Reading List on Proof-Carrying-Code

Hans-Wolfgang Loidl

June 29, 2005

Reading List on Proof-Carrying-Code

[Abadi and Leino, 1997] Martin Abadi and Rustan Leino. A logic of object-oriented programs. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214, pages 682–696, Springer-Verlag, New York, N.Y., 1997.

Abstract: We develop a logic for reasoning about object-oriented programs. The logic is for a language with an imperative semantics and aliasing, and accounts for self-reference in objects. It is much like a type system for objects with subtyping, but our specifications go further than types in detailing pre- and postconditions. We intend the logic as an analogue of Hoare logic for objectoriented programs. Our main technical result is a soundness theorem that relates the logic to a standard operational semantics.

File: abadi98logic.ps.gz.

[Appel, 2001] Andrew W. Appel. Foundational Proof-Carrying Code. In *LICS'01 — Symposium on Logic in Computer Science*, June 2001 **URL:** http://www.cs.princeton.edu/ appel/papers/fpcc.pdf, **File:** Appel-fpcc.pdf.

[Aspinall *et al.*, 2004a] D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In *CASSIS'04 — Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, LNCS, Marseille, France, March 10–13, 2004. Springer-Verlag **File:** cassis2004.pdf.

[Aspinall *et al.*, 2004b] David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. A program logic for resource verification. In *Proceedings of 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 34–49, Heidelberg, September 2004. Springer-Verlag. **Location:** Park City, Utah **URL:** http://groups.inf.ed.ac.uk/mrg/publications/mrg/tphol-paper.pdf.

[Berghofer and Nipkow, 2000] Stefan Berghofer and Tobias Nipkow. Proof Terms for Simply Typed Higher Order Logic. In *TPHOL'02 — Theorem Proving in Higher Order Logics*, volume 1869 of *Lecture Notes in Computer Science*, pages 38–52, 2000.

Abstract: This paper presents proof terms for simply typed, intuitionistic higher order logic, a popular logical framework. Unification-based algorithms for the compression and reconstruction of proof terms are described and have been implemented in the theorem prover Isabelle. Experimental results confirm the effectiveness of the compression scheme.

URL: http://www4.informatik.tu-muenchen.de/ nipkow/pubs/tphols00.ps.gz, **File:** berghofer-phols00.ps.gz.

[Chander *et al.*, 2005] Ajay Chander, David Espinosa, Nayeem Islam, Peter Lee, and George Necula. Enforcing Resource Bounds via Static Verification of Dynamic Checks. In *ESOP'05 — European Symposium on Programming*, volume 3444 of *LNCS*, pages 311–, Edinburgh, UK, April 4–8, 2005.

Abstract: We classify existing approaches to resource-bounds checking as static or dynamic. Dynamic checking performs checks during program execution, while static checking performs them before execution. Dynamic checking is easy to implement but incurs runtime cost. Static checking avoids runtime overhead but typically involves difficult, often incomplete program analyses. In particular, static checking is hard in the presence of dynamic data and complex program structure. We propose a new resource management paradigm that offers the best of both worlds. We present language constructs that let the code producer optimize dynamic checks by placing them either before each resource use, or at the start of the program, or anywhere in between. We show how the code consumer can then statically verify that the optimized dynamic checks enforce his resource bounds policy. We present a practical language that is designed to admit decidable yet efficient verification and prove that our procedure is sound and optimal. We describe our experience verifying a Java implementation of tar for resource safety. Finally, we outline how our method can improve the checking of other dynamic properties.

File: Necula-ESOP05.pdf.

[Chang *et al.*, 2002] Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. Trustless Grid Computing in ConCert. In *GRID 2002: Third International Workshop on Grid Computing*, volume 2536 of *LNCS*, Baltimore, MD, November 2002.

Abstract: We believe that fundamental to the establishment of a grid computing framework where all (not just large organizations) are able to effectively tap into the resources available on the global network is the establishment of trust between grid application developers and resource donors. Resource donors must be able to trust that their security, safety, and privacy policies will be respected by programs that use their systems. In this paper, we present a novel solution based on the notion of certified code that upholds safety, security, and privacy policies by examining intrinsic properties of code. Certified code complements authentication and provides

a foundation for a safe, secure, and efficient framework that executes native code. We describe the implementation of such a framework known as the ConCert software.

URL: http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/concert/www/papers/grid2002/grid2002.pdf, **File:** TrustlessGrid.pdf.

[Colby *et al.*, 2000] Christopher Colby, Peter Lee, and George C. Necula. A Proof-Carrying Code architecture for Java. In *CAV'00 — International Conference on Computer Aided Verification*, ACM Press, Chicago, IL, July 2000.

Abstract: In earlier work, Necula and Lee developed proof-carrying code(PCC) [3, 5], which is a mechanism for ensuring the safe behavior of programs. In PCC, a program contains both the code and an encoding of an easy-to-check proof. The validity of the proof, which can be automatically determined by a simple proof-checking program, implies that the code, when executed, will behave safely according to a user-supplied formal definition of safe behavior. Later, Necula and Lee demonstrated the concept of a certifying compiler[6, 7]. Certifying compilers promise to make PCC more practical by compiling high-level source programs into optimized PCC binaries completely automatically, as opposed to depending on semi-automatic theorem-proving techniques. Taken together, PCC and certifying compilers provide a possible solution to the code safety problem, even in applications involving mobile code [4]. In this paper we describe a PCC architecture comprising two tools: (1) A thin PCC layer implemented in C that protects a host system from unsafe software. The host system can be anything from a desktop computer down to a smartcard. The administrator of the host system specifies a safety policy in a variant of the Edinburgh Logical Framework (LF) [1]. This layer loads PCC binaries, which are Intel x86 ob ject files that contain a section providing a binary encoding of a safety proof, and checks them against the safety policy before installing the software. (2) A software-development tool that produces x86 PCC binaries from Java classfiles. It is implemented in Objective Caml [2]. From a developer's perspective, this tool works just like any other compiler, with an interface similar to javac or gcc. Behind the scenes, the tool produces x86 machine code along with a proof of type safety according to the Java typing rules.

URL: http://raw.cs.berkeley.edu/Papers/cav00.ps, **File:** colby-cav00.ps.

[Crary and Weirich, 2000] K. Crary and S. Weirich. Resource Bound Certification. In *POPL'00 — Symposium on Principles of Programming Languages*, pages 184–198,, Boston, MA, January 2000.

Abstract: Various code certification systems allow the certification and static verification of important safety properties such as memory and control-flow safety. These systems are valuable tools for verifying that untrusted and potentially malicious code is safe before execution. However, one important safety property that is not usually included is that programs adhere to specific bounds on resource consumption, such as running time. We present a decidable type system capable of specifying and certifying bounds on resource consumption. Our system makes two advances over previous resource bound certification systems, both of which are necessary for a practical system: We allow the execution time of programs and their subroutines to vary, depending on their arguments, and we provide a fully automatic compiler generating certified executables from source-level programs. The principal

device in our approach is a strategy for simulating dependent types using sum and inductive kinds.

URL: http://www-2.cs.cmu.edu/ crary/papers/1999/res/res.ps.gz, **File:** Crary-resbound.ps.

[Dem, 2005] Touchstone online demo, June 2005 **URL:** http://raw.cs.berkeley.edu/Ginseng/Images/pccdemo.html.

[Hofmann and Jost, 2003] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symposium on Principles of Programming Languages*, volume 38, pages 185–197, New Orleans, LA, USA, January 2003. ACM Press.

Abstract: We show how to efficiently obtain linear a priori bounds on the heap space consumption of first-order functional programs. The analysis takes space reuse by explicit deallocation into account and also furnishes an upper bound on the heap usage in the presence of garbage collection. It covers a wide variety of examples including, for instance, the familiar sorting algorithms for lists, including quicksort. The analysis relies on a type system with resource annotations. Linear programming (LP) is used to automatically infer derivations in this enriched type system. We also show that integral solutions to the linear programs derived correspond to programs that can be evaluated without any operating system support for memory management. The particular integer linear programs arising in this way are shown to be feasibly solvable under mild assumptions.

File: hofmann-jost.ps.

[Hofmann, 1998] M. Hofmann. Semantik und Verifikation. Lecture Notes, 1998, TU Darmstadt.

[Hofmann, 2000] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

Abstract: We show how linear typing can be used to obtain functional programs which modify heap-allocated data structures in place. We present this both as a "design pattern" for writing C-code in a functional style and as a compilation process from linearly typed first-order functional programs into malloc-free Ccode. The main technical result is the correctness of this compilation. The crucial innov ation over previous linear typing schemes consists of the introduction of a resource type D which controls the number of constructor symbols such as consin recursive definitions and ensures linear space while restricting expressive power surprisingly little. While the space efficiency brought about by the new typing scheme and the compilation into C can also be realised by with state-of-the-art optimising compilers for functional languages such as Ocaml[16], the present method provides guaranteed bounds on heap space which will be of use for applications such as languages for embedded systems or automatic certification of resource bounds. We show that the functions expressible in the system are precisely those computable on a linearly space-bounded T uring machine with an unbounded stack. By a result of Cook this equals the complexity class 'exponential time'. A tail recursive fragment of the language captures the complexity class 'linear space'. We discuss various extensions, in

particular an extension with FIFO queues admitting constant time catenation and enqueuing, and an extension of the type system to fully-fledged intuitionistic linear logic.

URL: http://www.dcs.ed.ac.uk/home/mxh/nordic.ps.gz, **File:** diamond_paper.ps.gz.

[Huisman and Jacobs, 2000] M. Huisman and B. Jacobs. Java Program Verfication via a Hoare Logic with Abrupt Termination. In T. Maibaum, editor, *FASE'00 — Fundamental Approaches to Software Engineering*, volume 1783 of *LNCS*, pages 284–303. Springer-Verlag, 2000.

Abstract: This paper formalises a semantics for statements and expressions (in sequential imperative languages) which includes non-termination, normal termination and abrupt termination (e.g. because of an exception, break, return or continue). This extends the traditional semantics underlying e.g. Hoare logic, which only distinguishes termination and non-termination. An extension of Hoare logic is elaborated that includes means for reasoning about abrupt termination (and side-effects). It prominently involves rules for reasoning about while loops, which may contain exceptions, breaks, continues and returns. This extension applies in particular to Java. As an example, a standard pattern search algorithm in Java (involving a while loop with returns) is proven correct using the proof-tool PVS.

URL: http://www.cs.kun.nl/ bart/PAPERS/FASE00.ps.Z, **File:** FASE00.ps.

[Klein and Nipkow, 2004] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004 **File:** jinja.pdf.

[Kleymann, 1999] Thomas Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs.* PhD thesis, LFCS, 1999.

Abstract: Investigating soundness and completeness of verification calculi for imperative programming languages is a challenging task. Many incorrect results have been published in the past. We take advantage of the computer-aided proof tool LEGO to interactively establish soundness and completeness of both Hoare Logic and the operation decomposition rules of the Vienna Development Method (VDM) with respect to operational semantics. We deal with parameterless recursive procedures and local variables in the context of total correctness. As a case study, we use LEGO to verify the correctness of Quicksort in Hoare Logic. As our main contribution, we illuminate the rle of auxiliary variables in Hoare Logic. They are required to relate the value of program variables in the final state with the value of program variables in the initial state. In our formalisation, we reflect their purpose by interpreting assertions as relations on states and a domain of auxiliary variables. Furthermore, we propose a new structural rule for adjusting auxiliary variables when strengthening preconditions and weakening postconditions. This rule is stronger than all previously suggested structural rules, including rules of adaptation. With the new treatment, we are able to show that, contrary to common belief, Hoare Logic subsumes VDM in that every derivation in VDM can be naturally embedded in Hoare Logic. Moreover, we establish completeness results uniformly as corollaries of Most General Formula theorems which remove the need to reason about arbitrary assertions.

URL: http://www.lfcs.informatics.ed.ac.uk/reports/98/ECS-LFCS-98-392/ECS-LFCS-98-392.pdf, **File:** PhD-Kleymann.pdf.

[Leino, 1998a] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. *Nordic Journal of Computing*, 5(4):330–360, 1998.

Abstract: This paper formalizes a small object-oriented programming notation. The notation features imperative commands where objects can be shared (aliased) and is rich enough to allow subtypes and recursive object types. The syntax, type checking rules, axiomatic semantics, and operational semantics of the notation are given. A soundness theorem showing the consistency between the axiomatic and operational semantics is also given. A simple corollary of the soundness theorem demonstrates the soundness of the type system. Because of the way types, fields, and methods are declared, no extra effort is required to handle recursive object types.

File: leino98recursive.ps.gz.

[Leino, 1998b] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. Technical Note 1997-025a, Digital Systems Research Center, Palo Alto, CA, January 1998, Superseded by [Leino, 1998a], but with complete operational semantics and soundness proof.

Abstract: This paper formalizes a small object-oriented programming notation. The notation features imperative commands where objects can be shared (aliased), and is rich enough to allow subtypes and recursive object types. The syntax, type checking rules, axiomatic semantics, and operational semantics of the notation are given. A soundness theorem, showing the consistency between the axiomatic and operational semantics is stated and proved. A simple corollary of the soundness theorem demonstrates the soundness of the type system. Because of the way types, fields, and methods are declared, no extra effort is required to handle recursive object types.

URL: ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/technical-notes/SRC-1997-025a.ps.gz, **File:** leino97recursive.ps.gz.

[Leino, 2001] K. Rustan M. Leino. Applications of extended static checking. In Patrick Cousot, editor, *SAS'01 — International Static Analysis Symposium*, volume 2126 of *LNCS*, pages 185–193. Springer-Verlag, July 2001.

Abstract: Extended static checking is a powerful program analysis technique. It translates into a logical formula the hypothesis that a given program has some particular desirable properties. The logical formula, called a verification condition, is then checked with an automatic theorem prover. The extended static checking technique has been built into a couple of program checkers. This paper discusses other possible applications of the technique to the problem of producing quality software more quickly.

URL: http://www.research.compaq.com/SRC/personal/rustan/papers/krml106.ps, **File:** krml106.ps.

[Mehta and Nipkow, 2005] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. *Information and Computation*, 2005, To appear **File:** ic05.ps.gz.

[Müller and Poetzsch-Heffter, 1999] P. Müller and A. Poetzsch-Heffter. A Programming Logic for Sequential Java. In *ESOP'99 — European Symposium on Programming*, LNCS, pages 208–225. Springer, 1999.

Abstract: A Hoare-style programming logic for the sequential kernel of Java is presented. It handles recursive methods, class and interface types, subtyping, inheritance, dynamic and static binding, aliasing via object references, and encapsulation. The logic is proved sound w.r.t. an SOS semantics by embedding both into higher-order logic.

File: MuPo99.ps.

[Murphy VII *et al.*, 2004] Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A Symmetric Modal Lambda Calculus for Distributed Computing. Technical Report CMU-CS-04-105, Carnegie Mellon University, 2004.

Abstract: We present a foundational language for distributed programming, called Lambda 5, that addresses both mobility of code and locality of resources. In order to construct our system, we appeal to the powerful propositions-as-types interpretation of logic. Specifically, we take the possible worlds of the intuitionistic modal logic IS5 to be nodes on a network, and the connectives Box and Diamond to reflect mobility and locality, respectively. We formulate a novel system of natural deduction for IS5, decomposing the introduction and elimination rules for Box and Diamond, thereby allowing the corresponding programs to be more direct. We then give an operational semantics to our calculus that is type-safe, logically faithful, and computationally realistic.

File: symmetric.ps.

[Necula and Lee, 1996] George C. Necula and Peter Lee. Proof-Carrying Code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University Pittsburgh, PA, September 1996.

Abstract: This report describes Proof-Carrying Code, a software mechanism that allows a host system to determine with certainty that it is safe to execute a program supplied by an untrusted source. For this to be possible, the untrusted code supplier must provide with the code a safety proof that attests to the code's safety properties. The code consumer can easily and quickly validate the proof without using cryptography and without consulting any external agents. In order to gain preliminary experience with proof-carrying code, we have performed a series of case studies. In one case study, we write safe assembly-language network packet filters. These filters can be executed with no run-time overhead, beyond a one-time cost of 1 to 3 milliseconds for validating the attached proofs. The net result is that our packet filters are formally guaranteed to be safe and are faster than packet filters created using Berkeley Packet Filters, Software Fault Isolation, or safe languages such as Modula-3. In another case study we show how proof-carrying code can be used to develop safe assembly-language extensions of the a simplified version of the TIL run-time system for Standard ML.

URL: http://www-2.cs.cmu.edu/ petel/papers/pcc/pcc-tr.ps, **File:** Lee-pcc-tr.ps.

[Necula and Lee, 1998a] G.C. Necula and P. Lee. Efficient Representation and Validation of Proofs. In *LICS'98 — Symposium on Logic in Computer Science*, Indianapolis, IN, 1998.

Abstract: This paper presents a logical framework derived from the Edinburgh Logical Framework (LF) that can be used to obtain compact representations of proofs and efficient proof checkers. These are essential ingredients of any application that manipulates proofs as first-class objects, such as a Proof-Carrying Code system, in which proofs are used to allow the easy validation of properties of safety-critical or untrusted code. Our framework, which we call LFi, inherits from LF the capability to encode various logics in a natural way. In addition, the LFi framework allows proof representations without the high degree of redundancy that is characteristic of LF representations. The missing parts of LFi proof representations can be reconstructed during proof checking by an efficient reconstruction algorithm. We also describe an algorithm that can be used to strip the unnecessary parts of an LF representation of a proof. The experimental data that we gathered in the context of a Proof-Carrying Code system shows that the savings obtained from using LFi instead of LF can make the difference between practically useless proofs of several megabytes and manageable proofs of tens of kilobytes. This paper is an abbreviated version of a longer (70 pages) technical report. Read it if you want to see the detailed (15 pages) proofs of soundness of the efficient proof-checking algorithm that is used in PCC.

URL: http://raw.cs.berkeley.edu/Papers/lfi_lics98.ps, **File:** lfi_lics98.ps.

[Necula and Lee, 1998b] George C. Necula and Peter Lee. Safe, Untrusted Agents Using Proof-Carrying Code. In *Special Issue on Mobile Agent Security*, volume 1419 of *LNCS*. Springer-Verlag, 1998.

Abstract: This paper is intended to be both a comprehensive implementation guide for a Proof-Carrying Code system and a case study for using PCC in a mobile agent environment. Specifically, the paper describes the use of PCC for enforcing memory safety, access control and resource usage bounds for untrusted agents that access a database.

URL: http://raw.cs.berkeley.edu/Papers/pcc_lncs98.ps, **File:** pcc_lncs98.ps.

[Necula and Lee, 2000] George C. Necula and Peter Lee. Proof Generation in the Touchstone Theorem Prover. In *CADE'00 — International Conference on Automated Deduction*, Pittsburgh, PA, June 2000 **URL:** http://raw.cs.berkeley.edu/Papers/proofgen_cade00.ps, **File:** proofgen_cade00.ps.

[Necula and Schneck, 2002] George C. Necula and Robert R. Schneck. A Sound Framework for Untrustred Verification-Condition Generators. In *LICS03 — Symposium on Logic in Computer Science*, July 2002 **URL:** http://raw.cs.berkeley.edu/Papers/vcgen_lics03.pdf, **File:** vcgen_lics03.pdf.

[Necula and Schneck, 2003] George C. Necula and Robert R. Schneck. A Sound Framework for Untrustred Verification-Condition Generators. In *LICS03 — IEEE Symposium on Logic in Computer Science*, July 2003 **File:** vcgen_lics03.pdf.

[Necula *et al.*, 2004] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2004 **File:** ccured_toplas.pdf.

[Necula, 1997] George Necula. Proof-carrying code. In *POPL'97 — Symposium on Principles of Programming Languages*, Paris, France, January 1997.

Abstract: This paper describes proof-carrying code (PCC) a mechanism by which a host system can determine with certainty that it is safe to execute a program supplied (possibly in binary form) by an untrusted source. For this to be possible, the untrusted code producer must supply with the code a safety proof that attests to the code's adherence to a previously defined safety policy . The host can then easily and quickly v alidate the proof without using cryptography and without consulting any external agents. In order to gain preliminary experience with PCC, we have performed several case studies. We show in this paper how proof-carrying code might be used to develop safe assembly-language extensions of ML programs. In the context of this case study , we present and prove the adequacy of concrete representations for the safety policy, the safety proofs, and the proof validation. Finally, we briefly discuss how we use proof-carrying code to develop network packet filters that are faster than similar filters developed using other techniques and are formally guaranteed to be safe with respect to a given operating system safety policy.

URL: http://raw.cs.berkeley.edu/Papers/pcc_popl97.ps, **File:** pcc_popl97.ps.

[Necula, 1998] George Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998 **URL:** http://raw.cs.berkeley.edu/Thesis/thesis.pdf, **File:** PhD-Necula.pdf.

[Necula, 2001] George C. Necula. *Proof and System Reliability*, chapter Proof-Carrying Code: Design and Implementation. Springer-Verlag, 2001 **URL:** http://raw.cs.berkeley.edu/Papers/marktoberdorf.pdf, **File:** Necula-marktoberdorf.pdf.

[Nipkow,] Tobias Nipkow. Jinja: Towards a comprehensive formal semantics for a Java-like language. In H. Schwichtenberg and K. Spies, editors, *Proc. Marktobderdorf Summer School 2003*. IOS Press, To appear **File:** mod2003.pdf.

[Nipkow, 2002] Tobias Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic (CSL 2002)*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119, 2002 **File:** csl02.pdf.

[O'Hearn *et al.*, 2001] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local Reasoning about Programs that Alter Data Structures. In *CSL'01 — Annual Conference of the European Association for Computer Science Logic*, LNCS, pages 1–19, Paris, 2001. Springer-Verlag.

Abstract: We describe an extension of Hoare's logic for reasoning about programs that alter data structures. We consider a low-level storage model based on a heap with associated lookup, update, allocation and deallocation operations, and unrestricted address arithmetic. The assertion language is based on a possible worlds model of the logic of bunched implications, and includes spatial conjunction and implication connectives alongside those of classical logic. Heap operations are axiomatized using what we call the "small axioms", each of which mentions only those cells accessed by a particular command. Through these and a number of examples we show that the formalism supports local reasoning: A specification and proof can concentrate on only those cells in memory that a program accesses. This paper builds on earlier work by Burstall, Reynolds, Ishtiaq and O'Hearn on reasoning about data structures.

File: localreasoning.pdf.

[Oheimb and Nipkow, 2002] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.

Abstract: We define NanoJava, a kernel of Java tailored to the investigation of Hoare logics. We then introduce a Hoare logic for this language featuring an elegant new approach for expressing auxiliary variables: by universal quantification on the outer logical level. Furthermore, we give simple means of handling side-effecting expressions and dynamic binding within method calls. The logic is proved sound and (relatively) complete using Isabelle/HOL.

URL: http://isabelle.in.tum.de/verificard/Publications/NanoJava.ps.gz, **File:** NanoJava.ps.

[Pfenning, 2001] Frank Pfenning. *Logical Frameworks*, chapter Chapter 16, pages 977–1061. Elsevier Science and MIT Press, 2001 **URL:** http://www.cs.cmu.edu/ fp/papers/handbook00.pdf, **File:** LF-Pfenning.pdf.

[Reynolds, 1978] J. C. Reynolds. Syntactic control of interference. In *POPL'78 — Symp. on Princ. of Prog. Lang.*, 1978 **URL:** ftp://ftp.cs.cmu.edu/user/jcr/syncontrol.ps.gz.

[Reynolds, 2002] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS'02 — Symposium on Logic in Computer Science*, Copenhagen, Denmark, July 22–25, 2002.

Abstract: In joint work with Peter O'Hearn and others, based on early ideas of Burstall, we have developed an extension of Hoare logic that permits reasoning about low-level imperative programs that use shared mutable data structure. The simple imperative programming language is extended with commands (not expressions) for accessing and modifying shared structures, and for explicit allocation and deallocation of storage. Assertions are extended by introducing a "separating conjunction" that asserts that its subformulas hold for disjoint parts of the heap,

and a closely related "separating implication". Coupled with the inductive definition of predicates on abstract data structures, this extension permits the concise and flexible description of structures with controlled sharing. In this paper, we will survey the current development of this program logic, including extensions that permit unrestricted address arithmetic, dynamically allocated arrays, and recursive procedures. We will also discuss promising future directions.

URL: ftp://ftp.cs.cmu.edu/user/jcr/seplogic.ps.gz, **File:** seplogic.ps.gz.

[von Oheimb, 2001] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

Abstract: This article presents a Hoare-style calculus for a substantial subset of Java Card, which we call Java_light. In particular, the language includes side-effecting expressions, mutual recursion, dynamic method binding, full exception handling, and static class initialization. The Hoare logic of partial correctness is proved not only sound (w.r.t. our operational semantics of Java_light, described in detail elsewhere) but even complete. It is the first logic for an object-oriented language that is provably complete. The completeness proof uses a refinement of the Most General Formula approach. The proof of soundness gives new insights into the role of type safety. Further by-products of this work are a new general methodology for handling side-effecting expressions and their results, the discovery of the strongest possible rule of consequence, and a flexible Call rule for mutual recursion. We also give a small but non-trivial application example. All definitions and proofs have been done formally with the interactive theorem prover Isabelle/HOL. This guarantees not only rigorous definitions, but also gives maximal confidence in the results obtained.

URL: http://isabelle.in.tum.de/Bali/papers/CPE01.ps.gz, **File:** CPE01.ps.

[Wildmoser and Nipkow, 2004] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320, 2004 **File:** tphols04.pdf.

[Wildmoser and Nipkow, 2005] Martin Wildmoser and Tobias Nipkow. Asserting Bytecode Safety. In *ESOP'05 — European Symposium on Programming*, volume 3444 of *LNCS*, pages 326–, Edinburgh, UK, April 4–8, 2005.

Abstract: We instantiate an Isabelle/HOL framework for proof carrying code to Jinja bytecode, a downsized variant of Java bytecode featuring objects, inheritance, method calls and exceptions. Bytecode annotated in a first order expression language can be certified not to produce arithmetic overflows. For this purpose we use a generic verification condition generator, which we have proven correct and relatively complete.

File: Necula-ESOP05.pdf.

[Wildmoser *et al.*, 2004] Martin Wildmoser, Tobias Nipkow, Gerwin Klein, and Sebastian Nanz. Prototyping proof carrying code. In J.-J. Levy, E. Mayer, and J. Mitchell, editors, *Exploring New Frontiers of Theoretical Informatics*, pages 333–347. Kluwer, 2004 **File:** tcs04.pdf.

[WWW, 2005a] Concert web page, June 2005 **URL:** http://www-2.cs.cmu.edu/ concert/main.html.

[WWW, 2005b] Touchstone web page, June 2005 **URL:** http://raw.cs.berkeley.edu/touchstone.html.