Logical Methods in Computer Science
www.lmcs-online.org

PREFACE

This special issue of Logical Methods in Computer Science (LMCS) contains extended and revised versions of selected papers presented at the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09), 21-23 January 2009, Savannah, Georgia, USA.

POPL'09 received 160 submissions, 36 of which were selected for presentation at the Symposium and publication in the proceedings. The papers collected in this LMCS special issue were selected by the guest editors with the help of the POPL'09 PC. Authors were invited to submit full versions to the special issue, which underwent a new and thorough reviewing and revision process, in accordance with the usual standards of LMCS. The resulting seven papers, all of which have been significantly extended, reflect both the high standards and the breadth of topics of POPL.

In *Lazy Evaluation and Delimited Control*, Ronald Garcia, Andrew Lumsdaine and Amr Sabry investigate the operational aspects of the call-by-need lambda calculus of Ariola et al. The authors uncover an abstract machine corresponding to the equational calculus which, strikingly, captures the laziness and sharing of call-by-need by a disciplined use of delimited control operations, rather than via a heap storing mutable thunks. The theory of the machine is carefully developed; this is somewhat subtle, amongst other things because of the need to maintain hygiene in the presence of evaluation under binders. As well as proving that the machine correctly implements standard-order reduction of the original calculus, the authors justify the intuition that what the machine does corresponds to delimited control by presenting, and showing the correctness of, a direct translation of the call-by-need calculus into a more standard call-by-value language with delimited control operations.

David Monniaux's *Automatic Modular Abstractions for Template Numerical Constraints* presents a technique for automatically generating the best abstract transformer for a block of instructions relative to a numerical abstract domain specified by conjunctions of linear inequalities (generalizing popular abstract domains such as intervals and octagons). The method, which also extends to loops, is based on quantifier elimination. Both the theory and the pragmatics of a large number of variations (including computations over reals, booleans, integers and floating-point numbers) are considered. As well as describing exciting and applicable new research, the presentation is particularly perspicuous and thorough, providing a highly accessible overview of the field of numerical abstract interpretation.

Bunched logics, which freely combine additive and multiplicative connectives, have a reading as logics of resources and have received much attention in recent years, particularly as the foundation for separation logic. In *Classical BI: Its Semantics and Proof Theory*, James Brotherston and Cristiano Calcagno present a foundational investigation of a novel bunched logic, CBI, in which the multiplicative (linear) connectives are treated classically, rather than intuitionistically. Soundness and completeness are shown for models which

All articles have already been published in the regular issues of Logical Methods in Computer Science.

DOI:10.2168/LMCS-POPL:2009

extend those of Boolean BI (in which the additives are classical) with an involution, understood as giving a dual for each resource. The proof theory of CBI is presented using a Belnap-style display calculus. The authors also describe a range of intriguing examples and possible applications for the new logic, ranging from finance to program verification.

In *Positive Supercompilation for a Higher-Order Call-By-Value Language*, Peter Jonsson and Johan Nordlander present a powerful supercompilation algorithm for a higher-order call-by-value functional language. A key component of the algorithm is a transformation that uses the results of a separate strictness analysis to inline function bodies when no risk of nontermination is detected, regardless of whether or not the functions' arguments are values. Computations are reordered by delaying transformation of function arguments until after inlining — but only to a degree that is consistent with call-by-value semantics. The authors not only prove that their supercompilation algorithm is itself terminating, but also that it preserves the semantics of its input programs; in particular, a supercompiled program obtained by applying the algorithm to a given functional program does not accidentally introduce termination on any input on which the original program diverges. An implementation and experiments with the algorithm on well-known benchmarks are also described. This work constitutes the first formal treatment of termination introduction for a higher-order call-by-value language, and is thus a valuable first step toward supercompiling impure call-by-value languages. The algorithm presented here improves upon previous supercompilation algorithms vis-a-vis call-by-value languages, and also compares favorably with previous such algorithms for call-by-name languages.

In *Automated Verification of Practical Garbage Collectors*, Chris Hawblitzel and Erez Petrank present the specification, implementation, and mechanical verification of functional correctness for two garbage collectors that are sufficiently realistic to compete with those currently in use in real systems. The x86 assembly instructions and macros, as well as the preconditions, postconditions, invariants, and assertions comprising the collectors and their allocators are written by hand in a suitable variant of BoogiePL. The Boogie tool is then used to generate verification conditions from the resulting specification, and to pass these conditions to the Z3 solver, which attempts to prove their validity. While most verifications of garbage collectors are accomplished with interactive provers and significant human effort, the verification reported here makes extensive use of automation; human interaction is therefore required only for specification and annotation of collectors. In addition, whereas previous verification efforts for garbage collectors have been concerned more with idealized implementations than with realistic ones, the sophistication of the collectors considered by the authors provides convincing evidence that realistic garbage collectors are well within the scope of automatic verification. Finally, the performance benchmarks presented suggests that automatically verified collectors can compare reasonably against native collectors on off-the-shelf C# benchmarks. Overall, the work reported here represents a substantial advance in the area.

Concurrent threads operating over shared memory are ubiquitous, but there have been comparatively few accounts of their denotational semantics. In *A Model of Cooperative Threads*, Martín Abadi and Gordon Plotkin develop a fully abstract trace-based semantics of an imperative language with cooperative (non-preemptive) threads. Whilst elementary, the definition of this semantics requires considerable subtlety to achieve adequacy and full asbtraction. The second half of the paper then shows how the semantics of this language may be derived in a principled way using the algebraic approach to effects, by which computational monads (in this case, particular variations on resumptions) are generated, in a nicely modular way, from an (in)equational theory of effectful operations. The development in the latter section is rather less elementary, but provides a convincing demonstration of the applicability of the algebraic approach in a non-trivial, and practically important, setting.

Finally, in *Equality Saturation: A New Approach to Optimization*, Ross Tate, Michael Stepp, Zachary Tatlock and Sorin Lerner propose a radically unconventional way of structuring analysis-based program optimizations in compilers. Rather than sequentially composing individual optimizing transformations, each program analysis first just adds to a collection of facts about the equivalence of program fragments, using a novel dataflow-like intermediate representation that compactly encodes many different, equivalent, versions of the program simultaneously. A second phase then uses a global cost model to select a single optimized version from amongst all these alternatives. This approach can essentially avoid the phase-ordering problem and effectively exploit synergies between different optimizations. The authors describe their new intermediate representation, the E-PEG, and the transformations between E-PEGs and ASTs in considerable detail. The paper also reports some extremely encouraging results concerning the practicality and effectiveness of an implementation of an optimizer for Java bytecode programs that uses equality saturation.

We thank the authors for their excellent submissions. We also thank the members of the POPL 2009 Programme Committee and their subreviewers for their reviews of the conference versions of these submissions, and those who served as reviewers of the versions submitted to this special issue. Finally, we are grateful to LMCS Managing Editor and POPL 2009 PC Chair Benjamin Pierce, LMCS Executive Editor Jiří Adámek and the LMCS Assistant Editors for their invaluable assistance throughout the process of assembling the issue.

Nick Benton and Patricia Johann
Guest Editors