Technical Communications of the International Conference on Logic Programming, 2010 (Edinburgh), pp. 248–254
http://www.floc-conference.org/ICLP-home.html

PROGRAM ANALYSIS TO SUPPORT CONCURRENT PROGRAMMING IN DECLARATIVE LANGUAGES

ROMAIN DEMEYER

University of Namur - Faculty of Computer Science
Rue Grandgagnage 21, 5000 Namur (Belgium)
E-mail address: rde@info.fundp.ac.be

Abstract. In recent years, manufacturers of processors are focusing on parallel architectures in order to increase performance. This shift in hardware evolution is provoking a fundamental turn towards concurrency in software development. Unfortunately, developing concurrent programs which are correct and efficient is hard, as the underlying programming model is much more complex than it is for simple sequential programs. The goal of this research is to study and to develop program analysis to support and improve concurrent software development in declarative languages. The characteristics of these languages offer opportunities, as they are good candidates for building concurrent applications while their simple and uniform data representation, together with a small and formally defined semantics makes them well-adapted to automatic program analysis techniques. In our work, we focus primarily on developing static analysis techniques for detecting race conditions at the application level in Mercury and Prolog programs. A further step is to derive (semi-) automatically the location and the granularity of the critical sections using a data-centric approach.

1. Introduction and Problem Description

Since the mid-70s, the power of the microprocessor, which is the basic component of the computer responsible for instruction execution and data processing, has increased constantly. For decades, we have witnessed a dramatic and continuous growth of clock speed, which is one of the main factors determining the performance of processors [Olu05]. Recently, however, this growth appears to have stabilized. Indeed, the manufacturers encounter several physical problems, notably the impossibility to dissipate the heat and a too high power consumption [Sut05]. Instead of driving clock speeds and straight-line instruction throughput ever higher, processor manufacturers are, for these reasons, turning to *hyperthreading* and *multicore* architectures, i.e. processors with multiple identical units of calculation [Her06, Sut05, Lu08].

This hardware revolution is going to change fundamentally the way people write software. Indeed, to benefit from the power of the new processors, software must be able to exploit their innate parallelism, which is not the case for traditional software which is, in

1998 ACM Subject Classification: D.1.3 [**Programming Techniques**]: Concurrent Programming; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages–*Program Analysis*; D.1.6 [**Programming Techniques**]: Logic Programming.

Key words and phrases: Program Analysis – Concurrent Programming – Logic Languages – Abstract Interpretation .

Technical Communications of the 26th International Conference on Logic Programming, Edinburgh, July, 2010
Editors: Manuel Hermenegildo, Torsten Schaub
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.ICLP.2010.248

most cases, written following the sequential model of programming. In this new context, software must be designed following the *concurrency model* [Her06, Her08, Mat04, Mag99, BA90, Hug08] of programming: the application is made of a set of interacting *processes* that are executed, at least conceptually, in parallel and often in a shared memory space [Don08].

Unfortunately, developing concurrent programs that are correct and efficient is really hard, as potential bugs related to concurrent execution are difficult to detect and to isolate. Indeed, the underlying programming model is much more complex than it is for simple sequential programs [Lu08, BA90, Gro07, AZ08]. What makes concurrent programming hard in any language is that one has to deal with the interactions between processes and the nondeterministic interleaving of executions, especially if these processes handle shared memory. That is how undesirable phenomena, which are called *race conditions*, occur: two or more threads attempt to change a shared piece of data at (almost) the same time and the final value of the data depends simply in what order threads access it [Hug08]. These race conditions occur because of a bad *synchronization* between threads [Lu08, BA90].

To avoid errors, so-called *critical sections* have to be identified in the source code and *mutual exclusion* between execution of these sections must be guaranteed, using for example locks [BA90] or software transactional memories (STM) [Sha97, Lar06, Jon07, Har05, Har03, Mul06, Mik07, Kel05]. Whatever the way in which this mutual exclusion is ensured, a crucial point is to determine the location and the size of the critical sections. On the one hand, if they are too small or badly located, it can introduce race conditions at the application level. One the other hand, it is essential to keep the critical section as small as possible in order not to loose more performance than necessary and to avoid inter-blocking [Gro07]. Moreover, ensuring mutual exclusion is far from trivial. Locks are not composable [Har05] – i.e. correctly protected pieces of codes can't be simply reused to form larger correctly protected operations – and using them can lead to deadlocks, livelocks, priority inversion [BA90, Ho05, Eng03, Nai07, Bec08] or security breaks [Tip06, Che04, How09]. While STM avoid these issues, their implementation is complex and irreversible operations, like i/o operations, are traditionnaly prohibited inside the atomic blocks [Gro07, Men08, Har09, Dal09, Luc08, Boe09].

Obviously, programmers desperately need a higher-level programming model for concurrency than what languages offer today [Sut05]. Logic programming languages are known to be particularly well-adapted to parallelism [Tic91] but program analysis is needed as it can be used to detect race conditions and other bugs related to concurrency. It can also be used to (help to) determine the appropriate location and granularity of the critical sections.

2. Background and Overview of the Existing Literature

The interest of program analysis to support concurrent programming is increasingly prevalent with the actual multicore crisis. In the context of *explicit parallelism* – where the programmer decides where and how to integrate the parallelism in his program, classical analysis tools target to detect race conditions [McC06, Pra06]. Some of these tools are on the base of transformation programs methods, the goal of which is to combine conceptual advantages of STM with those of locks [McC06, Pra06, Hic06]. But these tools are only able to detect low-level race conditions – i.e. simple reading and writing of a memory space. These tools are not able to detect the errors that occur at the level of the logic of the

application. For example, bad utilisation of critical sections can lead to violate an invariant related to a data structure of the application.

Recently, the problem has caught the attention in the context of declarative languages and very recent works are targeting race conditions detection in these languages [Chr10, Cla09]. In the context of object-oriented languages, [Bec08] proposes to use typestate specifications [Del04] and linear logic [Gir87] to express invariant related to an object and the input and output conditions of its methods. The goal is to statically detect race conditions involving these objects at the application level on the basis of the specified behaviour. A related work [Har06] presents a dynamic analysis for STM in Haskell which ensures that an invariant will not be violated during an execution.

A still more ambitious but complex objective is to (semi-)automatically determine the location and the size of the critical sections. Recent work [Vaz06, McC06] has proposed a *data-centric approach* to synchronize threads which consists of a two-step procedure: first, the programmer associates synchronization constraints to the data structures that must be accessed atomically; second, these information are used to complete, through program transformation, the source code with the adequate locking mechanism where the synchronisation is necessary. One main advantage of this approach is the control of the granularity of the concurrent system.

Despite numerous works about concurrency, we are far from being able to detect all kind of errors related to concurrent programming [Vaz06, Lu08]. In most cases, analysis is able to detect synchronization mistakes related to only one variable. Moreover, it generally does not deliver pertinent informations about the way one can correct it [Lu08]. *Test case generation* to expose concurrency errors must also be considered, but has to cope with a high complexity issue: the number of possible interleaving to consider is exponential [Tay92, Yan97]. This imposes to explore subtle methods to produce pertinent tests in practice [Lu08, Qad04].

Although the primary goal of these program analysis is to assist the programmer in writing concurrent programs, they can also be useful to guide so-called *implicit parallelism* transformation that aim at the automatic parallelisation of programs [Cos08]. This field is particularly active in the context of logic languages [Bon08, Gup01, Cha08, Mou08, Cas08, Cas07]. In this kind of code analysis, the detection and the exploitation of the parallelism is made completely automatically, often at compilation time, without clear contribution of the programmer.

3. Goal of the Research

The goal of this research is to study and to develop program analysis to support and improve concurrent software development in declarative languages. In contrast with more classic imperative ones, declarative languages allow to describe the logic behind a solution instead of having to describe the step-by-step process of how this solution must be computed by the computer. Declarative languages are mostly pure – i.e. they do not allow programs to provoke side-effects [Hen96] – which is known to increase the productivity of the developers and the reliability of the programs, and makes these languages good candidates to use for building concurrent applications. Moreover, their simple and uniform data representation, together with a small and formally defined semantics, makes these languages well-adapted to automatic program analysis techniques. In our work, we focus primarily on Mercury.

Mercury [Som96] is a modern logic programming language, which is designed to develop modular and reliable large-size software applications.

4. Current Status of the Research

This research is still in its beginning. For the moment, we focus on the dissection of the state of the art and we familiarize with the very large field of concurrency by reading papers, books and by trying to make constructive contacts with other researcher working on related projects. We are trying to figure out in what directions it is the most valuable to guide the research. Since the latter is highly related to logic programming, the ICLP Doctoral Consortium would be an excellent opportunity, not only to acquire a profound complement to the state of the art, but also to get in touch with both experts and other PhD students working on related topics and to exchange point of views and opinions about my future work. It goes without doubt that the consortium would be a valuable experience from which we will be able to take full advantage in pursuing our project.

5. Open Issues and Expected Achievements

We plan to develop static analysis techniques for detecting race conditions at the application level in Mercury in first phase, Prolog in a following phase, languages that are particularly well-suited for concurrency [Bon08, Tan07, Wan08]. Such an analysis can be done based on the location of the critical sections and a abstract specification of the behaviour of the shared data. We study how a very expressive formalism, such as linear temporal logic [Pnu77], can be used for this behavioural specification.

Also in the context of declarative languages, a further step is to derive (semi-) automatically the location and the granularity of the critical section, possibly by extending a data-centric approach as suggested by recent work [Vaz06, McC06]. The particular type representation in declarative languages, like Mercury, is expected to fit well with such an analysis and to open new perspectives compared to traditional imperative languages.

Further elements of interest are the automatic generation of test cases targeted to detecting concurrency bugs and how our techniques can be used to advance research on *implicit* parallelism [Cos08] in declarative languages.

Acknowledgements

This PhD research is under the supervision of Professor Wim Vanhoof.

References

[AZ08] Abdallah Deeb I. Al Zain, Kevin Hammond, Jost Berthold, Phil Trinder, Greg Michaelson, and Mustafa Aswad. Low-pain, high-gain multicore programming in Haskell: coordinating irregular symbolic computations on multicore architectures. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pp. 25–36. ACM, New York, NY, USA, 2008. doi:http://doi.acm.org/10.1145/1481839.1481843.

[BA90] M. Ben-Ari. *Principles of concurrent and distributed programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Bec08] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In Gail E. Harris (ed.), *OOPSLA*, pp. 227–244. ACM, 2008.
URL `http://doi.acm.org/10.1145/1449764.1449783`

[Boe09] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. Tech. Rep. HPL-2009-45, Hewlett Packard Laboratories, 2009.
URL `http://www.hpl.hp.com/techreports/2009/HPL-2009-45.html;http://www.hpl.hp.com/techreports/2009/HPL-2009-45.pdf`

[Bon08] Paul Bone. Calculating likely parallelism within dependant conjunctions for logic programs. October, 2008.

[Cas07] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. Annotation algorithms for unrestricted independent and-parallelism in logic programs. In Andy King (ed.), *LOPSTR*, *Lecture Notes in Computer Science*, vol. 4915, pp. 138–153. Springer, 2007.
URL `http://dx.doi.org/10.1007/978-3-540-78769-3_10`

[Cas08] Amadeo Casas, Manuel Carro, and Manuel V. Hermenegildo. A high-level implementation of non-deterministic, unrestricted, independent and-parallelism. In Maria Garcia de la Banda and Enrico Pontelli (eds.), *ICLP*, *Lecture Notes in Computer Science*, vol. 5366, pp. 651–666. Springer, 2008.
URL `http://dx.doi.org/10.1007/978-3-540-89982-2`

[Cha08] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. Partial vectorisation of Haskell programs. In M. Hermenegildo (ed.), *Workshop on Declarative Aspects of Multicore Programming*. 2008.

[Che04] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security & Privacy*, 2(6):76–79, 2004.
URL `http://doi.ieeecomputersociety.org/10.1109/MSP.2004.111`

[Chr10] Maria Christakis and Konstantinos Sagonas. Static detection of race conditions in erlang. In *Practical Aspects of Declarative Languages : PADL 2010*, no. 5937 in Lecture Notes in Computer Science, pp. 119–133. Springer-Verlag, 2010.

[Cla09] Koen Claessen, Michał Pałka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding race conditions in erlang with quickcheck and pulse. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 2009.

[Cos08] Vitor Santos Costa. On supporting parallelism in a logic programming system. In Manuel Hermenegildo (ed.), *Workshop on Declarative Aspects of Multicore Programming*. 2008.

[Dal09] Luke Dalessandro and Mickael L. Scott. Strong isolation is a weak idea. 2009. doi:http://transact09.cs.washington.edu/33_paper.pdf.

[Del04] Robert Deline and Manuel Fahndrich. Typestates for objects. In *In Proc. 18th ECOOP*, pp. 465–490. Springer, 2004.

[Don08] M. R. C. Van Dongen. Thread programming, 2008.

[Eng03] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks, 2003.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gro07] Dan Grossman. The transactional memory / garbage collection analogy. *ACM SIGPLAN Notices*, 42, 2007.

[Gup01] Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[Har03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pp. 388 – 402. 2003.

[Har05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 48–60. ACM, New York, NY, USA, 2005. doi:http://doi.acm.org/10.1145/1065944.1065952.

[Har06] Tim Harris and Simon Peyton-Jones. Transactional memory with data variants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*. Ottawa, 2006.

[Har09] Tim Harris. Language constructs for transactional memory. *SIGPLAN Notices*, 44(1):1–1, 2009. doi:http://doi.acm.org/10.1145/1594834.1480883.

[Hen96] Fergus Henderson, Thomas Conway, Zoltan Somogyi, David Jeffery, Peter Schachte, Simon Taylor, and Chris Speirs. The Mercury language reference manual. Tech. rep., 1996.

[Her06] Maurice Herlihy. The art of multiprocessor programming. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pp. 1–2. ACM, New York, NY, USA, 2006. doi:http://doi.acm.org/10.1145/1146381.1146382.

[Her08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. URL `http://www.worldcat.org/isbn/0123705916`

[Hic06] Michael Hicks, Jeffrey S. Foster, and Polyvios Prattikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. 2006.

[Ho05] Alex Ho, Steven Smith, and Steven Hand. On deadlock, livelock, and forward progress. Tech. Rep. UCAM-CL-TR-633, University of Cambridge Computing Laboratory, 2005.

[How09] Michael Howard. Basic training: Improving software security by eliminating the CWE top 25 vulnerabilities. *IEEE Security & Privacy*, 7(3):68–71, 2009. doi:http://dx.doi.org/10.1109/MSP.2009.69.

[Hug08] Cameron Hughes and Tracey Hughes. *Professional Multicore Programming: Design and Implementation for C++ Developers*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[Jon07] Simon Peyton Jones. Beautiful concurrency. In Andy Oram and Greg Wilson (eds.), *Beautiful Code*, pp. 385–406. O'Reilly & Associates, Inc., Sebastopol, CA 95472, 2007. Ch. 24.

[Kel05] Richard Kelsey, Jonathan Rees, and Mike Sperber. The incomplete scheme 48 reference manuel release 1.3, April 2005.

[Lar06] James Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[Lu08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In Susan J. Eggers and James R. Larus (eds.), *ASPLOS*, pp. 329–339. ACM, 2008.
URL `http://doi.acm.org/10.1145/1346281.1346323`

[Luc08] Victor Luchangco. Against lock-based semantics for transactional memory. In Friedhelm Meyer auf der Heide and Nir Shavit (eds.), *SPAA*, pp. 98–100. ACM, 2008.
URL `http://dblp.uni-trier.de/db/conf/spaa/spaa2008.html#Luchangco08`

[Mag99] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Mat04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[McC06] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006.

[Men08] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Single global lock semantics in a weakly atomic stm. *SIGPLAN Notices*, 43(5):15–26, 2008. doi:http://doi.acm.org/10.1145/1402227.1402235.

[Mik07] Leon Mika. Software transactional memory in Mercury, October 2007.

[Mou08] Paulo Moura, Ricardo Rocha, and Sara C. Madeira. Thread-based competitive or-parallelism. In Maria Garcia de la Banda and Enrico Pontelli (eds.), *ICLP*, *Lecture Notes in Computer Science*, vol. 5366, pp. 713–717. Springer, 2008.
URL `http://dx.doi.org/10.1007/978-3-540-89982-2`

[Mul06] Ulrich Muller. Introducing the atomic keyword into c/c++ using assembler code instrumentation and software transactional memory, 2006.

[Nai07] Lee Naish. Resource-oriented deadlock analysis. In Verónica Dahl and Ilkka Niemelä (eds.), *ICLP*, *Lecture Notes in Computer Science*, vol. 4670, pp. 302–316. Springer, 2007.
URL `http://dx.doi.org/10.1007/978-3-540-74610-2_21`

[Olu05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005. doi:http://doi.acm.org/10.1145/1095408.1095418.

[Pnu77] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977.

[Pra06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.

[Qad04] Shaz Qadeer and Dinghao Wu. KISS: keep it simple and sequential. *ACM SIGPLAN Notices*, 39(6):14–24, 2004.

[Sha97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[Som96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language, 1996.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005.

[Tan07] Jrme Tannier. Parallel Mercury. Tech. rep., Facults Universitaires Notre-Dame de la Paix, 2007. Mmoire fin d'tudes.

[Tay92] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Trans. on Softw. Eng.*, 18(3):206, 1992.

[Tic91] Evan Tick. *Parallel logic programming*. MIT Press, Cambridge, MA, USA, 1991.

[Tip06] Harold F. Tipton and Micki Krause (eds.). *Information security management handbook*. Auerbach Publications, Boca Raton, FL, USA, 5th edn., 2006.
URL `http://www.loc.gov/catdir/enhancements/fy0659/2003061151-d.html`

[Vaz06] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *ACM SIGPLAN Notices*, 41(1):334–345, 2006.

[Wan08] Peter Wang. Parallel Mercury. October, 2008.

[Yan97] Cheer-Sun Yang and Lori L. Pollock. The challenges in automated testing of multithreaded programs. In *In Proceedings of the 14th International Conference on Testing Computer Software*, pp. 157–166. 1997.