TECHNISCHE
UNIVERSITÄT
WIEN
Vienna University of Technology

computer languages

Institut für Computersprachen
AB Programmiersprachen und Übersetzer

15. Kolloquium

Programmiersprachen und Grundlagen der Programmierung

— Ergänzungsband —

Maria Taferl

12.-14. Oktober 2009

Jens Knoop, Adrian Prantl (Hrsg.)

Jens Knoop
Adrian Prantl
Institut für Computersprachen

Technische Universität Wien
http://www.complang.tuwien.ac.at

Bericht 2009-X-2
Schriftenreihe des Instituts für Computersprachen
Technische Universität Wien

Vorwort

Das Kolloquium *Programmiersprachen und Grundlagen der Programmierung* (*KPS*) findet dieses Jahr zum 15. Mal statt. Es setzt eine Reihe von Arbeitstagungen fort, die ursprünglich von den Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel) ins Leben gerufen wurde.

Aus den ursprünglich drei Arbeitsgruppen sind in der Zwischenzeit weitere Forschungsgruppen in ganz Deutschland und darüberhinaus hervorgegangen. Seit einigen Jahren präsentiert sich die Veranstaltung als ein offenes Forum für interessierte deutschsprachige Wissenschafter. Die folgende Liste gibt einen Überblick über die bisherigen Tagungsorte und Veranstalter und zeigt die lange Tradition der KPS-Treffen:

1980	Tannenfelde im Aukrug	Universität Kiel
1982	Altenahr	RWTH Aachen
1985	Passau	Universität Passau
1987	Midlum auf Föhr	Universität Kiel
1989	Hirschegg	Universität Augsburg
1991	Rothenberge bei Steinfurth	Universität Münster
1993	Garmisch-Partenkirchen	Universität der Bundeswehr München
1995	Alt-Reichenau	Universität Passau
1997	Avendorf auf Fehmarn	Universität Kiel
1999	Kirchhundem-Heinsberg	FernUniversität in Hagen
2001	Rurberg in der Eifel	RWTH Aachen
2004	Freiburg-Munzingen	Universität Freiburg
2005	Fischbachau	LMU München
2007	Timmendorfer Strand	Universität Lübeck

Das 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2009) wird vom Institut für Computersprachen der Technischen Universität Wien organisiert. Wir freuen uns, zu diesem Jubiläumstreffen mehr als 50 Teilnehmer von 28 Universitäten aus Österreich, der Schweiz, Deutschland, Holland, England und Kanada vom 12. bis 14. Oktober 2009 im Wallfahrtsort Maria Taferl am Österreichischen Jakobsweg begrüßen zu können. Besonders freuen wir uns, unter den Teilnehmern Prof. Dr. Dr.h.c. Hans Langmaack als einen der Gründerväter dieser Tagungsreihe begrüßen zu können, sowie Prof. Dr. Rudolf Berghammer, Prof. Dr. Peter Thiemann, Prof. Dr. Clemens Grelck und Dr. Annette Stümpel als Organisatoren früherer KPS-Treffen. Ganz besonders freuen wir uns, dass nahezu alle Teilnehmer am diesjährigen KPS-Treffen in einem Vortrag über ihre Forschungsarbeit berichten werden. Um dies im Rahmen der zur Verfügung stehenden Zeit zu ermöglichen, verzichten wir in diesem Jahr auf einen eingeladenen Hauptvortrag.

Der Tagungsband ist als Bericht 2009-X-1 des Instituts für Computersprachen der TU Wien erschienen. Er enthält 46 Beiträge, zum Teil in Form von Kurzzusammenfassungen, von 62 Autoren. Weitere spät eingegangene Beiträge sind in einem Ergänzungsband gesammelt, der als Bericht 2009-X-2 des Instituts für Computersprachen der TU Wien erscheint. Alle Beiträge aus Haupt- und Ergänzungsband zeigen die Breite der wissenschaftlichen Forschung im Bereich Programmiersprachen und Grundlagen der Programmierung im deutschsprachigen Raum. Eine CD-ROM fasst zusätzlich alle Beiträge der beiden Tagungsbände zusammen. Unser Dank gilt allen Autoren für ihren Einsatz und die gute Zusammenarbeit, die diese beiden Bände ermöglicht haben. Unser besonderer Dank gilt darüberhinaus Frau Ewa Vesely und Herrn Leonid Narinsky, die bei der Vorbereitung und Organisation dieses Treffens von der Erstellung der Webseite über die Planung der Exkursion bis zum schnellen Auftun zusätzlicher Unterbringungsmöglichkeiten nach dem Hinauslaufen aus der Kapazität des Tagungshauses Hervorragendes geleistet haben. Unseren besonderen Dank drücken

wir dabei auch allen Mitarbeitern des Hauses Hotel Schachner für die gute Zusammenarbeit bei der Planung dieses Treffens aus.

Wir wünschen allen Teilnehmern interessante und spannende Vorträge, fruchtbare Diskussionen und Anregungen für die Forschungsarbeit, das Kennenlernen neuer Kollegen und das Wiedersehen guter Bekannter, das Anbahnen und Knüpfen neuer Kooperationen und die Verbreiterung und Vertiefung bestehender, eine erlebnis- und abwechslungsreiche Exkursion zu dem zum UNESCO-Weltkulturerbe gehörenden Benediktinerklosters Stift Melk und einen angenehmen und anregenden Aufenthalt in Maria Taferl am Eingang zur Wachau.

Willkommen zur KPS 2009!

Wien, im Oktober 2009

Jens Knoop
Adrian Prantl

Vorwort zum Ergänzungsband

Der vorliegende Ergänzungsband zum 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2009) ist als Bericht 2009-X-2 des Instituts für Computersprachen der TU Wien erschienen. Er enthält 6 Beiträge, einen davon als Kurzzusammenfassung, die kurz nach Drucklegung für den Haupttagungsband, der als Bericht 2009-X-1 des Instituts für Computersprachen der TU Wien erschienen ist, eingegangen sind und auf diese Weise den Teilnehmern und allen anderen am Kolloquium Interessierten zur Verfügung gestellt werden können. Unser Dank gilt den Autoren für diese Beiträge. Unserer besonderer Dank gilt darüberhinaus den Mitarbeitern der Fa. `digitaldruck.at` in Leobersdorf für die ausgezeichnete und sehr kurzfristige Fertigstellung dieses Bandes.

Wien, im Oktober 2009

Jens Knoop
Adrian Prantl

Teilnehmer

Sebastian Altmeyer	Universität des Saarlandes, Saarbrücken
Wolfram Amme	Friedrich-Schiller-Universität Jena
Enes Bajrović	Universität Wien
Gergö Barany	Technische Universität Wien
Rudolf Berghammer	Christian-Albrechts-Universität zu Kiel
Dirk Beyer	Simon Fraser University, Surrey, B.C.
Walter Binder	Università della Svizzera italiana, Lugano
Florian Brandner	Technische Universität Wien
Stefan Brunthaler	Technische Universität Wien
Jens Dörre	Universität Passau
M. Anton Ertl	Technische Universität Wien
Christoph Feller	Technische Universität Kaiserslautern
Sebastian Fischer	Christian-Albrechts-Universität zu Kiel
Gerhard Goos	Universität Karlsruhe (TH)
Clemens Grelck	Universiteit van Amsterdam und University of Hertfordshire
Jürg Gutknecht	ETH Zürich
Sebastian Hack	Universität des Saarlandes, Saarbrücken
Michael Hanus	Christian-Albrechts-Universität zu Kiel
Thomas Heinze	Friedrich-Schiller-Universität Jena
Christoph Höger	Technische Universität Berlin
Christina Jansen	RWTH Aachen
Tudor Jebelean	RISC, Johannes Kepler Universität Linz
Raimund Kirner	Technische Universität Wien
Jens Knoop	Technische Universität Wien
Peter Lammich	Westfälische Wilhelms-Universität Münster
Oliver Lampl	Alpen-Adria-Universität Klagenfurt
Hans Langmaack	Christian-Albrechts-Universität zu Kiel
Florian Lorenzen	Technische Universität Berlin
Tim A. Majchrzak	Westfälische Wilhelms-Universität Münster
Volker Mattick	Technische Universität Dortmund
Eduard Mehofer	Universität Wien
Thomas Meyer	Universität Basel
Patrick Michel	Technische Universität Kaiserslautern
Leonid Narinsky	Technische Universität Wien
Ulrich Neumerkel	Technische Universität Wien
Nikolaj Popov	RISC, Johannes Kepler Universität Linz
Adrian Prantl	Technische Universität Wien
Franz Puntigam	Technische Universität Wien
Fabian Reck	Christian-Albrechts-Universität zu Kiel
Dirk Richter	Martin-Luther-Universität Halle-Wittenberg

Wolfgang Scholz	Universität Passau
Sven-Bodo Scholz	University of Hertfordshire
Markus Schordan	Fachhochschule Technikum Wien
Dietmar Schreiner	Technische Universität Wien
Martin Schwarz	Technische Universität München
Lukas Stadler	Johannes Kepler Universität Linz
Annette Stümpel	Universität zu Lübeck
Michael Tautschnig	Technische Universität Darmstadt
Peter Thiemann	Universität Freiburg
Baltasar Trancón y Widemann	Universität Bayreuth
Christian Tschudin	Universität Basel
Helmut Veith	Technische Universität Darmstadt
Alexander Wenner	Westfälische Wilhelms-Universität Münster
Thomas Würthinger	Johannes Kepler Universität Linz
Wolf Zimmermann	Martin-Luther-Universität Halle-Wittenberg
Florian Zuleger	Technische Universität Darmstadt

Inhaltsverzeichnis

CPAchecker: A Tool for Configurable Software Verification *

Dirk Beyer and M. Erkan Keremoglu

Simon Fraser University, B.C., Canada

Abstract. Configurable software verification is a recent concept for expressing different program analysis and model checking approaches in one single formalism. This paper presents CPAchecker, a tool and framework that aims at easy integration of new verification components. Every abstract domain, together with the corresponding operations, is required to implement the interface of configurable program analysis (CPA). The main algorithm is configurable to perform a reachability analysis on arbitrary combinations of existing CPAs. The major design goal during the development was to provide a framework for developers that is flexible and easy to extend. We hope that researchers find it convenient and productive to implement new verification ideas and algorithms using this platform and that it advances the field by making it easier to perform practical experiments. The tool is implemented in Java and runs as command-line tool or as Eclipse plug-in. We evaluate the efficiency of our tool on benchmarks from the software model checker Blast. The first released version of CPAchecker implements CPAs for predicate abstraction, octagon, and explicit-value domains. Binaries and the source code of CPAchecker are publicly available as free software.

1 Overview

The field of software verification is a fast growing area, and researchers contribute new ideas and approaches with enormous pace. The more new approaches are discovered, the more difficult it is to understand the essential insight or the fundamental difference that makes a new approach good and better. Experimental evaluation is often a deciding factor for whether or not a new approach is considered an advancement of the field. But it requires a considerable engineering effort to actually build the software infrastructure for evaluating verification algorithms. Adapting a suitable parser frontend and transforming the abstract syntax tree into a format that is convenient for verification algorithms is one example. The interaction with a theorem prover is yet another issue that needs to be considered. There are successful approaches in program analysis as well as in model checking, but these techniques are rarely combined; the reason being that it is indeed extremely difficult to combine them. Most published approaches are not even comparable, because the choice of the parser frontend, the choice

* This research was supported by the Canadian NSERC grant RGPIN 341819-07.

of the theorem prover, and the choice of the pointer-alias analysis algorithm in the corresponding tool implementation, considerably influence the performance and precision of the new verification algorithm. When evaluating a performance comparison of two approaches, it is often difficult to identify what the new approach contributes and what is due to the different environment. In practice, it was so far extremely difficult to perform an experimental performance evaluation of one component while keeping all other components constant.

Configurable program analysis (CPA) provides a conceptual basis for expressing different approaches in the same formal setting. The CPA formalism provides an interface for the definition of program analyses, which includes the abstract domain, the post operator, the merge operator, and the stop operator [5]. Consequently, the corresponding tool implementation CPAchecker provides an implementation framework that allows the seamless integration of program analyses that are expressed in the CPA framework. The comparison of different approaches in the same experimental setting becomes easy and the experimental results will be more meaningful (valid). The tool can be seen as a set of components that are loosely dependent on each other and that are easy to substitute.

In many respects, CPAchecker is similar to Blast [4]. For example, we implemented a predicate abstraction and an explicit-value analysis [6]. However, Blast has several limitations that we need to eliminate, most prominently, that the architecture and the design are not flexible enough to implement a pure CPA-based analysis. As in the Blast project already, many ideas were taken from Slam [2].

The source code, executables, and all benchmark programs for CPAchecker are available online at `http://www.sosy-lab.org/~dbeyer/CPAchecker`. The tool is free software, released under the Apache 2.0 license. CPAchecker is an open-source implementation of the framework of configurable program analysis (CPA). We hope that other researchers can integrate new techniques for software verification into CPAchecker and that software-verification technology becomes more accessible for practitioners using this platform.

We are currently working on integrating the bounded model checker CBMC as theorem prover, for a more precise counterexample analysis. Furthermore, we are working on a formulation of bounded model checking as CPA and look forward to integrating partial bounded model checking into CPAchecker. We have also evaluated different choices for the amount of control-flow that is considered in one single abstract-successor computation [3]. Also, we would like to integrate test-case generation and a specification language.

2 Architecture and Implementation

Figure 1 shows an overview of the CPAchecker architecture. The central data structure is a set of control-flow automata (CFA) (similar to control-flow graphs [1]), which consist of control-flow locations and control-flow edges. A location represents a program-counter value, and an edge represents a program

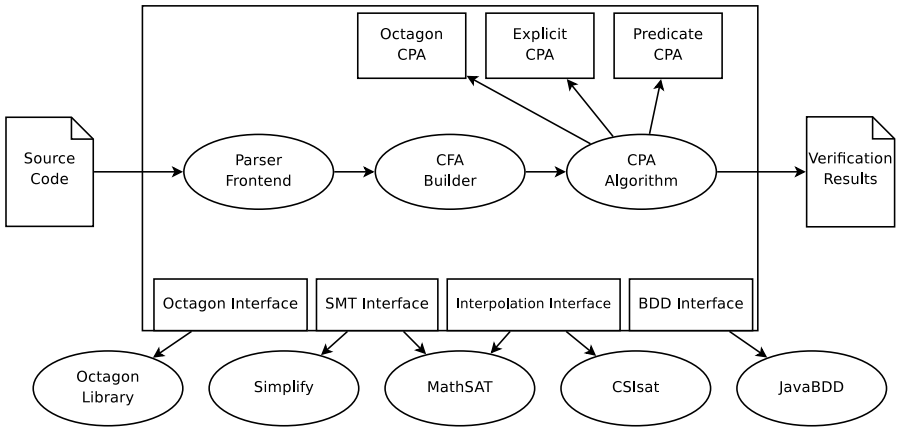


Fig. 1. CPAchecker — Architecture overview

operation, which is either an assume operation, an assignment block, a function call, or a function return (we do not consider more complex operations due to a well-known reduction called C intermediate language [8]). Before a program analysis starts, the input program is transformed into a syntax tree, and further into CFAs. The current version of CPAchecker uses the parser from the CDT [1], a fully functional C and C++ IDE plug-in for the Eclipse platform. Our framework provides interfaces to SMT solvers and interpolation procedures, such that the CPA operators can be written in a concise and convenient way. Currently we use Simplify[2] and MathSAT[3] as SMT solvers, and CSIsat[4] and MathSAT as interpolation procedures. We use JavaBDD[5] as BDD package and provide an interface to an Octagon[6] representation as well.

The central algorithm is the program-analysis algorithm that performs the reachability analysis [5]. (CPAchecker actually implements CPA+, i.e., CPA with precision adjustment, but we skip this detail for better presentation.) The analysis algorithm operates on an object of the abstract data type CPA, i.e., the algorithm applies operations from the CPA interface without knowing which concrete CPA it is analyzing. For most configurations, the concrete CPA will be a composite CPA [5], which implements the combination of several different CPAs.

In order to extend CPAchecker by integrating an additional CPA for a new abstract domain, only two steps are necessary. First, an entry in the global properties file is necessary in order to announce the new CPA for composition. Second, the interface for CPA needs to be implemented, and implementations of

[1] Available at `http://www.eclipse.org/cdt`
[2] Available at `http://secure.ucd.ie/products/opensource/Simplify`
[3] Available at `http://mathsat4.disi.unitn.it`
[4] Available at `http://www.cs.sfu.ca/~dbeyer/CSIsat`
[5] Available at `http://javabdd.sourceforge.net`
[6] Available at `http://www.di.ens.fr/~mine/oct`

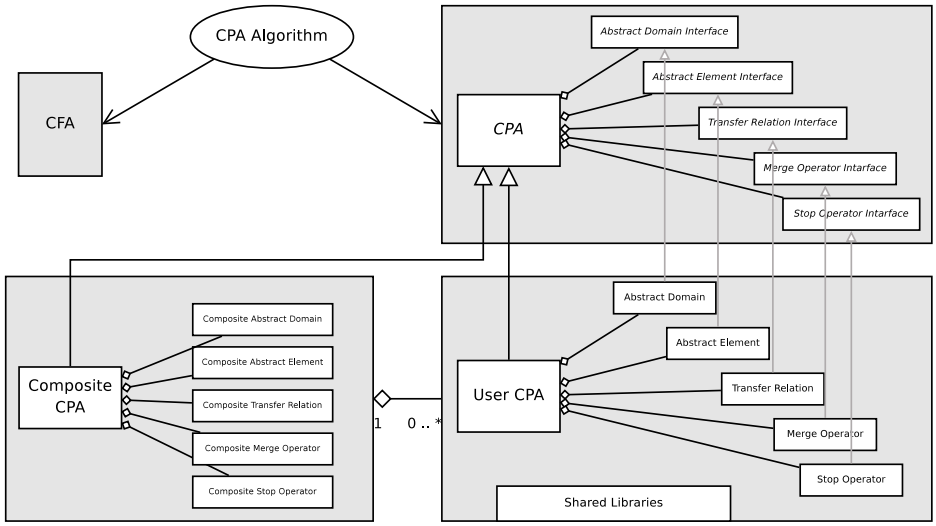


Fig. 2. CPAchecker — Design for extension

all CPA operation interfaces need to be provided. Figure 2 shows the interaction: The CPA algorithm (shown at the top in the figure) takes as input a set of control-flow automata (CFA) representing the program, and a CPA, which is in most cases a *Composite CPA*. The interfaces correspond one-to-one to the formal framework [5].

The elements in the gray box (top right) in Fig. 2 represent the abstract interfaces of the CPA and the CPA operations. The two gray boxes at the bottom of the figure show two implementations of the CPA interfaces, one is a *Composite CPA* that can combine several other CPAs, and the other is a *User CPA*. For example, suppose we want to implement a CPA for shape analysis. We would provide an implementation for *CPA*, possibly called *ShapeCPA*, and implementations for the operation interfaces on the right. If we want to experiment with several different merge operators, we would provide several different implementations of *Merge Operator Interface* that can be freely configured for use in various experiments.

3 Experiments

We report experiments in order to demonstrate that the tool implementation performs reasonably well on well-known benchmark examples. We pick a configuration for program analysis that was previously used [6], namely, the combination of an explicit-value analysis and a predicate-abstraction. Explicit-value analysis, also known as constant propagation, keeps track of values of integer variables. The predicate abstraction is based on Cartesian abstraction and lazy abstraction [7]. We run the analysis on various verification problems for simpli-

Table 1. Performance results; runtime given in seconds of processor time; the numbers in the column headings are the threshold values

Program	0	2	3	5	∞
cdaudio_simpl1	>1200.00	525.90	74.65	8.43	2.96
cdaudio_simpl1_BUG	167.67	88.45	17.09	3.28	0.62
diskperf_simpl1	>1200.00	>1200.00	36.95	21.19	280.10
floppy_simpl3	110.38	104.02	21.94	11.91	0.88
floppy_simpl3_BUG	42.33	37.55	7.98	2.37	0.35
floppy_simpl4	199.22	173.92	30.17	11.22	1.43
floppy_simpl4_BUG	42.95	36.15	8.03	2.16	0.36
kbfiltr_simpl1	13.77	4.59	3.50	1.02	0.42
kbfiltr_simpl2	30.89	9.98	5.48	1.83	0.89
kbfiltr_simpl2_BUG	16.17	5.76	1.24	0.73	0.32

Table 2. Statistical data observed during the experiments; a dash indicates that the experiment was aborted after 20 min; 'Preds' indicates the number of predicates used in the verification run, and 'Refines' indicates the number of refinement steps

Program	0		2		3		5	
	Preds	Refines	Preds	Refines	Preds	Refines	Preds	Refines
cdaudio_simpl1	-	-	81	332	12	76	2	11
cdaudio_simpl1_BUG	112	242	56	140	12	38	2	10
diskperf_simpl1	-	-	-	-	20	61	4	34
floppy_simpl3	81	219	51	167	20	51	4	21
floppy_simpl3_BUG	47	125	38	93	13	28	6	5
floppy_simpl4	96	307	54	219	20	58	4	19
floppy_simpl4_BUG	47	125	38	93	13	28	6	5
kbfiltr_simpl1	30	70	7	22	5	11	1	2
kbfiltr_simpl2	48	133	7	40	5	11	1	2
kbfiltr_simpl2_BUG	44	89	16	34	1	4	0	1

fied versions of Windows device drivers. The verification property is always a safety property (reachability of a certain error location under certain variable values) and is thus contained in the source code. The same program name ending with a different number indicates that the same program is present with a different simplification applied to the source code. If the program name ends with "BUG" then a defect was artificially introduced into the program.

The overall performance results obtained in our initial development phase of CPAchecker are satisfactory, although optimization was not the main design goal — rather we focussed on a portable and flexible environment to be used for many different analysis purposes. All experiments were performed on a GNU/Linux (Ubuntu 8.10) x86_32 machine with an Intel Core 2 Duo processor and 2 GB RAM. We limited the memory for the Java virtual machine to 1.8 GB and set the time limit for termination to 1200 s.

Table 1 shows the performance results for different configurations. The first column of the table lists the names of the programs. The next five columns report the runtimes for the analysis configuration where predicate abstraction and explicit-value analysis are used together. The threshold (the number in the

column heading) indicates how many different explicit values where tracked for each variable (cf. [6] for the details). After reaching this threshold the value of the variable is set to $\top$, i.e., nothing can be said about the value of the variable in the explicit analysis. This might lead to an infeasible path and the predicate-abstraction domain discovers predicates in order to track the missing variables and to eliminate the infeasible program path. We experimented with five different threshold values, where 0 represents the extreme case of pure predicate abstraction-based analysis, and ∞ represents the extreme case of pure explicit-value analysis. Table 1 indicates that the best performance in total for this set of programs is achieved with a threshold of 5, which represents a good tradeoff between the expensive but abstract predicate abstraction and the simple but exploding explicit-value analysis. It is interesting to observe that pure predicate abstraction is not tractable for some of the experiments (time out reached).

Table 2 shows the number of predicates and the number of refinement iterations needed to obtain the verification result. Surprisingly, many facts can be tracked by explicit values, and thus the number of predicates in the abstract-successor computations is drastically reduced. Also, the number of refinements that are necessary to discover predicates is significantly reduced (note that many different refinements might discover the same predicate for different locations).

Acknowledgments. We thank Tom Henzinger, Ranjit Jhala, and Rupak Majumdar for the fruitful collaboration in the Blast project. Blast served as example for CPAchecker in several aspects. We also thank Alberto Griggio, Andreas Holzer, and Michael Tautschnig for their valuable comments and for their code contributions to CPAchecker. We thank Alberto especially for his contribution to the predicate-abstraction analysis.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. T. Ball and S. K. Rajamani. The Slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
3. D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, 2009.
4. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
5. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.
6. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*. IEEE, 2008.
7. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
8. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC*, LNCS 2304, pages 213–228. Springer, 2002.

Inline Caching meets Quickening

Stefan Brunthaler

Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
brunthaler@complang.tuwien.ac.at

Abstract. Inline caches effectively eliminate the overhead implied by dynamic typing. Unfortunately, inline caching is mostly used in code generated by just-in-time compilers. We present efficient implementation techniques for using inline caches without dynamic translation, thus enabling future interpreter implementers to use this important optimization technique—we report speedups of up to 1.71—without the additional implementation and maintenance costs incurred by using a just-in-time compiler.

1 Motivation

Many of the currently popular programming language interpreters execute without dynamic code generation. The reason for this lies in their origins: Many of these languages were implemented by single developers, who maintained their—often extensive—standard libraries, too. Therefore, it was usually not an issue for them to substantially increase the a) the complexity and b) the maintenance efforts of their implementations by adding just-in-time compilers. Perl, Python, and Ruby are among the most popular of these programming languages that live without a dynamic compilation subsystem, but, nevertheless, seem to be major drivers behind many of advances in the internet's evolution.

In 2001, Ertl and Gregg found that there are certain optimization techniques for interpreters, e.g., threaded code [1,5], that cause them to perform at least an order of magnitude better than others [6]. Actually, their examination of Perl finds that its interpreter is not particularly efficient—which led us to subsequently analyze a similar interpreter [2]. Our analysis of the Python 3.0 interpreter shows that because of the nature of its interpreter, the application of exactly those techniques that cause other interpreters to perform significantly better, results in a comparatively lower speedup. Vitale and Abdelrahman report cases where the application of threaded code in the Tcl interpreter actually results in a slowdown [16].

This is due to the differing abstraction levels of the respective interpreters: While the Java virtual machine [14] reuses much of the native machine for operation implementation—i.e., it is a low abstraction-level virtual machine—, the interpreters of Perl, Python, and Ruby have a much more complex operation

implementation, which requires often significantly more native machine instructions; a characteristic of high abstraction-level interpreters. In consequence, optimization techniques that focus on minimizing the overhead in dispatching virtual machine instructions have a varying optimization potential with regard to the abstraction level of the underlying virtual machine. In low abstraction-level virtual machines the overhead in instruction dispatch is big, therefore using threaded code is particularly effective, resulting in reported speedups of a factor of 2.02 [6]. On the other hand, however, the same techniques achieve a much lower speedup in higher abstraction-level interpreters: the people implementing the Python interpreter report varying average speedups of about 20% in benchmarks, and significantly less (about 7%-8%) when running the Django[1] template benchmarks—a real world application.

In consequence, our examination of the operation implementation in the Python 3.x interpreter finds that there is substantial overhead caused by its dynamic typing—a finding that was true for Smalltalk systems 25 years ago. In 1984, however, Deutsch and Schiffman [4] published their seminal work on the "Efficient Implementation of the Smalltalk-80 System". Its major contributions were dynamic translation and inline caching. Subsequent research efforts on dynamic translation resulted in nowadays high performance just-in-time compilers, such as the Java Virtual Machine. Via polymorphic inline caches and type feedback [11], inline caching became an important optimization technique by itself. Unfortunately, inline caches are most often used together with dynamic translation. This paper presents our results on using *efficient* inline caching without dynamic translation in the Python 3.1 interpreter.

Our contributions are:

- We present a simple schema for efficiently using inline caching without dynamic translation. We describe a different instruction encoding that is required by our schema. (Section 2).
- We subsequently introduce a more efficient inline caching technique using instruction set extension (Section 3) with quickening (Section 3.1).
- We provide detailed performance figures on how our schemes compare with respect to the standard Python 3.x distribution on modern processors (Section 4). Our advanced technique achieves a speedup of up to 1.71. Using a combination of inline caching and threaded code results in a speedup of up to 1.92.

2 Basic Inline Caching without Dynamic Translation

In 1984, Deutsch and Schiffman describe their original version of inline caching [4]. During their optimization efforts on the Smalltalk-80 system, they observe a "*dynamic locality of type usage*", i.e., for any bytecode instruction within a given function or method, a call to the system default lookup routine is very likely to evaluate to the same address as it did during its previous invocation. Using

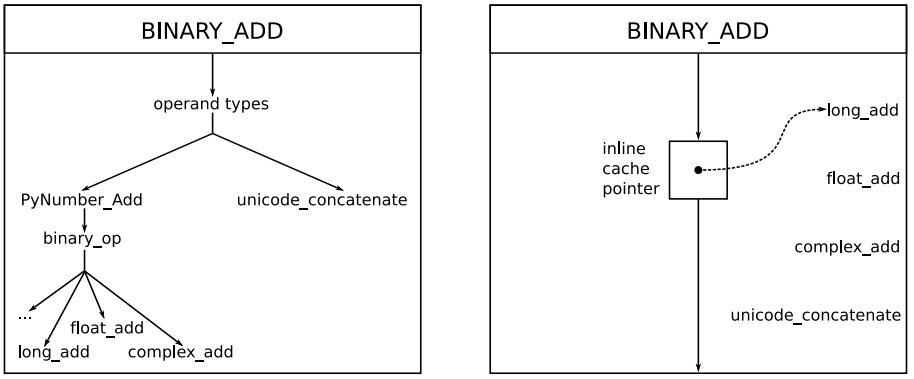
[1] Django is a popular Python Web application development framework.

this observation, they use their dynamic translation scheme to rewrite native machine call instructions from calling the system default lookup routine to directly calling its resulting target address. Like any caching technique, there is a strategy for detecting that the cache is invalid and what is to be done about that. For detecting an invalid inline cache entry, the interpreter ensures that the call circumventing the system default lookup routine depends upon the class operand—its *receiver*—having the same address as it did when the cache element was initialized. If that condition does not hold, the interpreter calls the system default lookup routine instead—which in turn takes care of properly updating the corresponding inline cache element.

With the notable exception of the native instruction rewriting, the previous paragraph does not indicate any prerequisites towards a dynamic translation schema. In fact, several method lookup caches—most often hash-tables—have been used in purely interpretative systems in order to cache a target address for a set of given instruction operands. The interpreter requires an indirect branch to call the address found in the cache. The premise is that the indirect branch is less expensive than the call to the system default lookup routine. In case of a cache-miss, the interpreter calls the system default lookup routine and places its returned address in the cache. In consequence, using a function pointer via an indirect call instruction eliminates the necessity of having a dynamic translator at all.

Still, using hash-table based techniques is relatively expensive: you need to deal with hashing in order to efficiently retrieve the keys, with collision when placing an element in the hash table, etc. However, we show that we can completely eliminate the need for method caches, too: During dynamic translation, a sequence of interpreter instructions is compiled to its corresponding native machine representation. Within this representation, the inline cache effectively specializes an interpreter instruction to a more efficient derivative—based on the "*dynamic locality of type usage*". Fortunately, we can project this information back at the purely interpretative level: By storing an additional machine word for every instruction within a sequence of bytecodes, we can lift the observed locality to the interpreter level. Consequently, we obtain a dedicated inline cache pointer for every interpreter instruction, i.e., instead of having immutable interpreter operation implementations, this abstraction allows us to think of specific instruction *instances*. At the expense of additional memory, this gives us a more efficient inline caching technique that is more in tune with the original technique of Deutsch and Schiffman [4], too.

Figure 1(a) shows how the ad-hoc polymorphism is resolved in the `BINARY_ADD` instruction of the Python 3.x interpreter. Here, an inline cache pointer would store the addresses of the leaf functions, i.e., either one of `long_add`, `float_add`, `complex_add`, and `unicode_concatenate`, and therefore an indirect jump circumvents the system default lookup path (cf. Figure 1(b)). The functions at the nodes which dominate the leaves need to update the inline cache element. In our example (cf. Figure 1), the `binary_op` function needs to update the inline cache pointer to `long_add`. If, there is no such dominating function (cf. Figure 1(a),



(a) Ad-hoc polymorphism in Python 3.x. (b) Basic Inline Caching `long_add`.

Fig. 1. Illustration of our basic inline caching technique compared to the standard Python 3.x ad-hoc polymorphism.

right branch to `unicode_concatenate`), we have to introduce an auxiliary function that mirrors the operation implementation and acts as a dedicated system default lookup routine for that instruction.

Even though Deutsch and Schiffman [4] report that the "*inline cache is effective about 95% of the time*", we need to account for the remaining 5% that invalidate the cache. Our implementation changes the implementation of the leaf functions to check whether their operands have their expected types. In case we have a cache miss, e.g., we called `long_add` with `float` operands, a call to `PyNumber_Add` will correct that mistake and properly update the inline cache with the new information, i.e., the address of the `float_add` function, along the way.

```
PyObject *long_add(PyObject *v, PyObject *w) {
    if (!(PyLong_Check(v) && PyLong_Check(w)))
        return PyNumber_Add(v, w);

    /* remaining implementation unchanged */
    ...
}
```

Finally, we show how we implement the inline cache pointer in Python 3.1. The interpreter has a conditional instruction format: if an instruction has an argument, i.e., its ordinal number is above some pre-defined threshold, the two consecutive bytes are arguments to that instruction. If the opcode is below that threshold, the next byte contains the next instruction. Hence, two instructions in the array of bytecodes are not necessarily adjacent, which complicates not only instruction decoding, but updating the inline cache pointers, too. Our implementation solves this by encoding the instruction opcode and its argument in one

machine word, and the inline cache pointer in the adjacent machine word. Thus, all instructions have even offsets, while the corresponding inline cache pointers have the subsequent odd offsets (cf. Figure 2).

Fig. 2. Changed instruction format.

In addition to being a more efficiently decode-able instruction format, this enables us to easily update the inline cache pointer for any instruction without having any expensive global references to that instruction. One minor change is still necessary, however: Since we have eliminated the argument bytes from our representation, jumps within the bytecode contain invalid offsets—they have to be relocated to the new offsets in order to work properly.

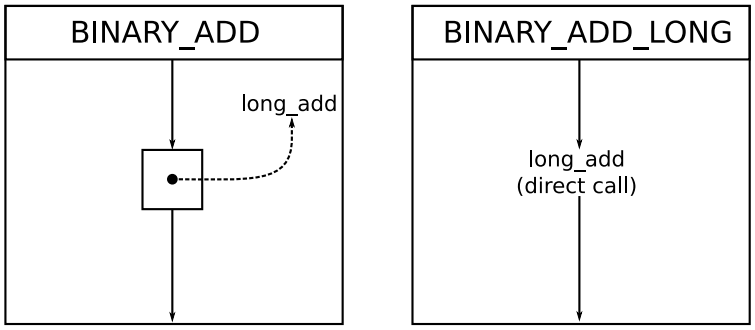
Summing up, this describes a basic and simple, yet more efficient version of an inline caching technique for interpreters without dynamic translation. This basic technique requires significantly more memory space: Instead of one byte for the instruction opcode, this technique requires two machine words per instruction. Therefore, this classic example for trading space for time is not recommended to be applied at all times. Our approach to compensating for this additional memory requirement is to use a simple low-overhead profiling technique to determine which code benefits from inline caching and needs to use this more efficient instruction format.

3 Instruction-Set Extension

Our basic inline caching technique from Section 2 introduces an additional indirect branch per instruction that uses an inline cache. Though this indirect branch is almost always cheaper than calling the system default routine, we can improve on that situation and remove this additional indirect branch completely. The new instruction format enables us to accommodate a lot more instructions than the original one used in Python 3.1: Instead of just one byte, the new instruction format encodes the opcode part in a half-word, i.e., it enables our interpreter to implement many more instructions in common 32 bit architectures (2^{16} instead of 2^8). Although a 64 bit architecture could implement 2^{32} instructions, for practical reasons it is unrealistic to even approach the limit of 2^{16}.

The original inline caching technique requires us to rewrite a call instruction target. In an interpreter without a dynamic translator this equals rewriting an

interpreter instruction; from the most generic instance to a more specific derivative. Figure 3 shows how we can eliminate the inline cache pointer all together by using a specialized instruction that directly calls the `long_add` function.



(a) Basic Inline Caching `long_add`.

(b) New instruction with direct call.

Fig. 3. Instruction-set extension illustrated for operands having `long` type.

Rewriting virtual machine instructions is a well known technique. In the Java virtual machine, this technique is called "quick instructions" [14]. Usually, *quickening* implies the specialization of an instruction towards an operand *value*, whereas our interpretation of that technique uses specialization with respect to the *result* of the system default lookup routine. Another significant difference between the well known application in the Java virtual machine and our technique is that whereas the actual quickening in the JVM is used for intialization, i.e., is only used once per affected bytecode, our technique requires instruction rewriting per cache miss, i.e., in the 5% of cases where our "guess" was wrong, we have to rewrite the instruction to match our new information.

Rewriting the bytecodes is somewhat the opposite of what we described in the previous section. Which approach performs better depends on the underlying native machine hardware. Since the rewriting approach increases the code size of the interpreter dispatch loop, this may have negative performance impacts on architectures with small instruction caches. For these architectures, the basic technique of Section 2 might perform better because of fewer instruction cache misses. On modern desktop and server hardware, however, the rewriting approach is clearly preferable. Figuratively speaking, both techniques are opposite ends on the same spectrum, and the actual choice of implementation technique largely depends on direct evaluation on the target hardware.

3.1 Inline Caching via Quickening

In Python, each type structure contains a list of function pointers that can be used on arguments of that type. Our specialization technique focuses on three sub-structures within that type structure that capture the context of the type:

1. Scalar/numeric context: this context captures the application of binary arithmetical and logical operators to operands of a given type. Examples include: add, subtract, multiply, power, floor, logical and, logical or, logical xor, etc.
2. List context: this context captures the use of a type in list context, e.g. list concatenation, containment, length, repetition (i.e. operation of a list and a scalar), etc.
3. Map context: this context captures the use of type in map context. Operations include the assignment of keys to values in a map, the fetching of values given a key in the map, and the length of the map.

	Context		
Type	Scalar	List	Map
`PyLong_Type`	x		
`PyFloat_Type`	x		
`PyComplex_Type`	x		
`PyBool_Type`	x		
`PyUnicode_Type`	x	x	x
`PyByteArray_Type`		x	x
`PyDict_Type`			x
`PyList_Type`		x	
`PyMap_Type`			x
`PyTuple_Type`		x	x
`PySet_Type`		x	

Table 1. Specialized types by context.

For each of the types in table 1, we can determine whether it implements a scalar-/list-/map-context dependent function. For use in the scalar/numeric context, each type has a sub-structure named `tp_as_number`, which contains a list of pointers to the actual implementations, e.g., the `nb_add` member points to the implementation of the binary addition for that type. A concrete example is for the integrated long type: `PyLong_Type.tp_as_number->nb_add` points to the `long_add` function, which implements the unbounded range integer addition of Python 3.x. We have a short Python program in a pre-compile step that generates the necessary opcode definitions and operation implementations for several types. Currently, the program generates 77 specialized instructions for several bytecode instructions.

Apart from the generation of dedicated instructions, we need to take care of rewriting the instructions, too. In the previous Section 2, we already explained

that we need to instrument suitable places to update the inline cache pointer. Our implementation has a function named `PyEval_SetCurCacheElement` that does that. This function already updates the inline cache pointer of the current instruction, therefore adding code that changes the instruction opcode of the current instruction is easy. Reusing this function as a means to rewrite instruction opcodes also ensures that we can reuse the cache-miss strategy of the basic technique.

Unfolding the Comparison Instruction: Depending on its operand, Python's `COMPARE_OP` instruction chooses which comparator it is going to use. It calls the `cmp_outcome` function which implements comparator selection using a switch statement:

```
static PyObject *
cmp_outcome(int op, PyObject *v, PyObject *w) {
    int res = 0;
    switch (op) {
      case PyCmp_IS:     res = (v == w); break;
      case PyCmp_IS_NOT: res = (v != w); break;
      case PyCmp_IN:     res = PySequence_Contains(w, v);
                         if (res < 0) return NULL;
                         break;
      case PyCmp_NOT_IN: res = PySequence_Contains(w, v);
                         if (res < 0) return NULL;
                         res = !res;
                         break;
      case PyCmp_EXC_MATCH:
        /* more complex implementation omitted! */
```

We eliminate this switch statement for the topmost four cases by promoting them to dedicated interpreter instructions: `COMPARE_OP_IS`, `COMPARE_OP_IS_NOT`, `COMPARE_OP_IN`, `COMPARE_OP_NOT_IN`. This is somewhat similar to an optimization technique that is described by Allen Wirfs-Brock's article on design decisions for a Smalltalk implementation [13], where he argues that it might be more efficient for an interpreter to pre-generate instructions for every (frequent) pair of (opcode, oparg). Since the operand is constant for any specific instance of the `COMPARE_OP` instruction, we assign the proper dedicated instruction when creating and initializing our optimized instruction encoding.

Unfolding the Iteration Instruction: Python has a dedicated instruction for iteration, `FOR_ITER`. It uses a function from the type structure (`tp_iternext`) of the top-of-stack element and calls this function with the top-of-stack element as its argument. This function returns the next value for the iterator variable, which is pushed onto the stack again.

```
TARGET(FOR_ITER)
    /* before: [iter]; after: [iter, iter()] *or* [] */
    v = TOP();
    x = (*v->ob_type->tp_iternext)(v);
    if (x != NULL) {
        PUSH(x);
        DISPATCH();
    }
    if (PyErr_Occurred()) {
        if (!PyErr_ExceptionMatches(PyExc_StopIteration))
            break;
        PyErr_Clear();
    }
    /* iterator ended normally */
    x = v = POP();
    Py_DECREF(v);
    JUMPBY(oparg);
    DISPATCH();
```

There is a set of dedicated types for use with this construct, and we have extracted 15 additional instructions that replace the indirect call of the standard Python 3.1 implementation with a specialized derivative, e.g. by the iterator over a range object, `PyRangeIter_Type`:

```
TARGET(FOR_ITER_RANGEITER)
    v = TOP();
    x = PyRangeIter_Type.tp_iternext(v);
    /* unchanged body */
```

Combination of Variable Caches and Quickening There are several instructions in Python that deal with lookups in environments. For instance, when we examine the `LOAD_GLOBAL` implementation, we find that there is a precedence lookup encoded, i.e., first there is a lookup in the `f->f_globals` member, and if the argument key was not found, a second lookup attempt using the `f->f_builtins` member is tried.

```
TARGET(LOAD_GLOBAL)
    w = GETITEM(names, oparg);
    if (PyUnicode_CheckExact(w)) {
        /* Inline the PyDict_GetItem() calls. */
    }
    /* This is the un-inlined version of the code above */
    x = PyDict_GetItem(f->f_globals, w);
    if (x == NULL) {
        x = PyDict_GetItem(f->f_builtins, w);
        if (x == NULL) {
```

```
            load_global_error:
              format_exc_check_arg(
                      PyExc_NameError,
                      GLOBAL_NAME_ERROR_MSG, w);
              break;
          }
      }
      Py_INCREF(x);
      PUSH(x);
      DISPATCH();
```

Now, dictionary lookup using complex objects is an expensive operation. If we can ensure that no destructive calls, i.e., calls invalidating an inline cached version, occur during the execution, we can cache the resulting object in our inline caching slot of Section 2 and rewrite the instruction to a faster version:

```
TARGET(FAST_LOAD_GLOBAL)
    Py_INCREF(GET_INLINE_CACHE());
    PUSH(GET_INLINE_CACHE());
    DISPATCH();
```

Our current implementation checks whether there occur any `STORE_GLOBAL` instructions in the bytecode, and only then rewrites the instruction. This is a simple way of dealing with this problem and we found no problems in building Python with its standard library and our benchmarks. However, an industrial strength implementation of this technique might require more sophisticated invalidation mechanisms. The same optimization applies to the `LOAD_NAME` instruction.

Unfolding the Call Instruction: The `CALL_FUNCTION` instruction requires the most work. In his dissertation, Hölzle already observed the importance of instruction set design with a case in point on the `send` bytecode in the SELF interpreter, which he mentions to be too abstract for efficient interpretation [9]. The same holds true for the Python interpreter: There are only a few bytecodes for calling a function, and the compiler generates `CALL_FUNCTION` instructions most often. Aside from their use for calling host-level functions, the same bytecode is used for calling C-functions. Because Python is a multi-paradigm programming language, the first issue is complicated by the fact that the targets can be either Python functions or methods—the latter being more complicated because of dynamic binding. The second issue—calling C functions—is important because C function targets can have a multitude of arguments, including variable arguments, as well as named arguments. Since we cannot provide inline caching variants for every possible combination of call types and the corresponding number of arguments, we decided to optimize frequently occurring combinations (cf. Table 2).

	Number of Arguments			
Target	0	1	2	3
C std. args	x	x		
C variable args	x	x	x	x
Python direct	x	x	x	
Python method	x	x	x	

Table 2. Specialized `CALL_FUNCTION` instructions.

4 Evaluation

We used several benchmarks from the computer language shootout game [7]. Since the adoption of Python 3.x is rather slow in the community, we cannot give more suitable benchmarks of well known Python applications, such as Zope, Django, and twisted. All benchmarks were run on an Intel i7 920, with 2.6 GHz running Linux 2.6.28-15 and gcc version 4.3.3. We used modified version of the `nanobench` program of the computer language shootout game [7] to measure the running times of each benchmark program. The `nanobench` program uses the `getrusage` function to get timings for elapsed user and system time. We add both values and use them as the basis for our benchmarks. In order to account for proper measurement and cache effects, we ran each program 1000 successive times with the Intel Turbo Boost technology turned off. This benchmark was repeated 10 times and our data provides averages over those repetitions.

Figure 4 contains our evaluation results. We calculated the speedup by normalizing against the standard Python 3.1 distribution with threaded code and inline caching optimizations turned off. The labels indicate the name of the benchmark and its command line argument combined into one symbolic identifier. The measured inline caching technique represents the technique of Section 3.1 with the modified instruction format of Section 2.

With the exception of the fannkuch benchmark, the combined approach of using threaded code with inline caching is always faster. From negligible improvements ($< 10\%$) in the case of the binarytrees, fasta, and mandelbrot benchmarks, to a significant speedup ($\geq 10\%$) in the case of the nbody, and spectralnorm benchmarks. In the chameneosredux benchmark, we can see that threaded code execution can result in negative performance, too. Yet, this particular benchmark is inline caching friendly, and therefore a combination of both techniques results in a visible speedup.

We find that the particularly beneficial benchmarks contain only a few function calls in the interpreted programming language. To that end there are several possible reasons for this: a) function calls requires the creation of stack frame objects, as already observed in the BrouHaHa implementation of Smalltalk [15], the creation of equivalent Smalltalk Context objects is expensive and therefore an inline cached function call is less expensive; b) our profiling infrastructure is too expensive, and c) our `CALL_FUNCTION` inline caching schema is restricted to the number of arguments and call types in Table 2, but in those benchmarks the

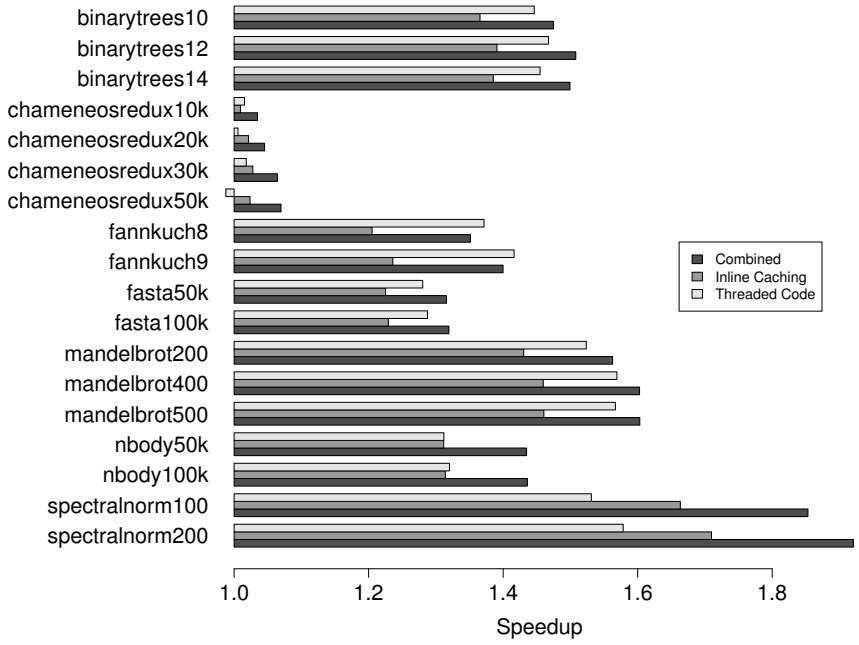


Fig. 4. Achievable speedups on various benchmarks.

expensive calls cannot be optimized and have to resort to the default method lookup. Further investigation is necessary to identify the root cause.

5 Related Work

In his PhD thesis of 1994 [9], Hölzle mentions the basic idea of the data structure underlying our basic technique of Section 2. The major difference is that we are not only proposing to use this data layout for `send`—or `CALL_FUNCTION` instructions in Python's case—but for all instructions, since there is enough caching potential in Python to justify that decision. Hölzle addresses the additional memory consumption issue, too. We use a simple low-overhead invocation based counter heuristic to determine when to apply this representation, it is only created in code we know is *hot*. Therefore, we argue that the increased memory consumption is negligible—particularly since the memory consumption caused by state of the art just-in-time compilers is much more intensive than what our approach requires.

In 2008, Haupt et al. [8] published a position paper describing details of adding inline caching to bytecode interpreters, specifically the Squeak interpreter. Their approach consists of adding dedicated inline caching slots to the

activation record, similar to dealing with local variables in Python or the constant pool in Java. In addition to a one-element inline cache, they also describe an elegant object-oriented extension that enables a purely interpretative solution to polymorphic inline caches [10]. The major difference to our approach lies in the relative efficiencies of the techniques: Whereas our techniques are tightly interwoven with the interpreter infrastructure promising efficient execution, their technique relies on less efficient target address lookup in the stack frame.

Regarding the use of lookup caches in purely interpretative systems, we refer to an article detailing various concerns of lookup caches, including efficiency of hashing functions, etc., which can be found in "Smalltalk-80: Bits of History, Words of Advice" [13]. Kiczales and Rodriguez describe the use of per-function hash-tables in a portable version of common lisp (PCL), which may provide higher efficiency than single global hash tables [12]. The major difference to our work is that our inline cache does not require the additional lookup and maintenance costs of hash-tables.

Lindholm and Yellin [14] provide details regarding the use of quick instructions in the Java virtual machine. Casey et al. [3] describe details of quickening, superinstructions and replication. The latter technical report provides interesting details on the performance of those techniques in a Java virtual machine implementation. The major difference to our use of instruction rewriting as described in Section 3 is that we are using quickening for inline caching. We are not aware of any other work in that area. However, our approach to replication is similar to theirs, as is the use of a code generator to compensate for the increased maintenance effort implied by adding new instructions. Our techniques do not use any form of superinstructions.

6 Conclusion

Inline caching is an important optimization technique for high abstraction level interpreters. We report achievable speedups of up to 1.71 in the Python 3.1 interpreter. Our quickening based technique from Section 3 uses the instruction format described in Section 2. Therefore, the measured performance includes the compensation times for the profiling code and the creation of the new instruction format. However, it is possible to use the quickening based inline caching approach without the new instruction format—thereby eliminating the compensation overhead, which we expect to positively affect performance. Future work on such an architecture will quantify these effects.

Efficient inline caching without dynamic translation is an optimization technique targeting operation implementation. Therefore, it is orthogonal to optimization techniques focusing on instruction dispatch and both techniques can be applied together. In the `spectralnorm` benchmark the application of both techniques results in a speedup of 1.92—only slightly lower than the maximum reported speedup of 2.02 achieved by efficient interpreters using threaded code alone [6].

Acknowledgments

I want to thank Urs Hölzle for details regarding the history of the basic technique of section 2. Furthermore, I am particularly grateful to Jens Knoop and Anton Ertl for valuable discussions and feedback on earlier drafts of this paper.

References

1. Bell, J.R.: Threaded code. Communications of the ACM 16(6), 370–372 (1973), the original reference for threaded code
2. Brunthaler, S.: Virtual-machine abstraction and optimization techniques. In: Proceedings of the 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE '09). pp. 19–30. Elsevier, York, UK (March 2009)
3. Casey, K., Ertl, M.A., Gregg, D.: Optimizations for a java interpreter using instruction set enhancement. Tech. Rep. 61, Department of Computer Science, University of Dublin. Trinity College (September 2005), `https://www.cs.tcd.ie/publications/tech-reports/reports.05/TCD-CS-2005-61.pdf`
4. Deutsch, L.P., Schiffman, A.M.: Efficient implementation of the Smalltalk-80 system. In: Proceedings of the SIGPLAN '84 Symposium on Principles of Programming Languages (POPL '84). pp. 297–302. ACM, New York, NY, USA (1984)
5. Ertl, M.A.: Threaded code variations and optimizations. In: EuroForth. pp. 49–55. TU Wien, Vienna, Austria (2001)
6. Ertl, M.A., Gregg, D.: The structure and performance of efficient interpreters. Journal of Instruction-Level Parallelism 5 (2003)
7. Fulgham, B.: The computer language benchmarks game. `http://shootout.alioth.debian.org/`
8. Haupt, M., Hirschfeld, R., Denker, M.: Type feedback for bytecode interpreters. Position Paper. (ICOOOLPS '07). `http://scg.unibe.ch/archive/papers/Haup07aPIC.pdf` (2008), `http://scg.unibe.ch/archive/papers/Haup07aPIC.pdf`
9. Hölzle, U.: Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Ph.D. thesis, Stanford University, Stanford, CA, USA (1995)
10. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP '92). pp. 21–38. Springer-Verlag, London, UK (1991)
11. Hölzle, U., Ungar, D.: Optimizing dynamically-dispatched calls with run-time type feedback. In: Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI '94). pp. 326–336 (1994)
12. Kiczales, G., Rodriguez, L.: Efficient method dispatch in PCL. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90). pp. 99–105. ACM, New York, NY, USA (1990)
13. Krasner, G. (ed.): Smalltalk-80: bits of history, words of advice. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1983)
14. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Boston, MA, USA, first edn. (1997)

15. Miranda, E.: Brouhaha–a portable smalltalk interpreter. In: Proceedings of the SIGPLAN '87 International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87). pp. 354–365. ACM, New York, NY, USA (1987)
16. Vitale, B., Abdelrahman, T.S.: Catenation and specialization for Tcl virtual machine performance. In: IVME '04: Proceedings of the 2004 Workshop on Interpreters, virtual machines and emulators (IVME '04). pp. 42–50. ACM, New York, NY, USA (2004), General Chair-Michael Franz and Program Chair-Etienne M. Gagnon

Lambdas und Schleifen in monotonen Logikprogrammen

Ulrich Neumerkel
Technische Universität Wien
Institut für Computersprachen
ulrich@mips.complang.tuwien.ac.at

Zusammenfassung

Lambda-Abstraktionen und Schleifenkonstrukte sind in der Logikprogrammierung bisher kaum aufgenommen worden. Diese Zurückhaltung hängt unmittelbar mit den algebraischen Eigenschaften der verwendeten Konstrukte zusammen. Die bisher vorgeschlagenen Ansätze verletzen grundlegende Eigenschaften wie die der Monotonie, wodurch nicht nur eine deklarative Betrachtungsweise verunmöglicht wird und diagnostische Verfahren stark eingeschränkt werden, sondern auch effiziente Implementierungen behindert werden. Mit der vorgestellten Umsetzung von Lambda-Ausdrücken werden die meisten Mängel unmittelbar vermieden. Es eröffnen sich nun neue Möglichkeiten monotone Schleifenkonstrukte einzubinden.

1 Einführung

Das Programmieren höherer Ordnung in purem, monotonen Prolog stellt seit jeher ein wenig beachtetes Gebiet dar. Es ist zwar prinzipiell auch in Prolog möglich, ähnlich der Funktionalen Programmierung vorzugehen, dennoch fanden diese Konstrukte bisher keinen Anklang. Mit ein Grund liegt darin, dass selbst Lambda-Abstraktionen nicht angenommen wurden. Man hat geradezu den Eindruck, dass das Lambda in Prolog *littera non grata* ist. Sehr ähnlich verhält es sich auch mit Schleifenkonstrukten. Auch hier zieht man es in Prolog vor, selbst einfache Schleifen als direkt rekursive Prädikate anzuschreiben, statt sich einer kompakteren Schreibweise zu bedienen. Die damit verbundenen Nachteile, wie etwa die Verwendung von sonst nicht weiter benötigten Namen nimmt man billigend in Kauf. Die bisherigen Versuche, endlich robuste Lambda-Abstraktionen und Schleifenkonstrukte zu entwickeln sind letztlich fehlgeschlagen, weil die Monotonieeigenschaften unnotwendigerweise verletzt werden und die Notwendigkeit eines komplexeren Übersetzungsvorgangs erforderlich ist. Rekursive monotone Programme werden durch diese Schleifenkonstrukte in Programme umgewandelt, die gelegentlich die Monotonieeigenschaft verletzen. Gerade in Constraintprogrammierung ist dies ein nicht zu vernachlässigendes Risiko. Dadurch werden etwa die Ergebnisse diagnostischer Verfahren verfälscht. Bevor wir diese neuen Lambda-Ausdrücke vorstellen, betrachten wir eine Form der Monotonieeigenschaft genauer.

2 Verallgemeinerungen

Anhand eines Beispiels der klassischen Programmierung höherer Ordnung wie sie schon seit langem in Prolog bekannt ist [1] verbunden mit CLPFD-Constraints [8] suchen wir Verallgemeinerungen eines monotonen Programms. Die Beispiele sind direkt in SWI-Prolog ausführbar.

```
maplist(_Pred, []).
maplist(Pred, [E|Es]) :-
   call(Pred, E),
   maplist(Pred, Es).

?- maplist(#\=(1), [2,1,3]).  % Entspricht 1 #\= 2, 1 #\= 1, 1 #\= 3
false.
```

In diesem Programm scheitert das Ziel — angezeigt durch `false`. Um zu verstehen warum, ist es naheliegend die dazugehörige Verallgemeinerungen zu betrachten. So scheitert bereits ein Ziel, bei dem `2` und `3` durch Variable ersetzt wurden.

```
?- maplist(#\=(1), [X,1,Y]).  % Entspricht 1 #\= X, 1 #\= 1, 1 #\= Y
false.
```

Wie weit wir mit derartigen Verallgemeinerung gehen dürfen, ohne mit Nichttermination in Berührung zu kommen, lässt sich oft durch Terminationsanalysen [6] bestimmen. Üblicherweise wird hier laut Terminationsinferenz die Länge der Liste endlich bleiben müssen. In diesem konkreten Fall könnten wir aber noch weiter gehen, womit wir allerdings nun möglicherweise Nichttermination in Kauf nehmen müssen.

```
?- maplist(#\=(X), [_,X|_]).
false.
```

Eine weitere Verallgemeinerung der übriggebliebenen Liste führt nun zu konkreten Lösungen. Den Term `#\=(X)` weiterzuverallgemeinern führt lediglich zu einem Instanzierungsfehler. Wir haben offenbar die maximal erreichbare Verallgemeinerung erreicht. Offenbar liegt die Ursache für das Scheitern der Anfrage in dem Umstand, dass ein Wert in der Liste ungleich zu sich selbst sein soll. Diese Technik kann als eine Adaptierung von Slicing-Techniken [9] gesehen werden.

Derartige Verallgemeinerungen sind nur in monotonen Logikprogrammen sinnvoll. Sobald wir uns auf die monotone Eigenschaft des Programms nicht mehr verlassen können, können derartige Verallgemeinerungen keine Aussagen mehr über das Programm tätigen. Es wäre nun wünschenswert, dass diese Monotonieeigenschaft auch für weitere Sprachkonstrukte gilt. Insbesondere für Lambda-Abstraktionen und den damit verbundenen Schleifenkonstrukten.

Traditionelle Lambda-Abstraktionen, wie sie aus der Funktionalen Programmierung bekannt sind, eignen sich nicht direkt zu einer Darstellung in Prolog. Um unterscheiden zu können, ob eine Variable frei oder gebunden ist, bedarf es einer Analyse des Kontexts. Erst durch diese Analyse kann eine Entscheidung getroffen werden. Lambda-Abstraktionen konnten so bislang nicht einfach in einer Bibliothek zur Verfügung gestellt werden. Je nach Realisierung dieser Analyse kann das Ergebnis einer derartigen Analyse anders aussehen. Dies lässt sich durch die Einbettung eines Ziels nachvollziehen. So wird etwas ein `Ziel` durch `NV = Ziel, NV` ersetzt, und damit die Analyse zu einem späteren Zeitpunkt erst ausgeführt.

Unsere neue Realisierung von Lambda-Abstraktionen basiert ausschließlich auf Prädikatsdefinitonen. Es ist keinerlei Veränderung des Übersetzers erforderlich. Eine einfache Bibliothek ist ausreichend um in ISO Prolog verwendet werden zu können [5].

3 Lambdas in ISO Prolog

Lambda-Ausdrücke bestehen aus zwei Komponenten. Ein Teil dient der Umbenennung, der andere der Parameterübergabe.

3.1 Umbenennung

In einem parameterlosen Lambda-Ausdruck \ werden nun sämtliche Variable als lokale Variable betrachtet. Bindungen von ihnen sind also nicht außen sichtbar.

```
?- X+Y = 1+2.
X = 1,
Y = 2.
```

```
?- \ ( X+Y = 1+2 ).
true.
```

Im Unterschied zum herkömmlichen Ansatz, dem Übersetzer die Einteilung der Variable in freie und gebundene zu überlassen, muss man nun selbst die Variablen einzeln deklarieren. Dazu wird `+\` verwendet. Werden sämtliche Variable deklariert, so erübrigt sich der Lambda-Ausdruck. Variable die undeklariert sowohl in einem Lambda-Ausdruck als auch außerhalb vorkommen, sollten idealerweise als Fehler gemeldet werden. Bei einfacher Interpretation führen sie zu unbeabsichtigten Bindungen.

```
?- X+\ ( X+Y = 1+2 ).
X = 1.
```

```
?- [X,Y]+\ ( X+Y = 1+2 ).
X = 1.
Y = 2
```

3.2 Parameter

Neben der Umbenennung von Variablen können nun Parameter explizit über `call/2..` übergeben werden. Für sich genommen ruft `call(Cont, Arg)` einen Term (Kontinuation) `Cont` mit noch einem weiteren Argument auf. Dieses vordefinierte Prädikat geht auf O'Keefe zurück [2]. Auf dieselbe Art und Weise können nun Parameter verwendet werden. Allerdings fehlt hier die Umbenennung. Man kann also noch die Bindung der Variable `Y` erkennen.

```
?- call(#=<(3),X).
X in 3..sup.

?- 3 #=< X.
X in 3..sup.
```

```
?- call(Y^(Y#>=3),X).
Y = X,
X in 3..sup.
```

Erst durch das gemeinsame Verwenden von `\` und `^` entstehen nun die eigentlichen Lambdas. Diese können nun auch mit globalen Variablen verwendet werden. Die `^`-Schreibweise geht auf Pereira zurück [3], der sie allerdings nicht zu einem vollständigen Lambdakonstrukt ausgebaut hat.

```
?- call(\Y^(Y#>=3),X).
X in 3..sup.

?- maplist(\Y^(Y#>=3),[X1,X2]).
X1 in 3..sup,
X2 in 3..sup.

?- maplist(Z+\Y^(Y#>=Z),[X1,X2]).
X2#>=Z,
X1#>=Z.
```

4 Realisierung

Zur einfachen Umbenennung verwenden wir lediglich das vordefinierte Prädikat `copy_term/2`. Da Umbenennungen aber nicht nur bei einem einfachen Ziel (erste Zeile) vorkommen können, sondern auch bei einem Konstrukt, das ein oder mehrere weitere Argumente benötigt, einer *continuation*. Die eigentlichen Parameter (`V1`, ...) werden unbehelligt weitergereicht.

```
\(FC) :- copy_term(FC,C),call(C).
\(FC, V1) :- copy_term(FC,C), call(C, V1).
\(FC, V1, V2) :- copy_term(FC,C), call(C, V1, V2).
    ...
```

Für Umbenennungen in Gegenwart globaler Variable benötigen wir eine Operatordeklaration. Die globalen Variablen `GV` werden nun nicht mehr kopiert, sondern nur mehr die restlichen variablen in `FC`. Prozedural gesehen werden die `GV` kopiert und gleich wieder mit sich gleichgesetzt. Auch hier bleiben die Parameter unverändert.

```
:- op(201,xfx,+\).

+\(GV, FC) :- copy_term(GV+FC,GC+C), call(C).
+\(GV, FC, V1) :- copy_term(GV+FC,GV+C), call(C, V1).
+\(GV, FC, V1, V2) :- copy_term(GV+FC,GV+C), call(C, V1, V2).
     ...
```

Letztendlich fehlt noch die Parameterübergabe. Hier wird jeweils das erste Argument mit dem ersten Argument der Parameter verbunden. Diese Parameter stellen in gewisser Weise eine auscompilierten Stapel dar.

```
^(V1, Goal, V1) :- call(Goal).
^(V1, Goal, V1, V2) :- call(Goal, V2).
^(V1, Goal, V1, V2, V3) :- call(Goal, V2, V3).
    ...
```

5 Schluss

Wir haben einen neuen besonders einfachen Ansatz von Lambda-Ausdrücken für Prolog vorgestellt, der lediglich durch direkte Prädikatsdefinitionen darstellbar ist und dennoch Variablenquantifikationen so umsetzt, dass es zu keinen Monotonieverletzungen kommt. Alle verwendeten Mittel sind zwar schon seit langem bekannt, dennoch ist die konkrete Zusammenstellung bisher noch nicht so vorgekommen. Die vollständige Implementierung der Bibliothek `lambda`, die auch Constraints (eine implementierungsspezifische Erweiterung der Norm) kann sowohl in SWI-Prolog als auch YAP direkt verwendet werden. Sie befindet sich unter http://www.complang.tuwien.ac.at/ulrich/Prolog-inedit/lambda.pl

Literatur

[1] D. H. D. Warren. Higher-Order Extensions to Prolog - Are They Needed?, In Hayes, Michie and Pao, Machine Intelligence 10. Ellis Horwood, 1982. Originally: International Machine Intelligence Workshop, Cleveland, April 1981, DAI Research Paper 154.

[2] R. O'Keefe. The Craft of Prolog. MIT-Press. 1990.

[3] F. C. N. Pereira, St. M. Shieber Prolog and Natural-Language Analysis. (CSLI Lecture Notes 10, ISBN 0-937073-18-0), 1987. Digital version

[4] D. Cabeza, M. Hermenegildo, and J. Lipton Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction ASIAN 2004, LNCS 3321, pp. 93-198, 2004. PDF

[5] ISO/IEC 13211-1 Programming languages - Prolog - Part 1: General core. 1995.

[6] F. Mesnard, U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. 8th Static Analysis Symposium (SAS'01), Paris 2001.

[7] U. Neumerkel, St. Kral. Declarative program development in Prolog with GUPU. 12th Workshop on Logic Programming Environments (WLPE), Copenhagen 2002.

[8] M. Triska, U. Neumerkel, J. Wielemaker. Better Termination for Prolog with Constraints. WLPE, Udine 2008.

[9] M. Weiser. Programmers Use Slices When Debugging. CACM 25(7): 446-452 (1982)

A formalisation of the OSEK concurrency model

Martin Schwarz

Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
{schwmart}@in.tum.de

1 Introduction

The OSEK[1] Operating System Specification [11] was created to increase portability of applications written for embedded systems with interrupt-driven concurrency. Its development started in 1993 as a joint effort of the European car industry and has now resulted in a partially ISO certified open standard. While a wealth of techniques are available for analysing sequential programs, much less is known about the analysis of interrupt-driven programs with priorities and task activation. Despite of the complex control-flow, programs to be executed on an OSEK-compliant system are meant to exhibit an essentially sequential behaviour. The goal is to make this explicit and exploit it for transferring techniques for the analysis of sequential programs to OSEK programs.

The OSEK specification defines a unified operating system (OSEK OS), which executes the tasks and interrupts of OSEK programs in a well-defined manner and provides a set of library functions for resource management and scheduling. We shall for convenience consider interrupts as a subset of tasks. Programs written for an OSEK OS consist of two parts: a static description file and C-files containing the actual code. In the description file the structure of the program is outlined and the attributes of each task are defined. Examples of information provided there can be seen in the code snippet in Figure 1 and include the priorities of tasks, the autostart flag which determines whether the program starts execution with the given task ready for execution, the number of instances of a task that can exist simultaneously, and the sets of accessible resources for each task. Interrupts have a more fixed behaviour, e.g. can't autostart, limited to one instance, and therefore define less attributes. We assume this information to be accessible to us through functions generated via preprocessing.

The basic scheduling policy of OSEK OS is to work through the activated tasks in priority order. Once started, a task will run to termination unless a task of higher priority is activated and pre-empts it, i.e. no time-slicing is used. This property allows the translation of OSEK programs to a sequential, stack-based execution model.

For synchronisation the Priority Ceiling Protocol (PCP) is used. The key idea of the PCP is to raise the priority of a task which has acquired a resource to the highest priority of all tasks declared to use that resource. We refer to

[1] "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug"

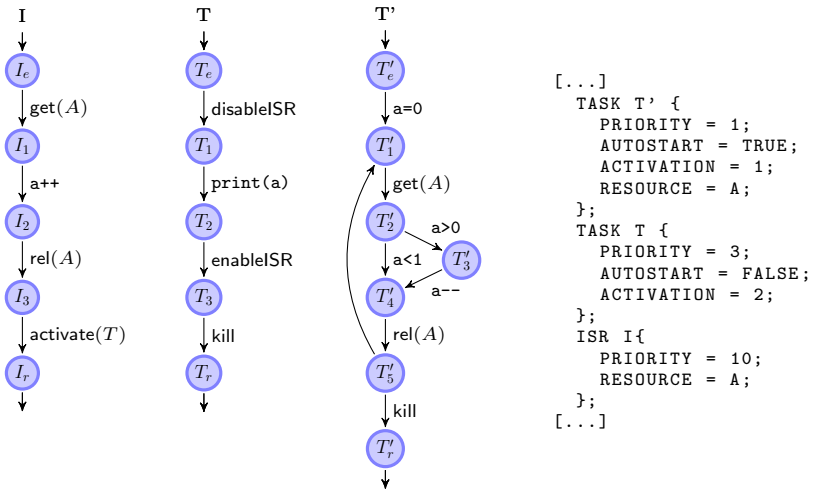


Fig. 1. Example program.

this priority as the *ceiling priority* of the resource. This policy ensures dead-lock freedom and minimises priority inversion, where a high-priority task waits for a lower-priority task to release a resource.

The program in Figure 1, an instance of a consumer-producer scheme, will serve as a running example. The program variable `a` is a counter of items available for consumption. The task T' consumes an item (not shown) and decreases the counter if there is something to be consumed; otherwise, it waits until the next item arrives. The ISR I is triggered as soon as a new item is produced. It increases the counter `a` and activates the printing task T. This small program already shows some of the interesting consequences of the PCP. As T has a higher priority than T' the printed output will always be at least one because T' will only be scheduled to resume after T has terminated.

The priorities place restrictions on the execution, ruling out many interleavings which otherwise would constitute a data race. For example, while holding the resource A, the task T' has a ceiling priority of 10 because the highest priority task declared to use the resource is I, which has a static priority of 10. Therefore, T' cannot be interrupted by I when holding this resource. Note that as long as the resource A is assigned to I in the description file, there is no data race even when the resource acquisition in the body of I is removed. This is because the interrupt cannot be pre-empted by lower-priority tasks and the ceiling priority of T' when it acquires the resource A is still 10, which ensures that it cannot be pre-empted by I. Only if we also remove the declaration that I may acquire A from the description file, can the task T' be interrupted as acquiring the resource no longer raises its priority. As will be seen in Section **??**, our analysis is sensitive to these distinction. We here provide a brief glimpse of our approach. To verify the absence of races in the program, we compute the

following information for the nodes where data is accessed:

Node	Task	Resource	Queue	Offense	Defense
I_1	I	$\{A\}$	$\{\emptyset\}$	10	10
T_1	T	$\{ISR\}$	$\{\emptyset, \{T\}\}$	3	10
T_3'	T'	$\{A\}$	$\{\emptyset\}$	1	10

As there is no pair of accesses to the variable a where one access has a strictly higher *offensive* priority than the *defensive* priority of the other, we conclude that the program is free from races.

2 Semantics of OSEK Programs

This section explains the principles of the OSEK OS and the PCP. Simultaneously we introduce a small step operational semantics which captures these principles. More precisely, we model the behaviour of systems which conform to the BCC2 conformance class of the OSEK specification. This excludes event-based task communication, the precise analysis of which is undecidable. For most OSEK library functions we use shortened versions of the OSEK names, replacing GetResource() with get(), for example.

Formally we notate the set of all tasks as Task and the set of all resources as Res. Each task $q \in \mathsf{Task}$ is represented as a *control flow graph* (CFG) as in Figure 1. A control flow graph $G_q = (N_q, E_q, q_e, q_r)$ consists of a set N_q of *program points*; a set of edges $E_q \subseteq N_q \times \mathsf{Stmt} \times N_q$ annotated with commands; a special *entry point* $q_e \in N_q$; and a special *return point* $q_e \in N_q$. We assume here that the program points of different procedures are disjoint, and denote by N and E the sets of all nodes and edges, respectively.

Since there is no *main*-task in an OSEK program, we construct an artificial *main*-task which consists of one node $main_e$ and one edge $(main_e, \mathsf{schedule}, main_e)$. As actual priorities are positive we set -1 as the priority of the main task which ensures that it does not interfere wiht the actual program. Execution starts at $main_e$, with no resources held and all tasks ready which according to the OIL-file, possess the AUTOSTART attribute. If there are no auto-starting tasks, nothing happens until an interrupt occurs.

To handle priorities we compute a function $\mathcal{P} : 2^{\mathsf{Res}} \to \mathbb{N}$ that maps sets of resources to the maximum of ceiling priorities of resources in the set. In the example $\mathcal{P}(A) = 10$.

During the execution of a OSEK program, the tasks move between the states *suspended*, *ready* or *running*. At most one task is *running* and when a task terminates it becomes *suspended*. In our semantics the current node is *running*. *Ready* tasks are kept in a priority queue Q which we represent as a list. Since the priority depends on the set of held resources, it is stored along with the node. The function $hd(\mathrm{Q})$ yields the first, highest priority, element and $tl(\mathrm{Q})$ yields the queue without the head. In case the queue is empty hd returns $\langle main, \emptyset \rangle$.

The auxiliary functions *add* and *push* insert an element into the queue as the newest or oldest element of its priority, respectively. Since the amount of

instances of a task q is limited, we write this number as $|q|$, $add(u, \mathrm{Q})$ may return Q unchanged if $u \in N_q$ and $|q|$ is reached. With π_i denoting the projection to the i-th component of a tuple and $\mathrm{Q}_{|q}$ the restriction of the queue to only nodes in N_q, *add* and *push* are defined as follows:

$$add(\langle u, R\rangle, \mathrm{Q}) = \begin{cases} \mathrm{Q} & u \in N_q,\ |\mathrm{Q}_{|q}| \geq |q| \\ \langle u, R\rangle : \mathrm{Q} & \mathcal{P}(R) > \mathcal{P}(\pi_2(hd(\mathrm{Q}))) \\ hd(\mathrm{Q}) : add(\langle u, R\rangle, tl(\mathrm{Q})) & \text{otherwise} \end{cases}$$

$$push(\langle u, R\rangle, \mathrm{Q}) = \begin{cases} \langle u, R\rangle : \mathrm{Q} & \mathcal{P}(R) \geq \mathcal{P}(\pi_2(hd(\mathrm{Q}))) \\ hd(\mathrm{Q}) : push(\langle u, R\rangle, tl(\mathrm{Q})) & \text{otherwise} \end{cases}$$

In our example program of Figure 1, $|T| = 2$ and $|T'| = 1$. Since I is an interrupt, $|I| = 1$. Since always one task will be running, there are at most three tasks in the queue.

According to the OSEK specification, the currently running task is counted as one active instance. Our definition of *add* as is, does not take care of this. This counting policy, though, can be simulated by pushing a copy of the running task into the queue before calling *add* and removing it again after this call.

Our semantics only considers the set of held resources to determine the priority of a task. According to the OSEK specification, though, further information must be taken into account. These are the static priorities defined in the description file; various OSEK library functions disabling interrupts as well as locking of the scheduler (viewed as an implicit resource) through get. This behaviour of the scheduler suggests to reduce all these cases to acquiring artificial resources via get. Accordingly, we create distinct artificial resources for the respective library functions, assign them to all affected tasks in the description file and replace the library function calls with $\mathsf{get}()$ for the corresponding artificial resources.

For example program of Figure 1, this means that the command disableISR is replaced by $\mathsf{get}(r_{ISR})$ where the new artificial resource r_{ISR} is assigned to I. The scheduler resource r_{sched} would be assigned to all tasks except interrupts.

Analogously, the static priority of a task q is represented by the artificial resource r_q which is assigned solely to q. The corresponding commands $\mathsf{get}()$ and $\mathsf{rel}()$, we include directly in the semantics instead of the source in order to avoid spurious pre-emptions. With these artificial resources all priority information is bundled in the set of held resources.

A state of our semantics consists of the current node $u \in N$, the set of held resources $R \in 2^{\mathsf{Res}}$ of the running task, and the queue $\mathrm{Q} \in \mathsf{Queue}(N \times 2^{\mathsf{Res}})$ of ready tasks. We don't explicitly carry around components for other global or intra-task information.

For commands c which are not related to scheduling, we write $c \in$BASIC and bundle them in a single rule. Besides all C statements, BASIC contains the OSEK commands $\mathsf{get}()$, $\mathsf{rel}()$ and $\mathsf{activate}()$ which operate on the state as follows:

$$[\![\mathsf{get}(r)]\!]\langle R, \mathrm{Q}\rangle = \langle R \cup \{r\}, \mathrm{Q}\rangle \qquad [\![\mathsf{rel}(r)]\!]\langle R, \mathrm{Q}\rangle = \langle R \setminus \{r\}, \mathrm{Q}\rangle$$
$$[\![\mathsf{activate}(q)]\!]\langle R, \mathrm{Q}\rangle = \langle R, add(\langle q_e, \{r_q\}\rangle, \mathrm{Q})\rangle$$

$$\frac{(u,c,v) \in E \qquad c \in \text{BASIC} \qquad \langle R', \mathrm{Q}' \rangle = [\![c]\!]\langle R, \mathrm{Q} \rangle}{\langle u, R, \mathrm{Q} \rangle \implies \langle v, R', \mathrm{Q}' \rangle} \text{ BASIC}$$

$$\frac{(u, \mathsf{schedule}, v) \in E \qquad \langle u', R' \rangle = hd(\mathrm{Q}) \qquad \mathcal{P}(R) < \mathcal{P}(R')}{\langle u, R, \mathrm{Q} \rangle \implies \langle u', R', push(\langle v, R \rangle, tl(\mathrm{Q})) \rangle} \text{ SWITCH}$$

$$\frac{(u, \mathsf{schedule}, v) \in E \qquad \langle u', R' \rangle = hd(\mathrm{Q}) \qquad \mathcal{P}(R) \geq \mathcal{P}(R')}{\langle u, R, \mathrm{Q} \rangle \implies \langle v, R, \mathrm{Q} \rangle} \text{ STAY}$$

$$\frac{(u, \mathsf{kill}, v) \in E \qquad \langle u', R' \rangle = hd(\mathrm{Q})}{\langle u, R, \mathrm{Q} \rangle \implies \langle u', R', tl(\mathrm{Q}) \rangle} \text{ KILL}$$

$$\frac{q \in \mathsf{Irpt} \qquad R' = \{r_q\} \qquad \mathcal{P}(r_q) > \mathcal{P}(R)}{\langle u, R, \mathrm{Q} \rangle \Rightarrow \langle q_e, R', add(\langle u, R \rangle, \mathrm{Q}) \rangle} \text{ IRPT}$$

Fig. 2. Queue-based semantics ($\Rightarrow$).

The PCP is realised by the *scheduler* routine of the OSEK OS. The scheduler is not constantly monitoring the queue. Instead, it acts when certain conditions occur or library functions are called. Scheduling, e.g., is invoked when a resource is released via the command rel(), when readying a task via activate(), or terminating a task via kill. The scheduler can also explicitly be called via the command schedule. In the example from the previous section, scheduling may occur at the nodes T_r, T_5' and T_r'. The completion of an interrupt calls the scheduler, hence a scheduling may indirectly also occur, e.g., at node T_1', allowing T to execute before T' resumes. On the other hand, the nodes T_2' to T_4' are visited contiguously as the priority at these nodes is so high that I is not allowed to run, and the nodes T_1 and T_2 are also visited contiguously since interrupts are disabled.

In order to make all calls of the scheduler explicit, we remove all implicit triggers and instead insert the command schedule after every triggering command. This separates commands modifying the program state from commands modifying the control flow. The only exception is the command kill where the running task is replaced with the next task which the scheduler would select. This is expressed by the KILL-rule in Figure 2. This may, due to the queue being empty, result in returning to the artificial node *main* where the program would idle until an interrupt occurs.

When the scheduler is called, it selects the ready task with the highest priority from the queue, switches its state into running and starts executing it. Should several tasks have the same priority, the oldest one is selected (see Figure 3). When a tasks becomes running, it keeps its age. In case of pre-emption, it will still be the oldest among the same priority tasks. This behaviour is already captured by the functions *push* and *add* defined above. The next task to be selected from the queue Q is thus retrieved by $hd(\mathrm{Q})$.

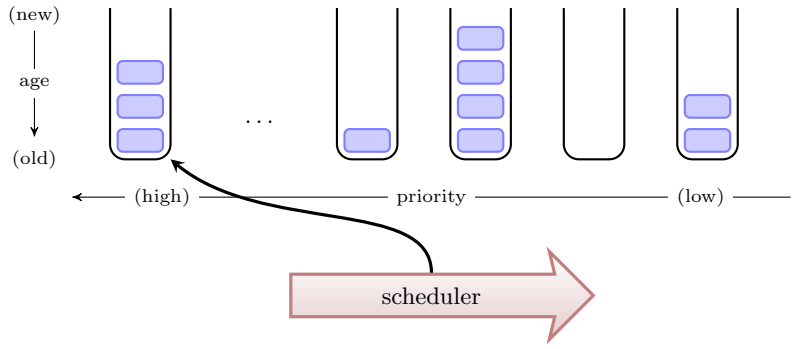


Fig. 3. Scheduling of tasks in the queues

After selecting the next task, the scheduler checks whether it has a strictly higher priority than the running task. If not, the scheduler will do nothing. See the STAY-rule. Otherwise, pre-emption occurs. This means that the current running task is stopped, its state is changed to ready and the selected task becomes running. See the SWITCH-rule.

In our semantic description, we used $[\![\bullet]\!]$ in the BASIC-rule to refer to the standard semantics of C. A specific analysis may replace this with a suitable abstraction of the semantics without interfering with the scheduling. The queue-based semantics of OSEK can be extended with possibly recursive functions. In this case, a configuration is represented as a call-stack for the running task together with a task queue which now consists of the call-stacks of fresh or pre-empted tasks.

3 Stack-Based Semantics of OSEK

In this section, we show that the queue-based small-step semantics of the last section is equivalent to a stack-based small-step semantics. There is almost a one-to-one correspondence between the rules of the semantics of Figure 2 and the rules of the new stack-based semantics as presented in Figure 4. Only the KILL-rule is split into two cases. What is different, however, are the data structures for storing ready tasks. Instead of maintaining a single queue, we maintain a stack of pre-empted tasks together with a simpler and smaller queue of names of fresh tasks only. Therefore, a call of the scheduler behaves similar to an indirect function call, where the SWITCH-rule and RESUME-rule, act as *call* and *return*, respectively. The two cases of the KILL-rule refer to resuming a pre-empted task from the stack and to starting the next task from the queue. The functions *add* and *push* still apply $\mathcal{P}$ to the resource set of their first argument. Now, however, this set consists only of the artificial resource corresponding to the static priority of the added task. This resource can be retrieved directly from the name of the task and therefore need not be stored inside the queue.

$$\frac{(u,c,v) \in E \qquad c \in \text{BASIC} \qquad \langle R', \mathrm{Q}' \rangle = [\![c]\!] \langle R, \mathrm{Q} \rangle}{\langle \Gamma \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle v, R'\rangle, \mathrm{Q}'\rangle}\ \text{BASIC}$$

$$\frac{(u,\mathsf{schedule},v) \in E \qquad R' = \{r_{hd(\mathrm{Q})}\} \qquad \mathrm{Q}' = tl(\mathrm{Q}) \qquad \mathcal{P}(R) < \mathcal{P}(R')}{\langle \Gamma \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle v, R\rangle \,;\langle hd(\mathrm{Q})_e, R'\rangle, \mathrm{Q}'\rangle}\ \text{SWITCH}$$

$$\frac{(u,\mathsf{schedule},v) \in E \qquad R' = \{r_{hd(\mathrm{Q})}\} \qquad \mathcal{P}(R) \geq \mathcal{P}(R')}{\langle \Gamma \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle v, R\rangle, \mathrm{Q}\rangle}\ \text{STAY}$$

$$\frac{(u,\mathsf{kill},v) \in E \qquad R' = \{r_{hd(\mathrm{Q})}\} \qquad \mathrm{Q}' = tl(\mathrm{Q}) \qquad \mathcal{P}(\tilde{R}) < \mathcal{P}(R')}{\langle \Gamma \,;\langle \tilde{u}, \tilde{R}\rangle \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle \tilde{u}, \tilde{R}\rangle \,;\langle hd(\mathrm{Q})_e, R'\rangle, \mathrm{Q}'\rangle}\ \text{NEXT}$$

$$\frac{(u,\mathsf{kill},v) \in E \qquad R' = \{r_{hd(\mathrm{Q})}\} \qquad \mathcal{P}(\tilde{R}) \geq \mathcal{P}(R')}{\langle \Gamma \,;\langle \tilde{u}, \tilde{R}\rangle \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle \tilde{u}, \tilde{R}\rangle, \mathrm{Q}\rangle}\ \text{RESUME}$$

$$\frac{q \in \mathsf{Irpt} \qquad \mathcal{P}(\{r_q\}) > \mathcal{P}(R)}{\langle \Gamma \,;\langle u, R\rangle, \mathrm{Q}\rangle \to \langle \Gamma \,;\langle u, R\rangle \,;\langle q_e, \{r_q\}\rangle, \mathrm{Q}\rangle}\ \text{IRPT}$$

Fig. 4. Stack-based semantics ($\to$).

In the following, we present a translation Φ of configurations of the queue-based semantics into configurations of the stack-based semantics:

$$\Phi : N \times (2^{\mathsf{Res}} \times \mathsf{Queue}(N \times 2^{\mathsf{Res}})) \to \mathsf{Stack}(N \times 2^{\mathsf{Res}}) \times \mathsf{Queue}(\mathsf{Task})$$

$$\Phi\langle u, \langle R, \mathrm{Q}\rangle\rangle = \langle \phi_\Gamma(\mathrm{Q}) \,;\langle u, R\rangle, \phi_Q(\mathrm{Q})\rangle$$

Note that Task only contains *task names* so it is equivalent to the set of all entry nodes of tasks, which is a small subset of N. Assume that the configuration of the queue-based semantics is given by $\langle u, \langle R, \mathrm{Q}\rangle\rangle$. Then the function ϕ_Q removes partially executed tasks from the queue Q returning a queue of fresh tasks only and is given by:

$$\phi_Q(\mathrm{Q}) = \begin{cases} [\,] & \mathrm{Q} = [\,] \\ push(q, \phi_Q(tl(\mathrm{Q}))) & hd(\mathrm{Q}) = \langle q_E, \{r_q\}\rangle \\ \phi_Q(tl(\mathrm{Q})) & \text{otherwise} \end{cases}$$

The function ϕ_Γ builds the stack of partially executed tasks from Q. The depth of this stack is at most the number of priorities below the static priority of the currently running task. The function is defined as follows:

$$\phi_\Gamma(\mathrm{Q}) = \begin{cases} [\,] & \mathrm{Q} = [\,] \\ \phi_\Gamma(tl(\mathrm{Q})) & hd(\mathrm{Q}) = \langle q_e, \{r_q\}\rangle \\ \phi_\Gamma(tl(\mathrm{Q})) \,;\langle u, R\rangle & hd(\mathrm{Q}) = \langle u, R\rangle \end{cases}$$

Note that since the introduced artificial resources are the same in both semantics and the sets of held resources are not modified the priority function $\mathcal{P}$ commutes with the translation Φ.

Theorem 1. *For any two reachable configurations $\langle u, \langle R, Q\rangle\rangle, \langle v, \langle R', Q'\rangle\rangle$, the following equivalence holds:*

$$\langle u, \langle R, Q\rangle\rangle \Rightarrow \langle v, \langle R', Q'\rangle\rangle \Longleftrightarrow \Phi\langle u, \langle R, Q\rangle\rangle \rightarrow \Phi\langle v, \langle R', Q'\rangle\rangle$$

Proof. Recall that there is a one-to-one correspondence between the rules of the queue-based and the stack-based semantics — up to the KILL-rule which is simulated by the two rules NEXT and RESUME. We perform a case distinction on the rules and prove that a rule is applicable to a configuration of the queue-based semantics if and only if the corresponding rule is applicable to the translated configuration where the results of the application are again in relation via the translation Φ. Note that we require all configurations to be reachable. i.e. proper priorities, thus we won't explicitly state it every time. Consider, e.g., the $(\Rightarrow)$-SWITCH-rule at an edge $(u, \mathsf{schedule}, v)$ and a configuration $\langle u, \langle R, \mathrm{Q}\rangle\rangle$ of the queue-based semantics. Let $\langle u', R'\rangle = hd(\mathrm{Q})$. Then the $(\Rightarrow)$-SWITCH-rule is applicable iff $\mathcal{P}(R) < \mathcal{P}(R')$, with resulting configuration $\langle u', \langle R', \mathrm{Q}'\rangle\rangle$ where $\mathrm{Q}' = push(\langle v, R\rangle, tl(\mathrm{Q}))$. On the other hand, consider the configuration $\Phi(\langle u, \langle R, \mathrm{Q}\rangle\rangle) = \langle \phi_\Gamma(\mathrm{Q}); \langle u, R\rangle, \phi_Q(\mathrm{Q})\rangle$ of the stack-based semantics. Let $R'' = \{r_{hd(\phi_Q(\mathrm{Q}))}\}$. Then the $(\rightarrow)$-SWITCH-rule is applicable iff $\mathcal{P}(R) < \mathcal{P}(R'')$. We first show that $R' = R''$. Let $\tilde{q}$ be the task within which u occurs. Then the condition $\mathcal{P}(R) < \mathcal{P}(R')$ implies that also $\mathcal{P}(r_{\tilde{q}}) < \mathcal{P}(R')$. Thus due to the PCP, $\langle u', R'\rangle = \langle q_e, \{r_q\}\rangle$ for some task $q \in \mathsf{Task}$ since a partially executed task of priority $\mathcal{P}(R')$ could not have been pre-empted by $\tilde{q}$. Since ϕ_Q preserves the relative ordering of tasks, and $\langle q_e, \{r_q\}\rangle = hd(\mathrm{Q})$, we conclude that also $q = hd(\phi_Q(Q))$.

It remains to prove that $\Phi\langle u', \langle R', \mathrm{Q}'\rangle\rangle$ equals the result of applying the $(\leftarrow)$-SWITCH-rule of the stack-based semantics to $\langle \phi_\Gamma(\mathrm{Q}); \langle u, R\rangle, \phi_Q(\mathrm{Q})\rangle$. The latter results in $\langle \phi_\Gamma(\mathrm{Q}); \langle v, R\rangle; \langle q_e, \{r_q\}\rangle, tl(\phi_Q(\mathrm{Q}))\rangle$. We have:

$$\begin{aligned}\Phi(\langle v, \langle R', \mathrm{Q}'\rangle\rangle) &= \langle \phi_\Gamma(\mathrm{Q}'); \langle u', R'\rangle, \phi_Q(\mathrm{Q}')\rangle \\ &= \langle \phi_\Gamma(push(\langle v, R\rangle, tl(\mathrm{Q})); \langle u', R'\rangle, \phi_Q(push(\langle v, R\rangle, tl(\mathrm{Q}))\rangle \\ &= \langle \phi_\Gamma(push(\langle v, R\rangle, tl(\mathrm{Q})); \langle q_e, \{r_q\}\rangle, \phi_Q(push(\langle v, R\rangle, tl(\mathrm{Q}))\rangle\end{aligned}$$

Regarding the stack, we verify that:

$$\phi_\Gamma(push(\langle v, R\rangle, tl(\mathrm{Q})) = \phi_\Gamma(\mathrm{Q})\ ; \langle v, R\rangle$$

By the same arguments used for the head, all tasks in Q of higher priority than $\mathcal{P}(R)$ are of the form $\langle q''_e, \{r_{q''}\}\rangle$. Regarding ϕ_Γ, this implies that not only $\phi_\Gamma(\mathrm{Q}) = \phi_\Gamma(tl(\mathrm{Q}))$ but also $\phi_\Gamma(push(\langle v, R\rangle, tl(\mathrm{Q}))) = \phi_\Gamma(tl(\mathrm{Q}))\ ; \langle v, R\rangle$ – which together yield the desired equality.

Finally we must show that the queues match, i.e.,

$$\phi_Q(push(\langle v, R\rangle, tl(\mathrm{Q}))) = tl(\phi_Q(\mathrm{Q}))$$

Since $\langle v, R\rangle$ is a partially executed task, $\phi_Q(push(\langle v, R\rangle, tl(\mathrm{Q}))) = \phi_Q(tl(\mathrm{Q}))$ and $\phi_Q(\mathrm{Q}) = push(q, \phi_Q(tl(\mathrm{Q}))$. By preservation of the relative orderings of tasks, we know that q is the head of the translated queue thus $tl(\phi_Q(\mathrm{Q}) = \phi_Q(tl(\mathrm{Q})$. This completes the proof for the SWITCH-rule.

The proofs for the remaining corresponding rules follow by similar arguments. Here, we only show how the two disjoint cases of the KILL-rule are simulated by means of the NEXT- and RESUME-rule, respectively. When applying the KILL-rule, either the node u' at the head of the queue is an entry node of a task or not. If $u' = q_e$ for some $q \in \mathsf{Task}$, then its set of held resources equals $\{r_q\}$. The latter implies that $q = hd(\phi_Q(\mathrm{Q}))$, and, therefor, that the priority of r_q is higher than the priority of the set of held resources of all partially executed tasks in Q. This ensures that the priority condition of the NEXT-rule is satisfied while at the same time the priority condition of the RESUME-rule is falsified. If the head of Q is a partially executed task, it becomes the top of $\phi_\Gamma(\mathrm{Q})$, and the priority of its set of held resources is higher than (or equal and older) the static priority of all tasks in Q, in particular all fresh tasks. This ensures that the priority condition of the RESUME-rule is satisfied while at the same time the priority condition of the NEXT-rule is falsified. □

The stack-based semantics can naturally be extended with potentially recursive functions. The stack then has layers corresponding to pre-empted tasks followed by called functions which have not yet returned at the moment of preemption. Theorem 1 however also holds for queue-based and stack-based semantics when extended with function calls since they do not interfere with the PCP. In this case, the type of the translation Φ is given by:

$$\Phi : \mathsf{Stack}(N \times 2^{\mathsf{Res}}) \times \mathsf{Queue}(\mathsf{Stack}(N \times 2^{\mathsf{Res}})) \to \mathsf{Stack}(N \times 2^{\mathsf{Res}}) \times \mathsf{Queue}(\mathsf{Task})$$

Since there are finitely many tasks with finite activation numbers, the set of task queues possibly occurring in the stack-based semantics is finite. Similarly, the occurring sets of held resources are finite. Hence, the rules of the stack-based semantics can be interpreted as the transfer function of a *push-down* system where the sets of push-down symbols and states are given by $N \times 2^{\mathsf{Res}}$ and $\mathsf{Queue}(\mathsf{Task})$, respectively.

This also remains true when finite abstractions of local and global data are added. Therefore, techniques for analysing push-down systems [3] can be applied. In particular, reachability is decidable. When dealing with more expressive abstractions for deriving program invariants such as, e.g., affine equalities of variables, alias analysis, etc., techniques from interprocedural analysis [4] can be applied.

4 Related Work

The context- and synchronization-sensitive analysis of concurrent programs is generally undecidable even for simple analyses [13]. Therefore, one either over-approximates the interaction between threads, as in thread-modular model-

checking [6] and nested fix-point abstract interpretation [15]; or one places restrictions on concurrency, analysing precisely up to a fixed number of context switches [1, 12] or analysing structured parallelism only [5]. We obtain a precise analysis by exploiting properties of a specific framework for interrupt-driven concurrency. There exists relatively few works considering interrupt driven programs with priorities and to our knowledge none at all using the OSEK specification.

Closest to our work is the model used by Atig et al. [2]. They show that the control point reachability problem is decidable for asynchronous programs with pre-emption consisting of tasks with priorities. Thanks to the OSEK scheduling policy and the PCP we are able to keep the task pool structured as a queue and reduce the system to push-down automata, while they use a multi-set and reduce to petri-nets. While not mentioned by the authors, we believe that interrupts can be added to their model without needing to adjust the reduction.

In a number of papers, Kahlon et al. [7–9] discus model checking of push-down systems communicating via locks and asynchronous rendezvous (wait/notify). Using acquisition histories, they show that in the case of only nested locks the problem is decidable. While their model does not use priorities, this approach might also help to improve analysis of OSEK programs since nested use of locks is a desired, but not enforced, property of OSEK programs. Lammich and Müller-Olm [10] use a restricted semantics to generate monitor-consistent inter-leavings in order to find conflicting states in a system with dynamic thread creation and re-entrant monitors. We build in this idea by including priority conditions encoding the OSEK scheduling in our semantics.

Finally, Regehr and Cooprider, [14], present a source-transformation technique to turn interrupt driven embedded code into thread-based code. Applying normal race detection tools to the transformed code yields good results. They also introduce artificial interrupt locks to make interrupt disabling/(re-)enabling visible to the analysis. We slightly modify this idea to work with the PCP.

References

1. M. F. Atig, A. Bouajjani, and S. Qadeer. Context-Bounded analysis for concurrent programs with dynamic creation of threads. In *TACAS'09*, LNCS, vol. 5505, pages 107–123. Springer, 2009.
2. M. F. Atig, A. Bouajjani, and T. Touili. Analyzing asynchronous programs with preemption. In *FSTTCS'08*. IBFI, 2008.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97*, LNCS, vol. 1243, pages 135–150. Springer, 1997.
4. P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Programs. In E. Neuhold, editor, *Formal Descriptions of Programming Concepts*, pages 237–277. North-Holland Publishing Company, 1977.
5. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *POPL'00*, pages 1–11. ACM Press, 2000.
6. C. Flanagan, S. N. Freund, S. Qadeer, and S. A. Seshia. Modular verification of multithreaded programs. *Theoretical Computer Science*, 338(1-3):153–183, 2005.

7. V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL'07*, pages 303–314. ACM Press, 2007.
8. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *CAV'05*, LNCS, vol. 3576, pages 505–518. springer, 2005.
9. V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV'07*, LNCS, vol. 4590, pages 226–239. Springer, 2007.
10. P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In *SAS'08*, pages 205–220. springer, 2008.
11. OSEK/VDX Group. *OSEK/VDX Operating System Specification, Version 2.2.3*, 2005.
12. S. Qadeer and J. Rehof. Context-Bounded model checking of concurrent software. In *TACAS'05*, LNCS, vol. 3440, pages 93–107. Springer, 2005.
13. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Trans. Prog. Lang. Syst.*, 22(2):416–430, 2000.
14. J. Regehr and N. Cooprider. Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science*, 174(9):139–150, 2007.
15. H. Seidl, V. Vene, and M. Müller-Olm. Global invariants for analyzing multi-threaded applications. *Proc. of the Estonian Academy of Sciences: Phys., Math.*, 52(4):413–436, 2003.

Programming by Equilibria

Christian Tschudin, Thomas Meyer

Computer Science Department, University of Basel
Bernoullistrasse 16, CH–4056 Basel, Switzerland
{christian.tschudin,thomas.meyer}@unibas.ch

Abstract. We present a reactive computing paradigm that seeks to solve problems by bringing the programmed system into an equilibrium state: The production of a certain (numeric) result, or of a certain side effect (routing), emerges from the system's tendency to strive for an equilibrium. We have identified important equilibria for properties like adaptivity, as well as self-healing where a program controls its own program code. By linking our execution model with well known laws from chemistry, we can predict a program's dynamic behavior and in some cases even provide elegant proofs of convergence. In this paper we recapitulate how one can compute results out of the random execution of instructions. We then introduce an artificial chemistry as our reactive programming environment, in which some networking protocols can be expressed in a natural way. We present a gossip-style protocol for the cooperative computation of an average value which is amenable to a formal stability analysis, as well as a load balancing protocol implementation that is self-healing.

Keywords: reactive programming, artificial chemistry, computer network protocols, provable program dynamics, self-healing.

1 An Introductory Example

Consider a checker board with two types of tokens, white and black. In each round, apply a simple rule: Randomly select two places and compare the color of the tokens found there. If they are identical, do nothing. If the colors are different, create a copy of each token and let them drop at random places (replacing any previously present token color). Then start the next round. We call this variation of Eigen's game [1] the "mate-and-spread game".

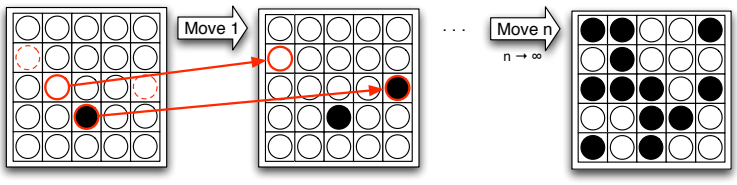


Fig. 1. The board is initialized with only one black token and 24 white tokens. After some rounds, the number of white and black tokens mutually approach each other.

If each color is presented at least once in a start configuration, the resulting system will always be the same: No matter of the initial distribution of the tokens, this game strives for an equilibrium where black and white tokens are present with the same abundance as shown in Fig. 2.

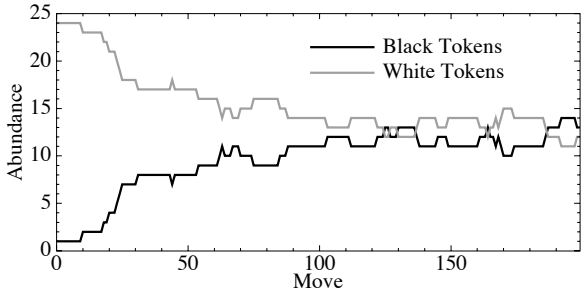


Fig. 2. In the mate-and-spread game, the amount of each token type eventually fluctuates around equality.

The microscopic process of a move in this game is random (random selection and random replacement) and we cannot predict the outcome of the next round. However, at the macroscopic level we observe the emergence of an equilibrium.

1.1 An Example of Programs in Equilibrium

We can produce the same system behavior if we turn the passive tokens into active *programs*. Instead of a central player picking tokens and replacing them if needed, tokens try to bind to (mate with) each other and, if necessary, replace themselves as well as other tokens.

Consider the two "programs" `[match x fork fork fork nop match x]` and `[x fork fork fork nop match x]` that represent our tokens from before. Their respective prefixes (see Section 2 for full details) makes them to bind with each other, producing the following intermediate program:

```
[fork fork fork nop match x fork fork fork nop match x]
```

The new program's prefix, `fork`, requests replacement by two copies of itself, with the second and third symbol being the new header symbols. Surprisingly, both resulting programs will be identical and read as follows:

```
[fork nop match x fork fork fork nop match x]
```

The next step, for each of these programs, is again the copy-yourself operation, leading to two different programs, namely `[nop x fork fork fork nop match x]` and `[match x fork fork fork nop match x]`. The `nop` prefix is simply executed (doing nothing), and we end up with the two programs we started with, except that now we have them twice.

These programs are "Quines"[1], as they regenerate, and even duplicate themselves. This system would grow unboundedly, wherefore we add the rule that new programs (created out of `fork` or `match`) replace other programs in the system randomly. Such a system will behave like the mate-and-spread game from above, always approaching an equilibrium where both types of programs are present in the same quantities. That is: even if we destroy all copies of one program type except one, the system will recover and "heal itself".

As simply as the logic of these prefix programs looks, there are still some elements to examine in more detail. Does, for a given set of initial program types, an equilibrium exist? Such proofs, especially as they relate to the dynamics of program execution, are quite hard to achieve in ordinary sequential programming. By studying our prefix programs above as a chemical reaction network, however, we can use well-known perturbation analysis tools to predict the macroscopic behavior, **iff** we carefully craft the scheduling of program execution.

1.2 Structure of this Paper

In Sect. 2, we expand a little more on the prefix programming language used in our approach and demonstrate a simple computation task for an artificial chemistry. Section 3 introduces a program scheduling that observes the chemical "law of mass action". Section 4 presents a load balancing protocol for which we prove the existence of the desired equilibrium and which, moreover, is a self-healing protocol implementation.

2 Prefix Programs (Fraglets)

We call tiny programs that bind with each other and rewrite themselves "fraglets", which stands for a small fragment of a computation. Fraglets react with other fraglets akin to molecules in chemistry. In this section, we provide a condensed description of the Fraglets language [2, 3], an artificial chemistry [4], whose corresponding chemical machine is executable and which serves as a simple platform to run "chemical programs".

Each fraglet $s \in \mathcal{S}$, is a string of symbols over a finite alphabet Σ. The first symbol of the string defines the string rewriting operation applied to this molecule by the virtual chemical machine: The prefix can be thought of an assembler instruction. The following list shows some of these instructions and their actions:

[1] after the philosopher and logician Willard van Orman Quine (1908–2000) who studied indirect self-reference

$$
\begin{aligned}
\texttt{[match}\ \alpha\ \Phi\texttt{]} + \texttt{[}\alpha\ \Omega\texttt{]} &\longrightarrow \texttt{[}\Phi\ \Omega\texttt{]} \\
\texttt{[matchp}\ \alpha\ \Phi\texttt{]} + \texttt{[}\alpha\ \Omega\texttt{]} &\longrightarrow \texttt{[matchp}\ \alpha\ \Phi\texttt{]} + \texttt{[}\Phi\ \Omega\texttt{]} \\
\texttt{[fork}\ \alpha\ \beta\ \Omega\texttt{]} &\longrightarrow \texttt{[}\alpha\ \Omega\texttt{]} + \texttt{[}\beta\ \Omega\texttt{]} \\
\texttt{[nop}\ \Omega\texttt{]} &\longrightarrow \texttt{[}\Omega\texttt{]} \\
\texttt{[send}\ k\ \Omega\texttt{]}_i &\longrightarrow \texttt{[}\Omega\texttt{]}_k \quad \text{if } i \text{ and } k \text{ are neighbor nodes}
\end{aligned}
$$

$\alpha, \beta \in \Sigma$ are arbitrary symbols, $\Phi, \Omega \in \Sigma^*$ are symbol strings and $i, k \in V$ are network nodes, or metaphorically, reaction vessels. Molecules starting with `match` or any non-instruction identifier are in their *normal form*. The `match` instruction can be used to join two molecules inside the same vessel by concatenating the second to the first after removing the processed headers. Subsequent instructions immediately reduce the product further until they again reach their normal form. For example, the two molecules `[match pkt send` n_2 `pkt]` and `[pkt data]` in node n_1 imply the following reaction:

$$
\begin{aligned}
\texttt{[match pkt send}\, n_2\, \texttt{pkt]}_{n_1} + \texttt{[pkt data]}_{n_1} & \\
\longrightarrow \texttt{[send}\, n_2\, \texttt{pkt data]}_{n_1} & \\
\text{(transmission to neighbor } n_2\text{)} \quad \longrightarrow \texttt{[pkt data]}_{n_2} &
\end{aligned}
$$

Such a chemical language allows us to "program" a reaction graph: Computation is realized by discrete string rewriting steps inside a node, but also across a network of nodes, spanning a global reaction network.

2.1 Chemically Computing an Average Value

In the following we present a chemical implementation of a gossip-style protocol in Fraglets that makes use of a chemical equilibrium in order to compute the average of values stored in distributed nodes (see [5] for other gossip protocols).

In each node of a network, and for each of its neighbors k we inject one instance of the following "shuttle service" fraglet: `[matchp X send` k `X]`. The input value to the computation is represented by the initial number of X molecules. Thus, in each node, we place as many X-molecules as the x value reads. The link-specific shuttle service fraglets[2] compete for the local X molecules in a random fashion and, when reacting, transfer one of them to the corresponding neighbor node. These fraglets build a virtual reaction network that spans over all network nodes.

In order to illustrate the principle of our protocol, we carried out an OMNeT++ [7] simulation in the network depicted in Fig. 3. We initialized node 1 with 1000 molecules of type X; all other nodes contain no X molecules at the beginning of the simulation. Figure 4 shows how the value of each node asymptotically converges to the expected average of $x_i(t \to \infty) = 250$ (see Sect. 3.2, and [8] for a formal convergence proof).

[2] For a variant of the protocol where the nodes do not require information about their neighbors, see [6]

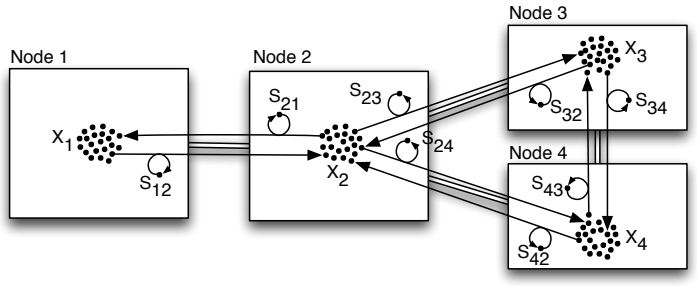


Fig. 3. The chemical protocol calculates the average concentration of X molecules in a network of virtual reaction vessels.

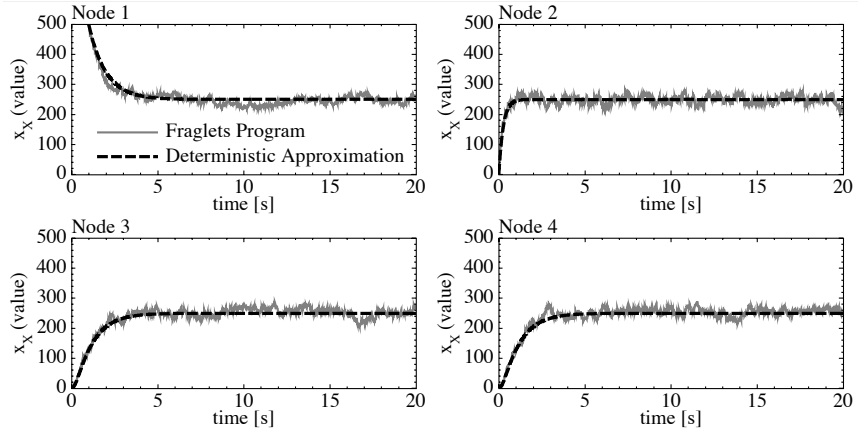


Fig. 4. Simulation of the chemical averaging protocol (Fraglets program run), and analytic prediction using a deterministic ODE model.

3 "Chemical" Scheduling and System Equilibrium

In the previous, distributed computation example, the system would not strive for an equilibrium if all network nodes would send X-molecules as fast as possible. An appropriate scheduler has not only to decide which reaction is executed next, but also *when* the reaction occurs. In this section we present a scheduling algorithm for Fraglets that helps us designing equilibrium solutions.

One scheduling algorithm that was proposed by Gibson and Bruck [9] for artificial chemistries, is based on laws discovered in statistical mechanics [10]: Molecules in a reaction vessel undergo Brownian motion, which leads to collision events. The more molecules there are in a given volume, the more frequently they collide. Consequently, in a reaction vessel of constant volume, the reaction rate is proportional to the product of the reactant's concentrations. This relation is known as the *law of mass action* [11].

The algorithm of Gibson and Bruck, which we adopted for our scheduler, calculates the next occurrence time for each reaction r individually. First it counts the combinatorial number of potential collision partners, i.e. the product of the reaction's reactant (input molecule) concentrations:

$$a_r = \prod_{i=1}^{n} x_i^{\alpha_{ir}} \tag{1}$$

where n is the number of molecule types, x_i is the concentration of molecule type i, and α_{ir} denotes the number of molecules of type i that are consumed by reaction r.

The Gibson-Bruck algorithm then draws an exponentially distributed random variable to determine the reaction interval of r based on the number of possible collision partners:

$$\tau_r \sim \mathrm{Exp}\left(\frac{1}{a_r}\right) \tag{2}$$

After reaction r has been executed at time t_{now}, its next occurrence time is computed as

$$t_{r,\mathrm{next}} = t_{\mathrm{now}} + \tau_r \tag{3}$$

In our implementation the so calculated occurrence times of all reactions are sorted into a priority queue. In a simple loop, the scheduler continuously extracts the first element from the queue, executes the corresponding reaction and computes the new reaction time according to (3). Between two reactions, the reaction vessel sleeps for a well-defined, but inherently stochastic, idle time.

3.1 Deterministic Approximation of the System Dynamics

Despite the randomness of the reactions on the microscopic level, and *because* of the forced idle time of the CPU, we can make predictions about the dynamic behavior of such an artificial chemistry. We approximate the reaction rate by only considering the mean value of the reaction interval $\langle \tau_j \rangle$:

$$v_r = \prod_{i=1}^{n} x_i^{\alpha_{ir}} \tag{4}$$

This rate reflects the macroscopic *law of mass action*. We translate the discrete set of reactions $\mathcal{R}$ into a continuous ODE (ordinary differential equation) model:

$$\dot{\boldsymbol{x}}(t) = \mathbf{N}\boldsymbol{v}(t) \tag{5}$$

where $\dot{\boldsymbol{x}}(t) = \left(\dot{x}_1(t)\ \dot{x}_2(t) \cdots\ \dot{x}_n(t)\right)^T$ is the vector of the time derivatives of all molecule concentrations, $\mathbf{N}$ is the stoichiometric matrix ($\mathbf{N} = [\gamma_{ij}]$ where γ_{ij} denotes the net number of molecules of type s_i that are generated by reaction

r_j), and $\boldsymbol{v}(t)$ is the vector of reaction rates $\boldsymbol{v}(t) = \big(v_1(t)\ v_2(t)\ \cdots\ v_m(t)\big)^T$ with v_r according to (4).

Based on such a traditional description of chemical kinetics, we are able to analyze the system's behavior and analyze properties like the development of concentrations in time, finding equilibria and analyzing their stability.

3.2 Perturbation Analysis

From the ODEs of a reaction system, there is a small step towards finding its equilibria (or fixpoints). If the chemical reaction system strives for an equilibrium, the number of molecules in equilibrium, $\boldsymbol{x}^*$, do not change anymore. Thus, in (5) we set $\dot{\boldsymbol{x}} \equiv 0$. For the averaging example in Sect. 2.1, we can show that there is a single fixpoint

$$x_i^* = \frac{\sum_{k=1}^{N} x_k^*}{N} = \langle x^* \rangle \tag{6}$$

where the value x_i in each node equals the average value (N: number of network nodes).

To prove that the fixpoint is stable when disturbed by a small perturbation, we linearize the system around the fixpoint by calculating the Jacobian matrix:

$$J = \left[\frac{\partial \dot{x}_i}{\partial x_j} \right] \tag{7}$$

The fixpoint is stable if the real part of all eigenvalues of the Jacobian, evaluated at the fixpoint, are smaller than zero. See [8] for the complete stability analysis of the averaging example.

4 A Self-Healing Protocol

In Sect. 1 we introduced a simple self-duplicating Quine in Fraglets. Such a Quine can easily be converted into one that processes some data stream while replicating itself (see [12] for more details). We now make use of such Quines as building blocks in order to assemble a self-healing protocol that balances a packet stream over two different network paths such that packet loss is minimized.

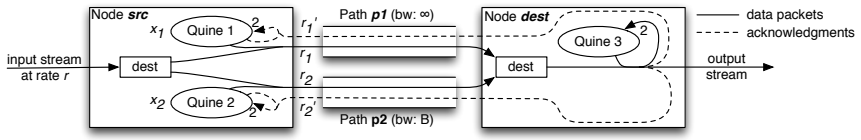


Fig. 5. In the self-healing load balancing protocol, packet streams control the replication of code.

As depicted in Fig. 5, we inject packets (=fraglets) at rate r into the source node where two quines, one for each path, compete for them and send them over

the corresponding paths. Instead of replicating as fast as possible, the quines depend on and react with acknowledgment packets. These acknowledgments are sent back over the reverse path by the third quine in the destination node, which also delivers the original packets to the data sink.

4.1 Formal Convergence Proof

This simple scheme leads to a perfect packet balance among the two paths, matching the available bandwidths. When the source node's reaction vessel is saturated, its molecules either belong to quine 1 or 2, as other molecules have been squeezed out. Let's denote the relative concentrations by x_1 and x_2, respectively, satisfying $x_1 + x_2 = 1$. Since replication is triggered by received acknowledgments, these concentrations are $x_1 = r'_1/(r'_1+r'_2)$ and $x_2 = r'_2/(r'_1+r'_2)$ where r'_n is the rate of acknowledgments received over path pn.

Let's assume that the bandwidth of *p1* is infinite whereas *p2* drops packets exceeding a rate of B packets/s. We examine the overload situation where the total rate $r > 2B$. Consequently, the rate of acknowledgments is $r'_1 = r_1$ and $r'_2 = \min(r_2, B)$. Due to the law of mass action, the fraction of packets sent over *p1* is proportional to the concentration of quine 1: $r_1 = x_1 r = r_1 r/(r_1 + \min(r_2, B))$. Hence

$$r_1 = r - B \quad \text{and} \quad r_2 = r - r_1 = B \tag{8}$$

Quine 2 reduced its concentration so as to only forward packets up to the bandwidth limitation of path *p2*, as was to be proved.

4.2 Surviving Code Attacks

Like in the computation example based on the gossip-style protocol, our load balancing protocol reaches a (chemical) equilibrium. Deviations from the equilibrium, for example by lost molecules on a network path, are compensated by increasing the population of quines that forward packets over the opposite path. Moreover, the code itself is organized in circuits of self-replicating molecules: If we forcefully destroy some code, the system will eventually regenerate it and, after some transition time, autonomously finds back to the equilibrium state. In fact, packet loss as well as *code loss* are treated by the same mechanism and in the same way.

Figure 6 depicts the protocol's response to such a code deletion attack[3], simulated in OMNeT++ for $r = 200\,\text{pkts/s}$ and $B = 40\,\text{pkts/s}$ and the network topology shown in Fig. 5. At $t = 50\,\text{s}$ we remove 80 % of all molecules from the source vessel: The code regenerates itself within 10 s while the traffic distribution r_1/r_2 remains unchanged.

[3] Code mutations can also be caught by mapping them onto code deletion events using a simple instruction encoding guarded by a parity-bit.

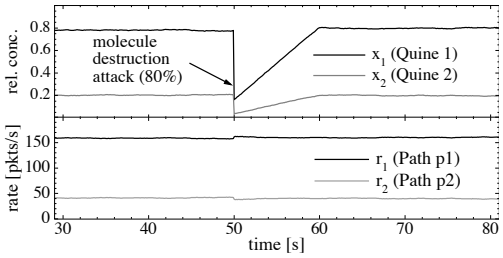


Fig. 6. Simulation of self-healing load balancing quines: Relative concentration of forwarding code in node *src* during a deletion attack, and the corresponding packet forwarding rate (unaffected) of the quines.

5 Conclusions

Representing a solution to a computation as an equilibrium of a dynamic system, lets the system "automatically" strive for this solution, starting from an unbalanced initial configuration (=input value) and tolerating perturbations during the execution.

We have demonstrated in this paper such an approach by showing a prefix programming language that permits to define artificial chemical reaction networks. Imposing a special scheduling of discrete instructions that is compatible with the "law of mass action", we obtain systems that can be described by continuous functions and whose dynamics is amenable for a formal perturbation analysis.

Finally, by abandoning the strict distinction between code and data, one can create reaction networks which regulate their own "code replication" as for example produced by Quine programs. This permits to write programs that provably strive for code equilibrium, which is another way of saying that a program has become self-healing.

Acknowledgments

This work has been partially supported by the Swiss National Science Foundation and the European Union, through SNP Project Self-Healing Protocols and FET Project BIONETS, respectively.

References

1. Eigen, Manfred und Winkler, R.: Das Spiel. Piper (1975)
2. Fraglets home page `http://www.fraglets.net`.
3. Tschudin, C.: Fraglets - a metabolistic execution model for communication protocols. In: Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS). (2003)
4. Dittrich, P., Ziegler, J., Banzhaf, W.: Artificial chemistries - a review. Artificial Life **7**(3) (2001) 225–275

5. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: Proc. 44th Annual IEEE Symposium on Foundations of Computer Science. (2003) 482–491
6. Meyer, T., Yamamoto, L., Tschudin, C.: An artificial chemistry for networking. Volume 5151 of Lecture Notes in Computer Science. Springer, Berlin / Heidelberg (2008) 45–57
7. Omnet++ homepage `http://www.omnetpp.org`.
8. Meyer, T., Tschudin, C.: Chemical networking protocols. In: Proc. 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII). (2009)
9. Gibson, M.A., Bruck, J.: Efficient exact stochastic simulation of chemical systems with many species and many channels. Journal of Physical Chemistry A **104**(9) (2000) 1876–1889
10. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry **81**(25) (1977) 2340–2361
11. Abrash, H.I.: Studies concerning affinity. Journal of Chemical Education **63** (1986) 1044–1047
12. Meyer, T., Schreckling, D., Tschudin, C., Yamamoto, L.: Robustness to code and data deletion in autocatalytic quines. Volume 5410 of Lecture Notes in Bioinformatics. Springer (2008) 20–40

The Reachability-Bound Problem

Florian Zuleger
Technische Universität Darmstadt

We define the reachability-bound problem to be the problem of finding a symbolic worst-case bound on the number of times a given control location inside a procedure is visited in terms of the inputs to that procedure. This has applications in bounding resources consumed by a program such as time, memory, network- traffic, power, as well as estimating quantitative properties (as opposed to boolean properties) of data in programs, such as amount of information leakage or uncertainty propagation.

Our approach to solving the reachability-bound problem brings together two very different techniques for reasoning about loops in an effective manner. One of these techniques is an abstract- interpretation based iterative technique for computing precise disjunctive invariants (to summarize nested loops). The other technique is a non-iterative proof-rules based technique (for loop bound computation) that takes over the role of doing inductive reasoning, while deriving its power from use of SMT solvers to reason about abstract loop-free fragments.

Autorenverzeichnis