Joe Hurd
Tom Melham (Eds.)

LNCS 3603

Theorem Proving in Higher Order Logics

18th International Conference, TPHOLs 2005
Oxford, UK, August 2005
Proceedings

Springer

Joe Hurd Tom Melham (Eds.)

Theorem Proving in Higher Order Logics

18th International Conference, TPHOLs 2005
Oxford, UK, August 22-25, 2005
Proceedings

Springer

Volume Editors

Joe Hurd
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK
E-mail: joe.hurd@comlab.ox.ac.uk

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

Library of Congress Control Number: 2005930490

CR Subject Classification (1998): F.4.1, I.2.3, F.3.1, D.2.4, B.6.3

ISSN 0302-9743
ISBN-10 3-540-28372-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28372-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11541868 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 3603

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

Editorial Board

Lecture Notes in Computer Science

For information about Vols. 1–3522

please contact your bookseller or Springer

Vol. 3654: S. Jajodia, D. Wijesekera (Eds.), Data and Applications Security XIX. X, 353 pages. 2005.

Vol. 3639: P. Godefroid (Ed.), Model Checking Software. XI, 289 pages. 2005.

Vol. 3638: A. Butz, B. Fisher, A. Krüger, P. Olivier (Eds.), Smart Graphics. XI, 269 pages. 2005.

Vol. 3634: L. Ong (Ed.), Computer Science Logic. XI, 567 pages. 2005.

Vol. 3633: C. Bauzer Medeiros, M. Egenhofer, E. Bertino (Eds.), Advances in Spatial and Temporal Databases. XIII, 433 pages. 2005.

Vol. 3632: R. Nieuwenhuis (Ed.), Automated Deduction – CADE-20. XIII, 459 pages. 2005. (Subseries LNAI).

Vol. 3627: C. Jacob, M.L. Pilat, P.J. Bentley, J. Timmis (Eds.), Artificial Immune Systems. XII, 500 pages. 2005.

Vol. 3626: B. Ganter, G. Stumme, R. Wille (Eds.), Formal Concept Analysis. X, 349 pages. 2005. (Subseries LNAI).

Vol. 3625: S. Kramer, B. Pfahringer (Eds.), Inductive Logic Programming. XIII, 427 pages. 2005. (Subseries LNAI).

Vol. 3623: M. Liśkiewicz, R. Reischuk (Eds.), Fundamentals of Computation Theory. XV, 576 pages. 2005.

Vol. 3621: V. Shoup (Ed.), Advances in Cryptology – CRYPTO 2005. XI, 568 pages. 2005.

Vol. 3620: H. Muñoz-Avila, F. Ricci (Eds.), Case-Based Reasoning Research and Development. XV, 654 pages. 2005. (Subseries LNAI).

Vol. 3619: X. Lu, W. Zhao (Eds.), Networking and Mobile Computing. XXIV, 1299 pages. 2005.

Vol. 3615: B. Ludäscher, L. Raschid (Eds.), Data Integration in the Life Sciences. XII, 344 pages. 2005. (Subseries LNBI).

Vol. 3608: F. Dehne, A. López-Ortiz, J.-R. Sack (Eds.), Algorithms and Data Structures. XIV, 446 pages. 2005.

Vol. 3607: J.-D. Zucker, L. Saitta (Eds.), Abstraction, Reformulation and Approximation. XII, 376 pages. 2005. (Subseries LNAI).

Vol. 3606: V. Malyshkin (Ed.), Parallel Computing Technologies. XII, 470 pages. 2005.

Vol. 3603: J. Hurd, T. Melham (Eds.), Theorem Proving in Higher Order Logics. IX, 409 pages. 2005.

Vol. 3602: R. Eigenmann, Z. Li, S.P. Midkiff (Eds.), Languages and Compilers for High Performance Computing. IX, 486 pages. 2005.

Vol. 3598: H. Murakami, H. Nakashima, H. Tokuda, M. Yasumura, Ubiquitous Computing Systems. XIII, 275 pages. 2005.

Vol. 3597: S. Shimojo, S. Ichii, T.W. Ling, K.-H. Song (Eds.), Web and Communication Technologies and Internet-Related Social Issues - HSI 2005. XIX, 368 pages. 2005.

Vol. 3596: F. Dau, M.-L. Mugnier, G. Stumme (Eds.), Conceptual Structures: Common Semantics for Sharing Knowledge. XI, 467 pages. 2005. (Subseries LNAI).

Vol. 3595: L. Wang (Ed.), Computing and Combinatorics. XVI, 995 pages. 2005.

Vol. 3594: J.C. Setubal, S. Verjovski-Almeida (Eds.), Advances in Bioinformatics and Computational Biology. XIV, 258 pages. 2005. (Subseries LNBI).

Vol. 3592: S. Katsikas, J. Lopez, G. Pernul (Eds.), Trust and Privacy in Digital Business. XII, 332 pages. 2005.

Vol. 3587: P. Perner, A. Imiya (Eds.), Machine Learning and Data Mining in Pattern Recognition. XVII, 695 pages. 2005. (Subseries LNAI).

Vol. 3586: A.P. Black (Ed.), ECOOP 2005 - Object-Oriented Programming. XVII, 631 pages. 2005.

Vol. 3584: X. Li, S. Wang, Z.Y. Dong (Eds.), Advanced Data Mining and Applications. XIX, 835 pages. 2005. (Subseries LNAI).

Vol. 3583: R.W. H. Lau, Q. Li, R. Cheung, W. Liu (Eds.), Advances in Web-Based Learning – ICWL 2005. XIV, 420 pages. 2005.

Vol. 3582: J. Fitzgerald, I.J. Hayes, A. Tarlecki (Eds.), FM 2005: Formal Methods. XIV, 558 pages. 2005.

Vol. 3581: S. Miksch, J. Hunter, E. Keravnou (Eds.), Artificial Intelligence in Medicine. XVII, 547 pages. 2005. (Subseries LNAI).

Vol. 3580: L. Caires, G.F. Italiano, L. Monteiro, C. Palamidessi, M. Yung (Eds.), Automata, Languages and Programming. XXV, 1477 pages. 2005.

Vol. 3579: D. Lowe, M. Gaedke (Eds.), Web Engineering. XXII, 633 pages. 2005.

Vol. 3578: M. Gallagher, J. Hogan, F. Maire (Eds.), Intelligent Data Engineering and Automated Learning - IDEAL 2005. XVI, 599 pages. 2005.

Vol. 3577: R. Falcone, S. Barber, J. Sabater-Mir, M.P. Singh (Eds.), Trusting Agents for Trusting Electronic Societies. VIII, 235 pages. 2005. (Subseries LNAI).

Vol. 3576: K. Etessami, S.K. Rajamani (Eds.), Computer Aided Verification. XV, 564 pages. 2005.

Vol. 3575: S. Wermter, G. Palm, M. Elshaw (Eds.), Biomimetic Neural Learning for Intelligent Robots. IX, 383 pages. 2005. (Subseries LNAI).

Vol. 3574: C. Boyd, J.M. González Nieto (Eds.), Information Security and Privacy. XIII, 586 pages. 2005.

Vol. 3573: S. Etalle (Ed.), Logic Based Program Synthesis and Transformation. VIII, 279 pages. 2005.

Vol. 3572: C. De Felice, A. Restivo (Eds.), Developments in Language Theory. XI, 409 pages. 2005.

Vol. 3571: L. Godo (Ed.), Symbolic and Quantitative Approaches to Reasoning with Uncertainty. XVI, 1028 pages. 2005. (Subseries LNAI).

Vol. 3570: A. S. Patrick, M. Yung (Eds.), Financial Cryptography and Data Security. XII, 376 pages. 2005.

Vol. 3569: F. Bacchus, T. Walsh (Eds.), Theory and Applications of Satisfiability Testing. XII, 492 pages. 2005.

Vol. 3568: W.-K. Leow, M.S. Lew, T.-S. Chua, W.-Y. Ma, L. Chaisorn, E.M. Bakker (Eds.), Image and Video Retrieval. XVII, 672 pages. 2005.

Vol. 3567: M. Jackson, D. Nelson, S. Stirk (Eds.), Database: Enterprise, Skills and Innovation. XII, 185 pages. 2005.

Vol. 3566: J.-P. Banâtre, P. Fradet, J.-L. Giavitto, O. Michel (Eds.), Unconventional Programming Paradigms. XI, 367 pages. 2005.

Vol. 3565: G.E. Christensen, M. Sonka (Eds.), Information Processing in Medical Imaging. XXI, 777 pages. 2005.

Vol. 3564: N. Eisinger, J. Małuszyński (Eds.), Reasoning Web. IX, 319 pages. 2005.

Vol. 3562: J. Mira, J.R. Álvarez (Eds.), Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, Part II. XXIV, 636 pages. 2005.

Vol. 3561: J. Mira, J.R. Álvarez (Eds.), Mechanisms, Symbols, and Models Underlying Cognition, Part I. XXIV, 532 pages. 2005.

Vol. 3560: V.K. Prasanna, S. Iyengar, P.G. Spirakis, M. Welsh (Eds.), Distributed Computing in Sensor Systems. XV, 423 pages. 2005.

Vol. 3559: P. Auer, R. Meir (Eds.), Learning Theory. XI, 692 pages. 2005. (Subseries LNAI).

Vol. 3558: V. Torra, Y. Narukawa, S. Miyamoto (Eds.), Modeling Decisions for Artificial Intelligence. XII, 470 pages. 2005. (Subseries LNAI).

Vol. 3557: H. Gilbert, H. Handschuh (Eds.), Fast Software Encryption. XI, 443 pages. 2005.

Vol. 3556: H. Baumeister, M. Marchesi, M. Holcombe (Eds.), Extreme Programming and Agile Processes in Software Engineering. XIV, 332 pages. 2005.

Vol. 3555: T. Vardanega, A.J. Wellings (Eds.), Reliable Software Technology – Ada-Europe 2005. XV, 273 pages. 2005.

Vol. 3554: A. Dey, B. Kokinov, D. Leake, R. Turner (Eds.), Modeling and Using Context. XIV, 572 pages. 2005. (Subseries LNAI).

Vol. 3553: T.D. Hämäläinen, A.D. Pimentel, J. Takala, S. Vassiliadis (Eds.), Embedded Computer Systems: Architectures, Modeling, and Simulation. XV, 476 pages. 2005.

Vol. 3552: H. de Meer, N. Bhatti (Eds.), Quality of Service – IWQoS 2005. XVIII, 400 pages. 2005.

Vol. 3551: T. Härder, W. Lehner (Eds.), Data Management in a Connected World. XIX, 371 pages. 2005.

Vol. 3548: K. Julisch, C. Kruegel (Eds.), Intrusion and Malware Detection and Vulnerability Assessment. X, 241 pages. 2005.

Vol. 3547: F. Bomarius, S. Komi-Sirviö (Eds.), Product Focused Software Process Improvement. XIII, 588 pages. 2005.

Vol. 3546: T. Kanade, A. Jain, N.K. Ratha (Eds.), Audio- and Video-Based Biometric Person Authentication. XX, 1134 pages. 2005.

Vol. 3544: T. Higashino (Ed.), Principles of Distributed Systems. XII, 460 pages. 2005.

Vol. 3543: L. Kutvonen, N. Alonistioti (Eds.), Distributed Applications and Interoperable Systems. XI, 235 pages. 2005.

Vol. 3542: H.H. Hoos, D.G. Mitchell (Eds.), Theory and Applications of Satisfiability Testing. XIII, 393 pages. 2005.

Vol. 3541: N.C. Oza, R. Polikar, J. Kittler, F. Roli (Eds.), Multiple Classifier Systems. XII, 430 pages. 2005.

Vol. 3540: H. Kalviainen, J. Parkkinen, A. Kaarna (Eds.), Image Analysis. XXII, 1270 pages. 2005.

Vol. 3539: K. Morik, J.-F. Boulicaut, A. Siebes (Eds.), Local Pattern Detection. XI, 233 pages. 2005. (Subseries LNAI).

Vol. 3538: L. Ardissono, P. Brna, A. Mitrovic (Eds.), User Modeling 2005. XVI, 533 pages. 2005. (Subseries LNAI).

Vol. 3537: A. Apostolico, M. Crochemore, K. Park (Eds.), Combinatorial Pattern Matching. XI, 444 pages. 2005.

Vol. 3536: G. Ciardo, P. Darondeau (Eds.), Applications and Theory of Petri Nets 2005. XI, 470 pages. 2005.

Vol. 3535: M. Steffen, G. Zavattaro (Eds.), Formal Methods for Open Object-Based Distributed Systems. X, 323 pages. 2005.

Vol. 3534: S. Spaccapietra, E. Zimányi (Eds.), Journal on Data Semantics III. XI, 213 pages. 2005.

Vol. 3533: M. Ali, F. Esposito (Eds.), Innovations in Applied Artificial Intelligence. XX, 858 pages. 2005. (Subseries LNAI).

Vol. 3532: A. Gómez-Pérez, J. Euzenat (Eds.), The Semantic Web: Research and Applications. XV, 728 pages. 2005.

Vol. 3531: J. Ioannidis, A. Keromytis, M. Yung (Eds.), Applied Cryptography and Network Security. XI, 530 pages. 2005.

Vol. 3530: A. Prinz, R. Reed, J. Reed (Eds.), SDL 2005: Model Driven. XI, 361 pages. 2005.

Vol. 3528: P.S. Szczepaniak, J. Kacprzyk, A. Niewiadomski (Eds.), Advances in Web Intelligence. XVII, 513 pages. 2005. (Subseries LNAI).

Vol. 3527: R. Morrison, F. Oquendo (Eds.), Software Architecture. XII, 263 pages. 2005.

Vol. 3526: S. B. Cooper, B. Löwe, L. Torenvliet (Eds.), New Computational Paradigms. XVII, 574 pages. 2005.

Vol. 3525: A.E. Abdallah, C.B. Jones, J.W. Sanders (Eds.), Communicating Sequential Processes. XIV, 321 pages. 2005.

Vol. 3524: R. Barták, M. Milano (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. XI, 320 pages. 2005.

Vol. 3523: J.S. Marques, N. Pérez de la Blanca, P. Pina (Eds.), Pattern Recognition and Image Analysis, Part II. XXVI, 733 pages. 2005.

¥528.64元

Preface

This volume constitutes the proceedings of the *18th International Conference on Theorem Proving in Higher Order Logics* (TPHOLs 2005), which was held during 22–25 August 2005 in Oxford, UK. TPHOLs covers all aspects of theorem proving in higher order logics as well as related topics in theorem proving and verification.

There were 49 papers submitted to TPHOLs 2005 in the full research category, each of which was refereed by at least three reviewers selected by the program committee. Of these submissions, 20 research papers and 4 proof pearls were accepted for presentation at the conference and publication in this volume. In keeping with longstanding tradition, TPHOLs 2005 also offered a venue for the presentation of work in progress, where researchers invited discussion by means of a brief introductory talk and then discussed their work at a poster session. A supplementary proceedings volume was published as a 2005 technical report of the Oxford University Computing Laboratory.

The organizers are grateful to Wolfgang Paul and Andrew Pitts for agreeing to give invited talks at TPHOLs 2005.

The TPHOLs conference traditionally changes continents each year to maximize the chances that researchers from around the world can attend. Starting in 1993, the proceedings of TPHOLs and its predecessor workshops have been published in the Springer Lecture Notes in Computer Science series:

1993 (Canada)	Vol. 780	2000 (USA)	Vol. 1869
1994 (Malta)	Vol. 859	2001 (UK)	Vol. 2152
1995 (USA)	Vol. 971	2002 (USA)	Vol. 2410
1996 (Finland)	Vol. 1125	2003 (Italy)	Vol. 2758
1997 (USA)	Vol. 1275	2004 (USA)	Vol. 3223
1998 (Australia)	Vol. 1479	2005 (UK)	Vol. 3603
1999 (France)	Vol. 1690		

We would like to thank our local organizers Ed Smith and Ashish Darbari for their help in many aspects of planning and running TPHOLs.

Finally, we thank our sponsors: Intel Corporation and the EPSRC UK Network in Computer Algebra.

June 2005

Joe Hurd and Tom Melham
TPHOLs 2005 Chairs

Organization

Program Committee

Mark Aagaard (Waterloo)
David Basin (ETH Zürich)
Ching-Tsun Chou (Intel)
Amy Felty (Ottawa)
Jacques Fleuriot (Edinburgh)
Elsa Gunter (UIUC)
Jason Hickey (Caltech)
Joe Hurd (Oxford)
Thomas Kropf (Tübingen & Bosch)
John Matthews (Galois)
Tobias Nipkow (München)
Christine Paulin-Mohring (Paris Sud)
Frank Pfenning (CMU)
Sofiène Tahar (Concordia)
Clark Barrett (NYU)
Yves Bertot (INRIA)
Thierry Coquand (Chalmers)
Jean-Christophe Filliâtre (Paris Sud)
Jim Grundy (Intel)
John Harrison (Intel)
Peter Homeier (US DoD)
Paul Jackson (Edinburgh)
Pete Manolios (Georgia Tech)
César Muñoz (Nat. Inst. Aerospace)
Sam Owre (SRI)
Lawrence Paulson (Cambridge)
Konrad Slind (Utah)
Burkhart Wolff (ETH Zürich)

Additional Referees

Amr Abdel-Hamid
Behzad Akbarpour
Tamarah Arons
Sylvain Conchon
Pierre Corbineau
Peter Dillinger
Lucas Dixon
Guillaume Dufay
Marcio Gemeiro
Ganesh Gopalakrishnan
Ali Habibi
Hugo Herbelin
Doug Howe
Robert Jones
Roope Kaivola
Felix Klaedtke
Farhad Mehta
Laura Meikle
Julien Narboux
Lee Pike
Tom Ridge
Hassen Saidi
Christelle Scharff
N. Shankar
Radu Siminiceanu
Sudarshan Srinivasan
Ashish Tiwari
Daron Vroon

Table of Contents

Invited Papers

Regular Papers

Proof Pearls

On the Correctness of Operating System Kernels

Mauro Gargano*, Mark Hillebrand*, Dirk Leinenbach*,**, and Wolfgang Paul

Saarland University, Computer Science Dept., 66123 Saarbrücken, Germany
{gargano, mah, dirkl, wjp}@wjpserver.cs.uni-sb.de

Abstract. The Verisoft project aims at the pervasive formal verification of entire computer systems. In particular, the seamless verification of the *academic system* is attempted. This system consists of hardware (processor and devices) on top of which runs a microkernel, an operating system, and applications. In this paper we define the computation model CVM (communicating virtual machines) in which concurrent user processes interact with a generic microkernel written in C. We outline the correctness proof for concrete kernels, which implement this model. This result represents a crucial step towards the verification of a kernel, e.g. that in the academic system. We report on the current status of the formal verification.

1 Introduction

There is no need to argue about the importance of computer security [1] and operating system security is in the center of computer security. Making operating systems comfortable and at the same time utmost reliable is extremely hard. However, some small and highly reliable operating system kernels, e.g. [2,3,4], have been developed. A reliable kernel opens the way to uncouple the safety-critical applications running under an operating system from the non-critical ones. One runs *two* operating systems under a trusted kernel, a small trusted one for the safety-critical applications and a conventional one for all others. This minimizes the total size of the trusted components. For example, [5] describes a small operating system and Linux running under the L4 microkernel [6].

For critical applications one wishes of course to estimate, how much trust one should put into a system. For this purpose the *common criteria* for information technology security evaluation [7] define a hierarchy of *evaluation assurance levels* EAL-1 to EAL-7. These are disciplines for reviewing, testing / verifying, and documenting systems during and after development. Even the highest assurance level, EAL-7, does not require formal verification of the system implementation. Clearly, the common criteria, in the current revision, stay *behind* the state of the art available at that time: already nine years before Bevier [8] reported on the full formal verification of KIT, a small multitasking operating system kernel written in machine language. KIT implements a fixed number of processes, each occupying a fixed portion of the processor's memory. It provides the following verified services: process scheduling, error handling, message passing, and an interface to asynchronous devices. In terms of complexity, KIT is near to small real-time operating systems like e.g. OSEKTime [9].

* Work partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft project under grant 01 IS C38.

** Work supported by DFG Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 1–16, 2005.

In this paper we outline an approach to the pervasive verification of a considerably more powerful kernel, supporting virtual memory, memory management, system calls, user defined interrupts, etc. We outline substantial parts of its correctness proof. We report on the current status of the formal verification. The results presented in this paper were obtained in and are of crucial importance to the Verisoft project [10], funded by the German Federal Government. Verisoft has the mission to provide the technology for the formal pervasive verification of entire computer systems of industrial complexity.

2 Overview

To handle the design complexity, computer systems are organized in layers some of which are modeled by well established formal models. Examples are (i) the hardware layer that is modeled by switching circuits and memory components, (ii) the machine language layer that is modeled by random access machines [11] with an appropriate instruction set, and (iii) the programming language layer, e.g. for C, is, for operational semantics, modeled by abstract interpreters, also called abstract C machines. Correctness theorems for components of computer systems are often simulation theorems between *adjacent* layers. Processor correctness concerns a simulation between Layers (i) and (ii). Compiler correctness concerns a simulation between Layers (ii) and (iii).

Aiming at formulating and proving a correctness theorem for an operating system kernel we take a similar approach. We introduce an abstract parallel model of computation called *communicating virtual machines* (CVM) that formalizes concurrent user processes interacting with an operating system kernel. In this model user processes are virtual machines, i.e. processors with virtual memory. The so-called *abstract kernel* is represented as an abstract C machine. Beyond the usual C functions the abstract kernel can call a few special functions, called the *CVM primitives*, that alter the configuration of user processes. For instance, there are CVM primitives to increase / decrease the memory size of a user process or to copy data between user processes (and I/O devices).

By linking abstract kernels with a program implementing the CVM functionality we obtain the *concrete kernel*. In particular, the concrete kernel contains the implementation of the CVM primitives and the implementation of handlers for page faults (not visible in the abstract model). A crucial observation is that the concrete kernel *necessarily* contains assembler code because neither processor registers nor user processes are visible in the variables of a C program. Thus the correctness theorem for the concrete kernel will establish a simulation between CVM and Layer (ii) instead of Layer (iii). Since reasoning on assembler level is tedious we minimize its use in the concrete kernel.

The remainder of this paper is structured as follows. In Sect. 3 we define virtual machines and summarize results from [12] on the simulation of virtual machines by physical machines, processors with physical and swap memory. In Sect. 4 we define abstract $C0$ machines and summarize the compiler correctness proof from [13]. In Sect. 5 we define the CVM model using virtual machines to model computation of the user and abstract $C0$ machines to model computation of an abstract kernel. Section 6 sketches the construction of the concrete kernel containing the CVM implementation. We state the correctness proof for the concrete kernel and outline its proof. In Sect. 7 we report on the status of the formal verification. In Sect. 8 we conclude and sketch further work.

3 Virtual Memory Simulation

Let us introduce some notation. We denote bitvectors by $a \in \{0,1\}^n$. Bit j of bitvector a is denoted by $a[j]$, the sub bitvector consisting of bits j to k (with $k < j$) is denoted by $a[j:k]$. The concatenation of two bitvectors $a \in \{0,1\}^n$ and $b \in \{0,1\}^m$ is denoted by $a \circ b \in \{0,1\}^{n+m}$. Occasionally we will abuse notation and identify bitvectors a with their value $\langle a \rangle = \sum_i a[i] \cdot 2^i$ and vice versa. Arithmetic is modulo 2^n. We model memories m as mappings from addresses $a \in \{0,1\}^{32}$ to byte values $m(a) \in \{0,1\}^8$. For natural numbers d we denote by $m_d(a)$ the content of d consecutive memory cells starting at address a, so $m_d(a) = m(a+d-1) \circ \cdots \circ m(a)$.

In the following sub sections we summarize results from [12].

3.1 Virtual Machines

Virtual machines consist of a processor operating on a (uniform) virtual memory. Configurations c_V of virtual machines have the following components:

- $c_V.R \in \{0,1\}^{32}$ for a variety of processor registers R. We consider here pipelined DLX machines [14] with a delayed branch mechanism that is implemented by *two* program counters, called delayed program counter $c_V.DPC \in \{0,1\}^{32}$ and program counter $c_V.PC \in \{0,1\}^{32}$. For details see [15].
- The size $c_V.V$ of the virtual memory measured in pages of 4K bytes. It defines the set of accessible virtual addresses $VA(c_V) = \{a \in \{0,1\}^{32} \mid a < c_V.V \cdot 4\text{K}\}$. We split virtual addresses $va = va[31:0]$ into page index $va.px = va[31:12]$ and byte index $va.bx = va[11:0]$.
- A byte addressable virtual memory $c_V.vm : VA(c_V) \to \{0,1\}^8$.
- A write protection function $c_V.p : VA(c_V) \to \{0,1\}$ that only depends on the page index of virtual addresses. A virtual address va is write protected if $c_V.p(va) = 1$.

Computation of the virtual machine is modeled by the function δ_V that computes for a given configuration c_V its successor configuration c'_V. The virtual machine accesses the memory in the following situations: it reads the memory to fetch instructions and to execute load instructions, it writes the memory to execute store instructions.

However, any access to a virtual address $va \notin VA(c_V)$ or a write access to va with $c_V.p(va) = 1$ is illegal and leads to an exception. For the CVM model (cf. Sect. 5) we do not consider write protected pages and assume $c_V.p(va) = 0$ for all $va \in VA(c_V)$.

Note that the effects of exceptions are not defined in a virtual machine model alone but in an extended context of a virtual machine running under a certain operating system (kernel). Also, the size of the virtual memory $c_V.V$ cannot be changed by the virtual machine itself. This is described in more detail in Sect. 5.

3.2 Physical Machines and Address Translation

Physical machines consist of a processor operating on physical memory and swap memory. Configurations c_P of physical machines have components $c_P.R$ for processor registers R, $c_P.pm$ for the physical memory, and $c_P.sm$ for the swap memory. The physical machine has several special purpose registers not present in virtual machines, e.g. the

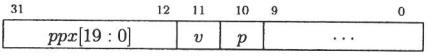

Fig. 1. Page Table Entry

mode register *mode*, the page table origin *pto*, and the page table length *ptl*. Computation of the physical machine is modeled by the next state function δ_{P}.

In system mode, i.e. if $c_{\mathrm{P}}.mode = 0$, the physical machine operates almost like a virtual machine with extra registers. In user mode, i.e. $c_{\mathrm{P}}.mode = 1$, memory accesses are subject to address translation: they either cause a page fault or are redirected to the translated physical memory address $pma(c_{\mathrm{P}}, va)$. The result of address translation depends on the contents of the *page table*, a region of the physical memory starting at address $c_{\mathrm{P}}.pto \cdot 4\mathrm{K}$ with $(c_{\mathrm{P}}.ptl + 1)$ entries of four bytes width.

The page table entry address for virtual address va is defined as $ptea(c_{\mathrm{P}}, va) = c_{\mathrm{P}}.pto \cdot 4\mathrm{K} + 4 \cdot va.px$ and the page table entry of va is defined as $pte(c_{\mathrm{P}}, va) = c_{\mathrm{P}}.pm_4(ptea(c_{\mathrm{P}}, va))$. As shown in Fig. 1, a page table entry consists of three components, the physical page index $ppx(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[31 : 12]$, the valid bit $v(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[11]$, and the write protection bit $p(c_{\mathrm{P}}, va) = pte(c_{\mathrm{P}}, va)[10]$.

On user mode memory access to address va, a page fault is signaling if the page index exceeds the page table length, $va.px > c_{\mathrm{P}}.ptl$, if the page table entry is not valid, $v(c_{\mathrm{P}}, va) = 0$, or if for a write access the write protection is active, $p(c_{\mathrm{P}}, va) = 1$. On page fault the page fault handler, an interrupt service, is invoked.

Without a page fault, the access is performed on the (translated) physical memory address $pma(c_{\mathrm{P}}, va)$ defined as the concatenation of the physical page index and the byte index, $pma(c_{\mathrm{P}}, va) = ppx(c_{\mathrm{P}}, va) \circ va.bx$.

For example, the instruction $I(c_{\mathrm{P}})$ fetched in configuration c_{P} is defined as follows. If $c_{\mathrm{P}}.mode = 0$ we define $I(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_4(c_{\mathrm{P}}.DPC)$, otherwise, provided that there is no page fault, we define $I(c_{\mathrm{P}}) = c_{\mathrm{P}}.pm_4(pma(c_{\mathrm{P}}, c_{\mathrm{P}}.DPC))$.

3.3 Virtual Memory Simulation

A physical machine with appropriate page fault handlers can simulate virtual machines. For a simple page fault handler, virtual memory is stored on the swap memory of the physical machine and the physical memory acts as a write back cache. In addition to the architecturally defined physical memory address $pma(c_{\mathrm{P}}, va)$, the page fault handler maintains a swap memory address function $sma(c_{\mathrm{P}}, va)$.

We use a simulation relation $B(c_{\mathrm{V}}, c_{\mathrm{P}})$ to indicate that a (user mode) physical machine configuration c_{P} encodes virtual machine configuration c_{V}. Essentially, $B(c_{\mathrm{V}},c_{\mathrm{P}})$ is the conjunction of the following three conditions:

- For every page of virtual memory there is a page table entry in the physical machine, $c_{\mathrm{V}}.V = c_{\mathrm{P}}.ptl + 1$.
- The write protection function of the virtual machine is encoded in the page table, $c_{\mathrm{V}}.p(va) = p(c_{\mathrm{P}}, va)$. As noted earlier in this paper we assume $p(c_{\mathrm{P}}, va) = c_{\mathrm{V}}.p(va) = 0$.
- The virtual memory is stored in physical and swap memory: if $v(c_{\mathrm{P}}, va)$ then $c_{\mathrm{V}}.vm(va) = c_{\mathrm{P}}.pm(pma(c_{\mathrm{P}}, va))$, else $c_{\mathrm{V}}.vm(va) = c_{\mathrm{P}}.sm(sma(c_{\mathrm{P}}, va))$.

The simulation theorem for a single virtual machine has the following form:

Theorem 1. *For all computations* $(c_V^0, c_V^1, \dots)$ *of the virtual machine there is a computation* $(c_P^0, c_P^1, \dots)$ *of the physical machine and there are step numbers* $(s(0), s(1), \dots)$ *such that for all* i *and* $S = s(i)$ *we have* $B(c_V^i, c_P^S)$.

Thus step i of the virtual machine is simulated after step $s(i)$ of the physical machine. Even for a simple handlers, the proof is not completely obvious since a single user mode instruction can cause two page faults. To avoid deadlock and guarantee forward progress, the page fault handler must not swap out the page that was swapped in during the last execution of the page fault handler.

3.4 Synchronization Conditions

If the hardware implementation of a physical machine is pipelined, then an instruction $I(c_P^i)$ that is in the memory stage may modify / affect a later instruction $I(c_P^j)$ for $j > i$ after it has been fetched. It may (i) overwrite the instruction itself, (ii) overwrite its page table entry, or (iii) change the mode. In such situations instruction fetch (in particular translated fetch implemented by a memory management unit) would not work correctly. Of course it is possible to detect such data dependencies in hardware and to roll back the computation if necessary. Alternatively, the software to be run on the processor must adhere to certain *software synchronization conventions*. Let $iaddr(c_P^j)$ denote the address of instruction $I(c_P^j)$, possibly translated. If $I(c_P^i)$ writes to address $iaddr(c_P^j)$, then an intermediate instruction $I(c_P^k)$ for $i < k < j$ must drain the pipe. The same must hold if c_P^j is in user mode and $I(c_P^i)$ writes to $ptea(c_P^j, c_P^j.DPC)$. Finally, mode can only be changed to user mode by an `rfe` (return from exception) instruction (and the hardware guarantees that `rfe` instructions drain the pipe).

Conditions of this nature are hypotheses of the hardware correctness proof in [12]. It will be easy to show that they hold for the kernels constructed in Sect. 6.

4 Compilation

We sketch the formal semantics of $C0$, a subset of C, and state the correctness theorem of a $C0$ compiler, summarizing result from [13]. In Section 4.3 we extend the $C0$ semantics to inline assembler code.

4.1 $C0$ Semantics

Eventually we want to consider several programs running under an operating system. The computations of these programs then are interleaved. Therefore our compiler correctness statement is based on a small steps / structured operational semantics [16,17].

In $C0$ types are elementary (*bool*, *int*, ...), pointer types, or composite (*array* or *struct*). A type is called simple if it is an elementary type or a pointer type. We define the (abstract) size of types for simple types t by $size(t) = 1$, for arrays by $size(t[n]) = n \cdot size(t)$, and for structures by $size(struct\{n_1 : t_1, \dots, n_s : t_s\}) = \sum_i size(t_i)$. Values of variables with simple type are called *simple values*. Variables with composite types have *composite values* that are represented flat as a sequence of simple values.

Configuration. An $C0$ machine configuration c_{C0} has the following components:

1. The *program rest* $c_{C0}.pr$. This is a sequence of $C0$ statements which still needs to be executed. In [16] the program rest is called *code component* of the configuration.
2. The *type table* $c_{C0}.tt$ collects information about types used in the program.
3. The *function table* $c_{C0}.ft$ contains information about the functions of a program. It maps function names f to pairs $c_{C0}.ft(f) = (c_{C0}.ft(f).ty, c_{C0}.ft(f).body)$ where $c_{C0}.ft(f).ty$ specifies the types of the arguments, the local variables, and the result of the function, whereas $c_{C0}.ft(f).body$ specifies the function body.
4. The *recursion depth* $c_{C0}.rd$.
5. The *local memory stack* $c_{C0}.lms$. It maps numbers $i \leq c_{C0}.rd$ to memory frames (defined below). The global memory is $c_{C0}.lms(0)$. We denote the top local memory frame of a configuration c_{C0} by $top(c_{C0}) = c_{C0}.lms(c_{C0}.rd)$.
6. A *heap memory* $c_{C0}.hm$. This is also a memory frame.

Memory Frames. We use a relatively explicit, low level memory model in the style of [18]. Memory frames m have the following components: (i) the number $m.n$ of variables in m (for local memory frames this also includes the parameters of the corresponding function definition), (ii) a function $m.name$ mapping variable numbers $i \in [0 : m.n - 1]$ to their names (not used for variables on the heap), (iii) a function $m.ty$ mapping variable numbers to their type. This permits to define the size of a memory frame $size(m)$ as the number of simple values stored in it, namely: $size(m) = \sum_{i=0}^{m.n-1} size(m.ty(i))$. (iv) a content function $m.ct$ mapping indices $0 \leq i < size(m)$ to simple values.

A *variable* of configuration c_{C0} is a pair $v = (m, i)$ where m is a memory frame of c_{C0} and $i < m.n$ is the number of the variable in the frame. The type of a variable (m, i) is defined by $ty((m, i)) = m.ty(i)$.

Sub variables $S = (m, i)s$ are formed from variables (m, i) by appending a *selector* $s = (s_1, \ldots, s_t)$, where each component of a selector has the form $s_i = [j]$ for selecting array element number j or the form $s_i = .n$ for selecting the struct component with name n. If the selector s is consistent with the type of (m, i), then $S = (m, i)s$ is a *sub variable* of (m, i). Selectors are allowed to be empty. In $C0$, pointers p may point to sub variables $(m, i)s$ in the global memory or on the heap. The value of such pointers simply has the form $(m, i)s$. Component $m.ct$ stores the current values $va(c_{C0}, (m, i)s)$ of the simple sub variables $(m, i)s$ in the canonical order. Values of composite variables x are represented in $m.ct$ in the obvious way by sequences of simple values starting from the abstract base address $ba(x)$ of variable x.

With the help of visibility rules and bindings we easily extend the definition of va, ty, and ba from variables and sub variables to expressions e.

Computation. For space restrictions we cannot give the definitions of the (small-step) transition function δ_{C0} mapping $C0$ configurations c_{C0} to their successor configuration $c'_{C0} = \delta_{C0}(c_{C0})$. As an example we give a partial definition of the function call semantics.

Assume the program rest in configuration c_{C0} begins with a call of function f with parameters $e_1, \ldots, e_n$ assigning the function's result to variable v, formally $c_{C0}.pr =$