Manuel Hermenegildo
Daniel Cabeza (Eds.)

LNCS 3350

Practical Aspects of Declarative Languages

7th International Symposium, PADL 2005
Long Beach, CA, USA, January 2005
Proceedings

Springer

Manuel Hermenegildo   Daniel Cabeza (Eds.)

Practical Aspects of Declarative Languages

7th International Symposium, PADL 2005
Long Beach, CA, USA, January 10-11, 2005
Proceedings

Springer

Volume Editors

Manuel Hermenegildo
University of New Mexico, Department of Computer Science
MSC 01 1130, Albuquerque, NM 87131, USA
E-mail: herme@unm.edu
and
Technical University of Madrid, Department of Computer Science
28660 Boadilla del Monte, Madrid, Spain
E-mail: herme@fi.upm.es

Daniel Cabeza
Technical University of Madrid, Department of Computer Science
28660 Boadilla del Monte, Madrid, Spain
E-mail: dcabeza@fi.upm.es

Library of Congress Control Number: 2004117186

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

ISSN 0302-9743
ISBN 3-540-24362-3 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springeronline.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11377474 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 3350

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

Preface

The International Symposium on Practical Aspects of Declarative Languages (PADL) 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. Declarative languages build on sound theoretical foundations to provide attractive frameworks for application development. These languages have been successfully applied to a wide array of different real-world situations, including database management, active networks, software engineering, decision support systems, or music composition; whereas new developments in theory and implementation have opened up new application areas. Inversely, applications often drive the progress in the theory and implementation of declarative systems, as well as benefit from this progress.

The 7th PADL Symposium was held in Long Beach, California on January 10–11, 2005, and was co-located with ACM's Principles of Programming Languages (POPL). From 36 submitted papers, the Program Committee selected 17 papers for presentation at the symposium, based upon at least three reviews for each paper, provided from Program Committee members and additional referees.

Two invited talks were presented at the conference: one by Norman Ramsey (Harvard University) entitled "Building the World from First Principles: Declarative Machine Descriptions and Compiler Construction"; and a second by Saumya Debray (University of Arizona) entitled "Code Compression."

Following what has become a tradition in PADL symposia, the Program Committee selected one paper to receive the "Most Practical Paper" award. This year the paper judged the best in terms of practicality, originality, and clarity was "A Provably Correct Compiler for Efficient Model Checking of Mobile Processes," by Ping Yang, Yifei Dong, C.R. Ramakrishnan, and Scott A. Smolka. This paper presents an optimizing compiler for the pi-calculus that improves the efficiency of model-checking specifications in a logic-programming-based model checker.

The PADL symposium series is sponsored in part by the Association for Logic Programming (`http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/`) and COMPULOG Americas (`http://www.cs.nmsu.edu/~complog/`). Thanks are also due to the University of Texas at Dallas for its support. Finally, we want to thank the authors who submitted papers to PADL 2005 and all who participated in the conference.

November 2004

Manuel Hermenegildo
Daniel Cabeza

Program Chairs

Manuel Hermenegildo	University of New Mexico, USA *and* Technical University of Madrid, Spain
Daniel Cabeza	Technical University of Madrid, Spain

Program Committee

Kenichi Asai	Ochanomizu University, Japan
Manuel Carro	Technical University of Madrid, Spain
Bart Demoen	K.U.Leuven, Belgium
Robert Findler	University of Chicago, USA
John Gallagher	Roskilde University, Denmark
Hai-Feng Guo	University of Nebraska at Omaha, USA
Gopal Gupta	U. of Texas at Dallas, USA (General Chair)
Chris Hankin	Imperial College London, UK
Joxan Jaffar	National U. of Singapore, Singapore
Alan Mycroft	Cambridge University, UK
Gopalan Nadathur	U. of Minnesota, USA
Lee Naish	U. of Melbourne, Australia
Simon Peyton-Jones	Microsoft Research, USA
John Reppy	University of Chicago, USA
Morten Rhiger	Roskilde University, Denmark
Francesca Rossi	University of Padova, Italy
Vitor Santos-Costa	U. Federal do Rio de Janeiro, Brazil
Terrance Swift	S.U. of New York at Stony Brook, USA
David S. Warren	S.U. of New York at Stony Brook, USA

Referees

Maurice Bruynooghe	Ricardo Lopes	Tom Schrijvers
Ins de Castro Dutra	Noelia Maya	David Scott
Chiyan Chen	Dale Miller	Mark Shinwell
Henning Christiansen	Rudradeb Mitra	Leon Sterling
Gregory Cooper	Andrew Moss	Tom Stuart
Yifei Dong	Pablo Nogueira	Peter Stuckey
Mrio Florido	Michael O'Donnell	Eric Van Wyk
David Greaves	Bernard Pope	Kristen Brent Venable
ngel Herranz	Ricardo Rocha	Joost Vennekens
Bharat Jayaraman	Mads Rosendahl	Razvan Voicu
Siau-Cheng Khoo	Abhik Roychoudhury	Hongwei Xi

Lecture Notes in Computer Science

For information about Vols. 1–3257

please contact your bookseller or Springer

Vol. 3385: R. Cousot (Ed.), Verification, Model Checking, and Abstract Interpretation. XII, 483 pages. 2004.

Vol. 3382: J. Odell, P. Giorgini, J.P. Müller (Eds.), Agent-Oriented Software Engineering V. X, 239 pages. 2004.

Vol. 3381: M. Bieliková, B. Charon-Bost, O. Sýkora, P. Vojtáš (Eds.), SOFSEM 2005: Theory and Practice of Computer Science. XV, 428 pages. 2004.

Vol. 3363: T. Eiter, L. Libkin (Eds.), Database Theory - ICDT 2005. XI, 413 pages. 2004.

Vol. 3358: J. Cao, L.T. Yang, M. Guo, F. Lau (Eds.), Parallel and Distributed Processing and Applications. XXIV, 1058 pages. 2004.

Vol. 3356: G. Das, V.P. Gulati (Eds.), Intelligent Information Technology. XII, 428 pages. 2004.

Vol. 3353: J. Hromkovič, M. Nagl, B. Westfechtel (Eds.), Graph-Theoretic Concepts in Computer Science. XI, 404 pages. 2004.

Vol. 3350: M. Hermenegildo, D. Cabeza (Eds.), Practical Aspects of Declarative Languages. VIII, 269 pages. 2004.

Vol. 3348: A. Canteaut, K. Viswanathan (Eds.), Progress in Cryptology - INDOCRYPT 2004. XIV, 431 pages. 2004.

Vol. 3347: R.K. Ghosh, H. Mohanty (Eds.), Distributed Computing and Internet Technology. XX, 472 pages. 2004.

Vol. 3344: J. Malenfant, B.M. Østvold (Eds.), Object-Oriented Technology. ECOOP 2004 Workshop Reader. VIII, 215 pages. 2004.

Vol. 3341: R. Fleischer, G. Trippen (Eds.), Algorithms and Computation. XVII, 935 pages. 2004.

Vol. 3340: C.S. Calude, E. Calude, M.J. Dinneen (Eds.), Developments in Language Theory. XI, 431 pages. 2004.

Vol. 3339: G.I. Webb, X. Yu (Eds.), AI 2004: Advances in Artificial Intelligence. XXII, 1272 pages. 2004. (Subseries LNAI).

Vol. 3338: S.Z. Li, J. Lai, T. Tan, G. Feng, Y. Wang (Eds.), Advances in Biometric Person Authentication. XVIII, 699 pages. 2004.

Vol. 3337: J.M. Barreiro, F. Martin-Sanchez, V. Maojo, F. Sanz (Eds.), Biological and Medical Data Analysis. XI, 508 pages. 2004.

Vol. 3336: D. Karagiannis, U. Reimer (Eds.), Practical Aspects of Knowledge Management. X, 523 pages. 2004. (Subseries LNAI).

Vol. 3334: Z. Chen, H. Chen, Q. Miao, Y. Fu, E. Fox, E.-p. Lim (Eds.), Digital Libraries: International Collaboration and Cross-Fertilization. XX, 690 pages. 2004.

Vol. 3333: K. Aizawa, Y. Nakamura, S. Satoh (Eds.), Advances in Multimedia Information Processing - PCM 2004, Part III. XXXV, 785 pages. 2004.

Vol. 3332: K. Aizawa, Y. Nakamura, S. Satoh (Eds.), Advances in Multimedia Information Processing - PCM 2004, Part II. XXXVI, 1051 pages. 2004.

Vol. 3331: K. Aizawa, Y. Nakamura, S. Satoh (Eds.), Advances in Multimedia Information Processing - PCM 2004, Part I. XXXVI, 667 pages. 2004.

Vol. 3329: P.J. Lee (Ed.), Advances in Cryptology - ASIACRYPT 2004. XVI, 546 pages. 2004.

Vol. 3328: K. Lodaya, M. Mahajan (Eds.), FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science. XVI, 532 pages. 2004.

Vol. 3326: A. Sen, N. Das, S.K. Das, B.P. Sinha (Eds.), Distributed Computing - IWDC 2004. XIX, 546 pages. 2004.

Vol. 3323: G. Antoniou, H. Boley (Eds.), Rules and Rule Markup Languages for the Semantic Web. X, 215 pages. 2004.

Vol. 3322: R. Klette, J. Žunić (Eds.), Combinatorial Image Analysis. XII, 760 pages. 2004.

Vol. 3321: M.J. Maher (Ed.), Advances in Computer Science - ASIAN 2004. XII, 510 pages. 2004.

Vol. 3320: K.-M. Liew, H. Shen, S. See, W. Cai (Eds.), Parallel and Distributed Computing: Applications and Technologies. XXIV, 891 pages. 2004.

Vol. 3316: N.R. Pal, N.K. Kasabov, R.K. Mudi, S. Pal, S.K. Parui (Eds.), Neural Information Processing. XXX, 1368 pages. 2004.

Vol. 3315: C. Lemaître, C.A. Reyes, J.A. González (Eds.), Advances in Artificial Intelligence – IBERAMIA 2004. XX, 987 pages. 2004. (Subseries LNAI).

Vol. 3314: J. Zhang, J.-H. He, Y. Fu (Eds.), Computational and Information Science. XXIV, 1259 pages. 2004.

Vol. 3312: A.J. Hu, A.K. Martin (Eds.), Formal Methods in Computer-Aided Design. XI, 445 pages. 2004.

Vol. 3311: V. Roca, F. Rousseau (Eds.), Interactive Multimedia and Next Generation Networks. XIII, 287 pages. 2004.

Vol. 3309: C.-H. Chi, K.-Y. Lam (Eds.), Content Computing. XII, 510 pages. 2004.

Vol. 3308: J. Davies, W. Schulte, M. Barnett (Eds.), Formal Methods and Software Engineering. XIII, 500 pages. 2004.

Vol. 3307: C. Bussler, S.-k. Hong, W. Jun, R. Kaschek, D.. Kinshuk, S. Krishnaswamy, S.W. Loke, D. Oberle, D. Richards, A. Sharma, Y. Sure, B. Thalheim (Eds.), Web Information Systems – WISE 2004 Workshops. XV, 277 pages. 2004.

Vol. 3306: X. Zhou, S. Su, M.P. Papazoglou, M.E. Orlowska, K.G. Jeffery (Eds.), Web Information Systems – WISE 2004. XVII, 745 pages. 2004.

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

Vol. 3303: J.A. López, E. Benfenati, W. Dubitzky (Eds.), Knowledge Exploration in Life Science Informatics. X, 249 pages. 2004. (Subseries LNAI).

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

Vol. 3300: L. Bertossi, A. Hunter, T. Schaub (Eds.), Inconsistency Tolerance. VII, 295 pages. 2004.

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

Vol. 3298: S.A. McIlraith, D. Plexousakis, F. van Harmelen (Eds.), The Semantic Web – ISWC 2004. XXI, 841 pages. 2004.

Vol. 3296: L. Bougé, V.K. Prasanna (Eds.), High Performance Computing - HiPC 2004. XXV, 530 pages. 2004.

Vol. 3295: P. Markopoulos, B. Eggen, E. Aarts, J.L. Crowley (Eds.), Ambient Intelligence. XIII, 388 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, Part II. XXV, 824 pages. 2004.

Vol. 3290: R. Meersman, Z. Tari (Eds.), On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, Part I. 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. Martínez 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. 3285: S. Manandhar, J. Austin, U.B. Desai, Y. Oyanagi, A. Talukder (Eds.), Applied Computing. XII, 334 pages. 2004.

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

Vol. 3283: F.A. Aagesen, C. Anutariya, V. Wuwongse (Eds.), Intelligence in Communication Systems. XIII, 327 pages. 2004.

Vol. 3282: V. Guruswami, List Decoding of Error-Correcting Codes. XIX, 350 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. 3279: G.M. Voelker, S. Shenker (Eds.), Peer-to-Peer Systems III. XI, 300 pages. 2004.

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

Vol. 3275: P. Perner (Ed.), Advances in Data Mining. VIII, 173 pages. 2004. (Subseries LNAI).

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. 3272: L. Baresi, S. Dustdar, H. Gall, M. Matera (Eds.), Ubiquitous Mobile Information and Collaboration Systems. VIII, 197 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. 3268: W. Lindner, M. Mesiti, C. Türker, Y. Tzitzikas, A. Vakali (Eds.), Current Trends in Database Technology - EDBT 2004 Workshops. XVIII, 608 pages. 2004.

Vol. 3267: C. Priami, P. Quaglia (Eds.), Global Computing. VIII, 377 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. 3259: J. Dix, J. Leite (Eds.), Computational Logic in Multi-Agent Systems. XII, 251 pages. 2004. (Subseries LNAI).

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

Table of Contents

Invited Talks

Papers

Building the World from First Principles: Declarative Machine Descriptions and Compiler Construction
(Abstract)

Norman Ramsey

Division of Engineering and Applied Sciences
Harvard University
http://www.eecs.harvard.edu/~nr

For at least 25 years, the most effective way to retarget systems software has been by using machine descriptions. But "machine description" doesn't mean what you think. A traditional machine description does contain information about the machine, but its utility is compromised in one of two ways:

– The description is useful only in support of a particular algorithm, such as instruction-set emulation, LR parsing, or bottom-up tree matching.
– Information about the machine is inextricably intertwined with information about a particular tool's internal representation, such as a compiler's intermediate code.

The result is that a machine description used to build one tool – a compiler, assembler, linker, debugger, disassembler, emulator, simulator, binary translator, executable editor, verification-condition generator, or what have you – is typically useless for any other purpose. Another difficulty is that to write a machine description, you have to be a double expert: for example, to write the machine description used to retarget a compiler, you must know not only about the target machine but also about the internals of the compiler.

My colleagues, my students, and I have been exploring an alternative: the *declarative machine description.*

– It tries to favor no algorithm over any other.
– It is independent of any tool's internal representation, and indeed, independent of any tool's implementation language.
– It describes only properties of the machine, preferably in a way that is designed for *analysis*, not for execution.

We are focusing on properties that are used in the construction of systems software. We have three long-term goals:

– Declarative machine descriptions should be *reusable.* That is, from just a few descriptions of a machine, we want to build *all* of the software needed to support that machine.

M. Hermenegildo and D. Cabeza (Eds.): PADL 2005, LNCS 3350, pp. 1–4, 2005.

- Declarative machine descriptions should *decouple machine knowledge from tool knowledge.* That is, if you know all about a machine, you should be able to retarget a tool by writing a description of the machine, even if you know nothing about the tool.
- Declarative machine descriptions should *meet the hardware halfway.* That is, our descriptions should be provably consistent with the formal descriptions used to synthesize hardware.

We can realize these benefits only if we can solve a hard problem: instead of relying on a human programmer to apply machine knowledge to the construction of a particular tool, we must somehow build tool knowledge into a program generator that can read a machine description and generate the tool[1]. For example, in our machine-description language SLED, we specify encoding and decoding of machine instructions declaratively, by sets of equations. We then use an equation solver to generate encoders and decoders (assemblers and disassemblers) by applying two kinds of tool knowledge: knowledge about relocatable object code and knowledge about decoding algorithms.

All of which brings us to the title of this talk. What if, instead of writing a code generator in a domain-specific language and calling the result a machine description, we start with a true declarative machine description and build the code generator from first principles? What *are* the first principles? What kinds of tool knowledge are neded to generate a code generator? Why is the problem hard?

We start with a simple, declarative machine description that answers two questions:

- What is the mutable state of the machine?
- When an instruction is executed, how does that state change?

Given answers to these questions, building a simulator is straightforward. But to build a compiler, we must be able to take a source program, understand its semantics in terms of state change, then find a sequence of machine instructions implementing that state change. This problem lies at the hard of building not only a compiler but also many other tools: we must somehow generalize and invert the information in the machine description.

The inversion problem has lain fallow for years. The key insight we bring is that a code generator based on inversion need not produce *good* code – it is enough to produce *correct* code. We know this because of the work of Jack Davidson and his colleagues, who developed the following compilation strategy:

- Generate very naïve code
- Improve the code *under the invariant* that every node in the flow graph can be represented by a single instruction on the target machine.

[1] Program generators often dodge this problem by allowing a machine description to "escape" to hand-written code. But hand-written code used to build one tool is likely to be useless in building another, and especially if it contains library calls, hand-written code can be nearly impossible to analyze.

This simple strategy leads to very good machine code, and it has been applied successfully in the *po*, *vpo*, and *gcc* compilers.

Using Davidson's compilation strategy, we need to read a machine description and generate four components:

- A register allocator, to map temporaries to machine registers
- A "code expander," to select machine instructions
- A "recognizer," to maintain the single-instruction invariant
- An emitter, to emit assembly language for each instruction

The talk will describe these components and how we can hope to generate them from declarative machine descriptions. Our work is still very much in progress, but we have two reasons for optimism:

- We don't need descriptions of very many properties.
- We get a lot of mileage from one idea: *binding time.*

We also hope to be able to take machine-specific human knowledge and capture it as universal truths of mathematics, which will then enable us to apply that knowledge to new machines.

References

Manuel E. Benitez and Jack W. Davidson. 1988 (July). A portable global optimizer and linker. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices,* 23(7):329–338.

Jack W. Davidson and Christopher W. Fraser. 1984 (October). Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems,* 6(4):505–526.

Lee D. Feigenbaum. 2001 (April). Automated translation: Generating a code generator. Technical Report TR-12-01, Harvard University, Computer Science Technical Reports.

Mary F. Fernández and Norman Ramsey. 1997 (May). Automatic checking of instruction specifications. In *Proceedings of the International Conference on Software Engineering,* pages 326–336.

Norman Ramsey. 1996 (May)a. Relocating machine instructions by currying. *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation,* in *SIGPLAN Notices,* 31(5):226–236.

Norman Ramsey. 1996 (April)b. A simple solver for linear equations containing nonlinear operators. *Software - Practice & Experience,* 26(4):467–487.

Norman Ramsey. 2003 (May). Pragmatic aspects of reusable program generators. *Journal of Functional Programming,* 13(3):601–646. A preliminary version of this paper appeared in *Semantics, Application, and Implementation of Program Generation,* LNCS 1924, pages 149–171.

Norman Ramsey and Cristina Cifuentes. 2003 (March). A transformational approach to binary translation of delayed branches. *ACM Transactions on Programming Languages and Systems,* 25(2):210–224.

Norman Ramsey and Jack W. Davidson. 1998 (June). Machine descriptions to build tools for embedded systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98),* volume 1474 of *LNCS,* pages 172–188. Springer Verlag.

Norman Ramsey, Jack W. Davidson, and Mary F. Fernández. 2001. Design principles for machine-description languages. Unpublished draft available from http://www.eecs.harvard.edu/ nr/pubs/desprin-abstract.html.

Norman Ramsey and Mary F. Fernández. 1995 (January). The New Jersey Machine-Code Toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, New Orleans, LA.

Norman Ramsey and Mary F. Fernández. 1997 (May). Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524.

Kevin Redwine and Norman Ramsey. 2004 (April). Widening integer arithmetic. In *13th International Conference on Compiler Construction (CC 2004)*, volume 2985 of *LNCS*, pages 232–249.

Michael D. Smith, Norman Ramsey, and Glenn Holloway. 2004 (June). A generalized algorithm for graph-coloring register allocation. *ACM SIGPLAN '04 Conference on Programming Language Design and Implementation*, in *SIGPLAN Notices*, 39(6): 277–288.

Code Compression
(Abstract)

Saumya Debray

Department of Computer Science
University of Arizona
Tucson, AZ 85721
debray@cs.arizona.edu

Increasingly, we see a trend where programmeable processors are incorporated into a wide variety of everyday devices, ranging from "smart badges," copy and fax machines, phones, and automobiles to traffic lights and wireless sensor networks. At the same time, the functionality expected of the software deployed on such processors becomes increasingly complex (e.g., general-purpose operating systems such as Linux on cell phones, intrusion-detection and related security security measures on wireless sensor devices). The increasing complexity of such software, and the reliability expected of them, suggest a plausible application of declarative languages. However, programs in declarative languages very often experience a significant increase in code size when they are compiled down to native code. This can be a problem in situations where the amount of memory available is limited. This talk discusses a number of different techniques for reducing the memory footprint of executables.

We begin with a discussion of classical compiler optimizations that can be used to reduce the size of the generated code. While such optimizations have traditionally focused on improving execution speed, they can be adapted quite easily to use code size as the optimization criterion instead. Especially effective are optimizations such as dead and unreachable code elimination, as well as targeted function inlining (e.g., where the callee has exactly one call site, or where inlining a function results in the elimination of so many instructions that the resulting code is smaller than the original). These optimizations can be made even more effective via aggressive interprocedural constant propagation and alias analysis, since this can propagate information from the call sites of a function into its body, potentially allowing conditionals in the body to be evaluated statically, thus making it possible to identify more of the code as unreachable.

Further code size reduction is possible using various techniques for *code factoring*, which aims to reduce code size by getting rid of repeated code fragments. This is, in essence, simply an application of procedural abstraction: repeated occurrences of a code sequence at various locations in a program are replaced by a single instance of that code that is instead called from those locations. For this to be effective, it is necessary to be able to handle code sequences that are similar but may not be identical. We currently sue a low-level approach to dealing with this, via register renaming at the basic block level. An alternative would be to

M. Hermenegildo and D. Cabeza (Eds.): PADL 2005, LNCS 3350, pp. 5–6, 2005.

use some sort of partial tree matching on a higher level program representation such as syntax trees.

Classical optimizations, coupled with code factoring, gives code size reductions of around 30% on average. The main reason this value is not higher is the constraint that the code be maintained in executable form. We can relax this constraint by keeping code in a non-executable compressed form, and decompressing it on the fly into a runtime buffer when needed. The main drawback here is the runtime cost of decompression, which can be quite substantial. Fortunately, most programs follow the so-called "80-20 rule," which states in essence that most of a program's time is spent executing a small portion of its code; a corollary is that most of a program's code is executed only infrequently, if at all. Judicious use of profile information to guide the selection of which code is decompressed at runtime yields additional code size reductions of about 15% on average, with runtime overheads of around 4%.

An orthogonal direction to code size reduction involves dynamic code mutation. The idea here is to identify a set of "similar" code fragments and keep just one representative copy of their code. At runtime, we simply edit the text section of the executable to change the code of the representative appropriately to construct the code fragment that is needed. The runtime mutations are carried out by a "code editor" that is driven by an edit script that describes the edits necessary to change one code fragment into another. This is conceptually similar to classical sequence alignment, except that in our case the edits are carried out *in situ*, which makes insertion operations very expensive. We use clustering algorithms driven by a notion of "distance" between code fragments that aims to estimate the cost of editing one sequence to construct another. Initial experiments suggest that such an approach may be useful for constructs such as C++ templates.

Functional Framework for Sound Synthesis

Jerzy Karczmarczuk

Dept. of Computer Science, University of Caen, France
karczma@info.unicaen.fr

Abstract. We present an application of functional programming in the domain of sound generation and processing. We use the lazy language Clean to define purely functional stream generators, filters and other processors, such as reverberators. Audio signals are represented (before the final output to arrays processed by the system primitives) as co-recursive lazy streams, and the processing algorithms have a strong dataflow taste. This formalism seems particularly appropriate to implement the 'waveguide', or 'physically-oriented' sound models. Lazy programming allocates the dynamical memory quite heavily, so we do not propose a real-time, industrial strength package, but rather a pedagogical library, offering natural, easy to understand coding tools. We believe that, thanks to their simplicity and clearness, such functional tools can be also taught to students interested in audio processing, but with a limited competence in programming.

Keywords: Lazy streams, Sounds, DSP, Clean.

1 Introduction

The amplitude of a sound (for one channel) may be thought of as a real function f of time t, and it is fascinating how much structural information it may contain [1]. In order to produce some audible output, this function must be sampled, and transformed into a *signal*, and this is the basic data type we shall work on. Sound may be represented at many different levels, and if one is interested in the structure of sequences of musical events, chords, phrases, etc., there is no need to get down to the digital signal processing primitives. It may seem more interesting and fruitful to speak about the algebra of musical events, music combinators, etc. This was the idea of Haskore [2], whose authors used Haskell to define and to construct a whole spectrum of musical "implementable abstractions". Haskore deals with high-level musical structures, and consigns the low-level, such as the interpretation of the MIDI streams, or the spectral structure of sounds to some back-end applications, MIDI players or CSound [3].

We decided to use the functional approach for the specification and the coding of this "low end" sound generation process. This is usually considered a highly numerical domain involving filter and wave-guide design [4], Fourier analysis, some phenomenological "magic" of the Frequency Modulation approach [5], or some models based on simplified physics, such as the Karplus-Strong algorithm [6] for the plucked string, and its extensions. But the generation and transformation of sounds is a *constructive* domain, dealing with complex abstractions (such as timbre, reverberation, etc.), and it may be based on a specific algebra as well. A possible application of functional programming paradigms as representation and implementation tools seems quite natural.

M. Hermenegildo and D. Cabeza (Eds.): PADL 2005, LNCS 3350, pp. 7–21, 2005.