.), Personal Wireless Communications. XIV, 478 pages. 2004.

Vol. 3258: M. Wallace (Ed.), Principles and Practice of Constraint Programming – CP 2004. XVII, 822 pages. 2004.

Vol. 3257: E. Motta, N.R. Shadbolt, A. Stutt, N. Gibbins (Eds.), Engineering Knowledge in the Age of the Semantic Web. XVII, 517 pages. 2004. (Subseries LNAI).

Vol. 3256: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (Eds.), Graph Transformations. XII, 451 pages. 2004.

Vol. 3255: A. Benczúr, J. Demetrovics, G. Gottlob (Eds.), Advances in Databases and Information Systems. XI, 423 pages. 2004.

Vol. 3254: E. Macii, V. Paliouras, O. Koufopavlou (Eds.), Integrated Circuit and System Design. XVI, 910 pages. 2004.

Vol. 3253: Y. Lakhnech, S. Yovine (Eds.), Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems. X, 397 pages. 2004.

Vol. 3252: H. Jin, Y. Pan, N. Xiao, J. Sun (Eds.), Grid and Cooperative Computing - GCC 2004 Workshops. XVIII, 785 pages. 2004.

Vol. 3251: H. Jin, Y. Pan, N. Xiao, J. Sun (Eds.), Grid and Cooperative Computing - GCC 2004. XXII, 1025 pages. 2004.

Vol. 3250: L.-J. (LJ) Zhang, M. Jeckle (Eds.), Web Services. X, 301 pages. 2004.

Vol. 3249: B. Buchberger, J.A. Campbell (Eds.), Artificial Intelligence and Symbolic Computation. X, 285 pages. 2004. (Subseries LNAI).

Vol. 3246: A. Apostolico, M. Melucci (Eds.), String Processing and Information Retrieval. XIV, 332 pages. 2004.

Vol. 3245: E. Suzuki, S. Arikawa (Eds.), Discovery Science. XIV, 430 pages. 2004. (Subseries LNAI).

Vol. 3244: S. Ben-David, J. Case, A. Maruoka (Eds.), Algorithmic Learning Theory. XIV, 505 pages. 2004. (Subseries LNAI).

Vol. 3243: S. Leonardi (Ed.), Algorithms and Models for the Web-Graph. VIII, 189 pages. 2004.

Vol. 3242: X. Yao, E. Burke, J.A. Lozano, J. Smith, J.J. Merelo-Guervós, J.A. Bullinaria, J. Rowe, P. Tiňo, A. Kabán, H.-P. Schwefel (Eds.), Parallel Problem Solving from Nature - PPSN VIII. XX, 1185 pages. 2004.

Vol. 3241: D. Kranzlmüller, P. Kacsuk, J.J. Dongarra (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface. XIII, 452 pages. 2004.

Vol. 3240: I. Jonassen, J. Kim (Eds.), Algorithms in Bioinformatics. IX, 476 pages. 2004. (Subseries LNBI).

Vol. 3239: G. Nicosia, V. Cutello, P.J. Bentley, J. Timmis (Eds.), Artificial Immune Systems. XII, 444 pages. 2004.

Vol. 3238: S. Biundo, T. Frühwirth, G. Palm (Eds.), KI 2004: Advances in Artificial Intelligence. XI, 467 pages. 2004. (Subseries LNAI).

Vol. 3236: M. Núñez, Z. Maamar, F.L. Pelayo, K. Pousttchi, F. Rubio (Eds.), Applying Formal Methods: Testing, Performance, and M/E-Commerce. XI, 381 pages. 2004.

Vol. 3235: D. de Frutos-Escrig, M. Nunez (Eds.), Formal Techniques for Networked and Distributed Systems – FORTE 2004. X, 377 pages. 2004.

Vol. 3234: M.J. Egenhofer, C. Freksa, H.J. Miller (Eds.), Geographic Information Science. VIII, 345 pages. 2004.

Vol. 3232: R. Heery, L. Lyon (Eds.), Research and Advanced Technology for Digital Libraries. XV, 528 pages. 2004.

Vol. 3231: H.-A. Jacobsen (Ed.), Middleware 2004. XV, 514 pages. 2004.

Vol. 3230: J.L. Vicedo, P. Martínez-Barco, R. Muñoz, M. Saiz Noeda (Eds.), Advances in Natural Language Processing. XII, 488 pages. 2004. (Subseries LNAI).

Vol. 3229: J.J. Alferes, J. Leite (Eds.), Logics in Artificial Intelligence. XIV, 744 pages. 2004. (Subseries LNAI).

Vol. 3226: M. Bouzeghoub, C. Goble, V. Kashyap, S. Spaccapietra (Eds.), Semantics of a Networked World. XIII, 326 pages. 2004.

Vol. 3225: K. Zhang, Y. Zheng (Eds.), Information Security. XII, 442 pages. 2004.

Vol. 3224: E. Jonsson, A. Valdes, M. Almgren (Eds.), Recent Advances in Intrusion Detection. XII, 315 pages. 2004.

Vol. 3223: K. Slind, A. Bunker, G. Gopalakrishnan (Eds.), Theorem Proving in Higher Order Logics. VIII, 337 pages. 2004.

Vol. 3222: H. Jin, G.R. Gao, Z. Xu, H. Chen (Eds.), Network and Parallel Computing. XX, 694 pages. 2004.

Vol. 3221: S. Albers, T. Radzik (Eds.), Algorithms – ESA 2004. XVIII, 836 pages. 2004.

Vol. 3220: J.C. Lester, R.M. Vicari, F. Paraguaçu (Eds.), Intelligent Tutoring Systems. XXI, 920 pages. 2004.

Vol. 3219: M. Heisel, P. Liggesmeyer, S. Wittmann (Eds.), Computer Safety, Reliability, and Security. XI, 339 pages. 2004.

Vol. 3217: C. Barillot, D.R. Haynor, P. Hellier (Eds.), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2004. XXXVIII, 1114 pages. 2004.

Vol. 3216: C. Barillot, D.R. Haynor, P. Hellier (Eds.), Medical Image Computing and Computer-Assisted Intervention – MICCAI 2004. XXXVIII, 930 pages. 2004.

Vol. 3215: M.G.. Negoita, R.J. Howlett, L.C. Jain (Eds.), Knowledge-Based Intelligent Information and Engineering Systems. LVII, 906 pages. 2004. (Subseries LNAI).

Vol. 3214: M.G.. Negoita, R.J. Howlett, L.C. Jain (Eds.), Knowledge-Based Intelligent Information and Engineering Systems. LVIII, 1302 pages. 2004. (Subseries LNAI).

Vol. 3213: M.G.. Negoita, R.J. Howlett, L.C. Jain (Eds.), Knowledge-Based Intelligent Information and Engineering Systems. LVIII, 1280 pages. 2004. (Subseries LNAI).

Vol. 3212: A. Campilho, M. Kamel (Eds.), Image Analysis and Recognition. XXIX, 862 pages. 2004.

Vol. 3211: A. Campilho, M. Kamel (Eds.), Image Analysis and Recognition. XXIX, 880 pages. 2004.

Vol. 3210: J. Marcinkowski, A. Tarlecki (Eds.), Computer Science Logic. XI, 520 pages. 2004.

Vol. 3209: B. Berendt, A. Hotho, D. Mladenic, M. van Someren, M. Spiliopoulou, G. Stumme (Eds.), Web Mining: From Web to Semantic Web. IX, 201 pages. 2004. (Subseries LNAI).

Vol. 3208: H.J. Ohlbach, S. Schaffert (Eds.), Principles and Practice of Semantic Web Reasoning. VII, 165 pages. 2004.

Vol. 3207: L.T. Yang, M. Guo, G.R. Gao, N.K. Jha (Eds.), Embedded and Ubiquitous Computing. XX, 1116 pages. 2004.

Vol. 3206: P. Sojka, I. Kopecek, K. Pala (Eds.), Text, Speech and Dialogue. XIII, 667 pages. 2004. (Subseries LNAI).

Vol. 3205: N. Davies, E. Mynatt, I. Siio (Eds.), UbiComp 2004: Ubiquitous Computing. XVI, 452 pages. 2004.

Vol. 3204: C.A. Peña Reyes, Coevolutionary Fuzzy Modeling. XIII, 129 pages. 2004.

Vol. 3203: J. Becker, M. Platzner, S. Vernalde (Eds.), Field Programmable Logic and Application. XXX, 1198 pages. 2004.

Vol. 3202: J.-F. Boulicaut, F. Esposito, F. Giannotti, D. Pedreschi (Eds.), Knowledge Discovery in Databases: PKDD 2004. XIX, 560 pages. 2004. (Subseries LNAI).

Vol. 3201: J.-F. Boulicaut, F. Esposito, F. Giannotti, D. Pedreschi (Eds.), Machine Learning: ECML 2004. XVIII, 580 pages. 2004. (Subseries LNAI).

Vol. 3199: H. Schepers (Ed.), Software and Compilers for Embedded Systems. X, 259 pages. 2004.

Table of Contents

Implementation of Functional Languages

Lazy Assertions

Olaf Chitil, Dan McNeill, and Colin Runciman

Department of Computer Science, The University of York, UK

Abstract. Assertions test expected properties of run-time values without disrupting the normal working of a program. So in a lazy functional language assertions should be lazy – not forcing evaluation, but only examining what is evaluated by other parts of the program. We explore the subtle semantics of lazy assertions and describe sequential and concurrent variants of a method for checking lazy assertions. All variants are implemented in Haskell.

1 Introduction

A programmer writing a section of code often has in mind certain assumptions or intentions about the values involved. Some of these assumptions or intentions are expressed in a way that can be verified by a compiler, for example as part of a type system. Those beyond the expressive power of static types could perhaps be proved separately as theorems, but such a demanding approach is rarely taken. Instead of leaving key properties unexpressed and unchecked, a useful and comparatively simple option is to express them as *assertions* – boolean-valued expressions that the programmer assumes or intends will always be true. Assertions are checked at run-time as they are encountered, and any failures are reported. If no assertion fails, the program runs just as it would normally, apart from the extra time and space needed for checking.

The usefulness of assertions in conventional state-based programming has long been recognised, and many imperative programming systems include some support for them. In these systems, each assertion is attached to a *program point*; whenever control reaches that point the corresponding assertion is immediately evaluated to a boolean result. Important special cases of program points with assertions include points of entry to, or return from, a procedure.

In a functional language, the basic units of programs are expressions rather than commands. The commonest form of expression is a function application. So our first thought might be that an assertion in a functional language can simply be attached to an expression: an assertion about arguments (or 'inputs') alone can be checked before the expression is evaluated and an assertion involving the result (or 'output') can be checked afterwards. But in a lazy language this view is at odds with the need to preserve normal semantics. Arguments may be unevaluated when the expression is entered, and may remain unevaluated or only partially evaluated even after the expression has been reduced to a result. The result itself may only be evaluated to *weak head-normal form*. So neither arguments nor result can safely be the subjects of an arbitrary boolean assertion that could demand their evaluation in full.

P. Trinder, G. Michaelson, and R. Peña (Eds.): IFL 2003, LNCS 3145, pp. 1–19, 2004.

How can assertions be introduced in a lazy functional language? How can we satisfy our eagerness to evaluate assertions, so that failures can be caught as soon as possible, without compromising the lazy evaluation order of the underlying program to which assertions have been added? We aim to support assertions by a small but sufficient library defined in the programming language itself. This approach avoids the need to modify compilers or run-time systems and gives the programmer a straightforward and familiar way of using a new facility. Specifically, we shall be programming in Haskell[3].

The rest of the paper is organised as follows. Section 2 uses two examples to illustrate the problem with eager assertions in a lazy language. Section 3 outlines and illustrates the contrasting nature of lazy assertions. Section 4 first outlines an implementation of lazy assertions that postpones their evaluation until the underlying program is finished; it then goes on to describe alternative implementations in which each assertion is evaluated by a concurrent thread. Section 5 uncovers a residual problem of sequential demand within assertions. Section 6 gives a brief account of our early experience using lazy assertions in application programs. Section 7 discusses related work. Section 8 concludes and suggests future work.

2 Eager Assertions Must Be True

A library provided with the Glasgow Haskell compiler[1] already includes a function `assert :: Bool -> a -> a`. It is so defined that `assert True x = x` but an application of `assert False` causes execution to halt with a suitable error message. An application of `assert` always expresses an *eager* assertion because it is a strict function: evaluation is driven by the need to reduce the boolean argument, and no other computation takes place until the value `True` is obtained.

Example: Sets as Ordered Trees

Consider the following datatype.

```
data Ord a => Set a  = Empty
                     | Union (Set a) a (Set a)
```

Functions defined over sets include `with` and `elem`, where `s 'with' x` represents $s \cup \{x\}$ and `x 'elem' s` represents the membership test $x \in s$.

```
with :: Ord a => Set a -> a -> Set a
Empty           'with' x = Union Empty x Empty
(Union s1 y s2) 'with' x = case compare x y of
                             LT -> Union (s1 'with' x) y s2
                             EQ -> Union s1 y s2
                             GT -> Union s1 y (s2 'with' x)
```

[1] `http://www.haskell.org/ghc`

```
elem :: Ord a => a -> Set a -> Bool
x `elem` Empty            = False
x `elem` (Union s1 y s2) = case compare x y of
                             LT -> x `elem` s1
                             EQ -> True
                             GT -> x `elem` s2
```

The `Ord a` qualification in the definition of `Set` and in the signatures for `with` and `elem` only says that comparison operators are defined for the type `a`. It does *not* guarantee that `Set a` values are strictly ordered trees, which is what the programmer intends. To assert this property, we could define the following predicate.

```
strictlyOrdered :: Ord a => Set a -> Bool
strictlyOrdered = soBetween Nothing Nothing
    where
    soBetween _  _  Empty           = True
    soBetween lo hi (Union s1 x s2) = between lo hi x &&
                                      soBetween lo (Just x) s1 &&
                                      soBetween (Just x) hi s2
    between lo hi x = maybe True (< x) lo && maybe True (> x) hi
```

Something else the programmer intends is a connection between `with` and `elem`. It can be expressed by asserting `x `elem` (s `with` x)`. Combining this property with the ordering assertion we might define:

```
s `checkedWith` x = assert post s'
                    where
                    s'   = assert pre s `with` x
                    pre  = strictlyOrdered s
                    post = strictlyOrdered s' && x `elem` s'
```

Observations. The eager assertions in `checkedWith` may 'run ahead' of evaluation actually required by the underlying program, forcing fuller evaluation of tree structures and elements. The strict-ordering test is a conjunction of two comparisons for *every* internal node of a tree, forcing the entire tree to be evaluated (unless the test fails). Even the check involving `elem` forces the path from the root to `x`.

Does this matter? Surely some extra evaluation is inevitable when non-trivial assertions are introduced? It does matter. If assertion-checking forces evaluation it could degenerate into a pre-emptive, non-terminating and unproductive process. What if, for example, a computation involves the set of all integers, represented as in Figure 1? Functions such as `elem` and `with` still produce useful results. But `checkedWith` eagerly carries the whole computation away on an infinite side-track!

Even where eager assertions terminate they may consume time or space out of proportion with normal computation. Also, assertions are often checked in the

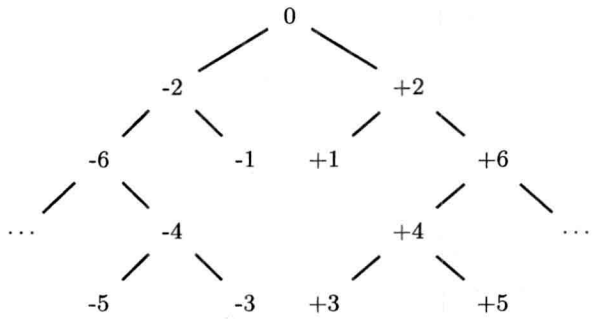


Fig. 1. A tree representation of the infinite set of integers. Each integer i occurs at a depth no greater than $2\log_2(\text{abs}(i) + 1)$. Differences between adjacent elements on leftmost and rightmost paths are successive powers of two.

hope of shedding light on a program failure; it could be distracting to report a failed assertion about values that are irrelevant as they were never needed by the failing program.

3 Lazy Assertions Must Not Be False

So assertions should only examine those parts of their subject data structures that are in any case demanded by the underlying program. Lazy assertions should make a (provisional) assumption of validity about other data not (yet) evaluated. Computation of the underlying program should proceed not only if an assertion reduces to `True`, but also if it cannot (yet) be reduced to a value at all; the only constraint is that an assertion must never reduce to `False`.

If we are to guard data structures that are the subjects of assertions from over-evaluation, we cannot continue to allow arbitrary boolean expressions involving these structures. We need to separate the *predicate* of the assertion from the *subject* to which it is applied. An implementation of assertions should combine the two using only a special evaluation-safe form of application. So the type of `assert` becomes

```
assert :: (a -> Bool) -> a -> a
```

where `assert p` acts as a lazy partial identity.

Example Revisited

If we had an implementation of this lazy `assert`, how would it alter the ordered-tree example? In view of the revised type of `assert`, the definition of `checkedWith` must be altered slightly, making `pre` and `post` predicates rather than booleans.

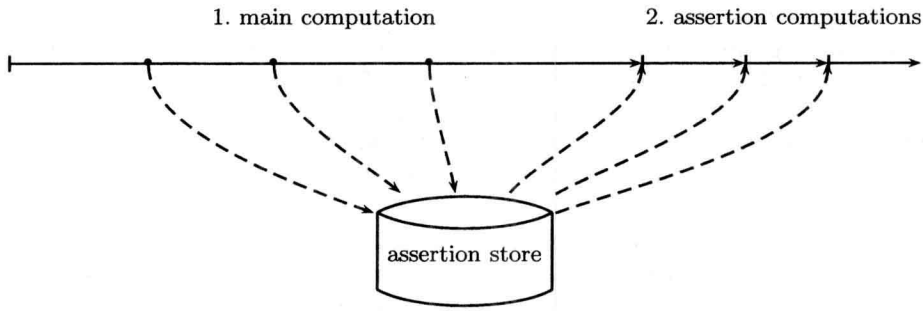


Fig. 2. Delayed Assertions in Time.

```
s `checkedWith` x = assert post (assert pre s `with` x)
                    where
                    pre  = strictlyOrdered
                    post = \s' -> strictlyOrdered s' && x `elem` s'
```

Now the computation of a `checkedWith` application proceeds more like a normal application of `with`. Even if infinite sets are involved, the corresponding assertions are only partially computed, up to the limits imposed by the finite needed parts of these sets.

4 Implementation

Having established the benefits of lazy assertions we now turn to the question of how they can be implemented in Haskell. We develop an assertion library in steps: we start with a simple version, criticise it, and then refine it to the next version.

4.1 Delayed Assertions

We have to ensure that the evaluation of the assertions cannot disturb the evaluation of the underlying program. A very simple idea for achieving this is to evaluate all assertions *after* termination of the main computation.

Figure 2 illustrates the idea. The main computation only evaluates the underlying program and collects all assertions in a global store. After termination of the main computation assertions are taken from the store and evaluated one after the other.

We are certain that lazy assertions cannot be implemented within pure Haskell 98. In particular we need the function `unsafePerformIO :: IO a -> a` to perform actions of the IO monad without giving `assert` a monadic type. We aim to minimise the use of language extensions and restrict ourselves to extensions supported by most Haskell systems. Our implementation is far more concise and potentially portable than any modification of a compiler or run-time system could be.

Which extensions do we need for delayed assertions? Extended exceptions enable a program to catch all erroneous behaviour of a subcomputation. They ensure that all assertions are evaluated, even if the main computation or any other assertion evaluated earlier fails. A mutable variable of type `IORef` implements the global assertion store. Finally `unsafePerformIO :: IO a -> a` enables us to implement `assert` using exceptions and mutable variables [7].

Properties of the Implementation. This simple implementation does not prevent an assertion from evaluating a test argument further than the main computation did. Because assertion checking is delayed, over-evaluation cannot disturb the main computation, but it can cause run-time errors or non-termination in the evaluation of an assertion (see Section 2).

4.2 Avoiding Over-Evaluating

To avoid over-evaluation do we need any non-portable "function" for testing if an expression is evaluated? No, exceptions and the function `unsafePerformIO` are enough. We can borrow and extend a technique from the Haskell Object Observation Debugger (HOOD) [4]. We arrange that as evaluation of the underlying program demands the value of an expression wrapped with an assertion, the main computation makes a copy of the value. Thus the copy comprises exactly those parts of the value that were demanded by the evaluation of the underlying program.

We introduce two new functions, `demand` and `listen`. The function `demand` is wrapped around the value that is consumed by the main computation. The function returns that value and, whenever a part of the value is demanded, the function also adds the demanded part to the copy. The assertion uses the result of the function `listen`. The function `listen` simply returns the copy; because `listen` is only evaluated after the main computation has terminated, `listen` returns those parts of the value that were demanded by the main computation. If the result of `listen` is evaluated further, then it raises an exception. For every part of a value there is a `demand`/`listen` pair that communicates via an `IORef`. The value of the `IORef` is `Unblocked v` to pass a value `v` (weak head normal form) or `Blocked` to indicate that the value was not (yet) demanded. The implementation of `demand` is specific for every type. Hence we introduce a class `Assert` and the type of `assert` becomes `Assert a => String -> (a -> Bool) -> a -> a`. Appendix A gives the details of the implementation.

Properties of the Implementation. An assertion can use exactly those parts of values that are evaluated by the main computation, no less, no more. However, if an assertion fails, the programmer is informed rather late; because of the problem actually detected by the assertion, the main computation may have run into a run-time error or worse a loop. The computation is then also likely to produce a long, fortunately ordered, list of failed assertions. A programmer wants to know about a failed assertion before the main computation uses the faulty value!

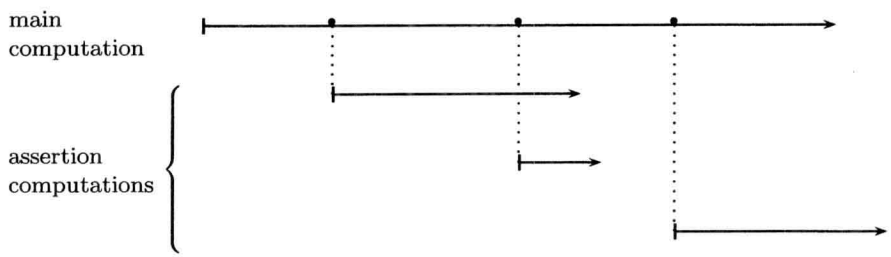


Fig. 3. Concurrent Assertions in Time.

4.3 Concurrent Assertions

How can we evaluate assertions as eagerly as possible yet still only using data that is demanded by the main computation? Rather than delaying assertion checking to the end, we can evaluate each assertion in a separate thread concurrently to the main computation. We require a further extension of Haskell 98: Concurrent Haskell [7].

Figure 3 illustrates the idea. Each evaluation of `assert` in the main computation starts a new thread for evaluating the assertion itself. As before, the value tested by an assertion is copied as it is demanded by the main computation and the copy is used by the assertion. Replacing the `IOVar` shared by a `demand/listen` pair by an `MVar` synchronises the assertion thread with the demand of the main computation. The assertion thread has to wait when it tries to evaluate parts of the copy that do not (yet) exist.

Properties of the Implementation. Concurrency ensures that even if the main computation runs into an infinite loop, a failed assertion will be reported. In general failed assertions may be reported earlier. However, there is no guarantee, because the scheduler is free to evaluate assertions at any time. They may – and in practice often are – evaluated after the main computation has terminated.

4.4 Priority of Assertions

To solve the problem we need to give assertion threads priority over the main computation. Unfortunately Concurrent Haskell does not provide threads with different priorities. However, coroutining enables us to give priority to assertions. We explicitly pass control between each assertion thread and the main thread. When an assertion demands a part of a value that has not yet been demanded by the main computation, the assertion thread is blocked and control is passed to the main thread. Whenever the main thread demands another part of the tested value and an assertion thread is waiting for that value, the main thread is blocked and control is passed to the assertion thread. Thus the assertion always gets a new part of the value for testing *before* it is used by the main computation. Figure 4 illustrates the idea and Appendix B gives the details of the implementation which uses semaphores to pass control.