Letter from the President

Dear EATCS members,

As usual this time of the year, I have the great pleasure to announce the assignments of this year's Gódel Prize, EATCS Award and Presburger Award.

The Gödel Prize 2012, which is co-sponsored by EATCS and ACM SIGACT, has been awarded jointly to Elias Koutsoupias, Christos H. Papadimitriou, Tim Roughgarden, Éva Tardos, Noam Nisan and Amir Ronen. In particular, the prize has been awarded to Elias Koutsoupias and Christos H. Papadimitriou for their paper Worst-case equilibria, *Computer Science Review, 3(2): 65-69, 2009; to Tim Roughgarden and Éva Tardos for their paper* How Bad Is Selfish Routing?, *Journal of the ACM, 49(2): 236-259, 2002; and to Noam Nisan and Amir Ronen for their paper* Algorithmic Mechanism Design, *Games and Economic Behavior, 35: 166-196, 2001. As you can read in the laudation published in this issue of the bulletin, these three papers contributed highly influential concepts and results that laid the foundation for an explosive growth in algorithmic game theory, a trans-disciplinary combination of the theory of algorithms and the theory of games that has greatly enriched both fields. The purpose of all three papers was to improve our understanding of how the internet and other complex computational systems behave when users and service providers in these systems act selfishly. On behalf of this year's Gödel Prize Committee (consisting of Sanjeev Arora, Josep Díaz, Giuseppe F. Italiano, Daniel*

Spielman, Eli Upfal and Mogens Nielsen as chair) and the whole EATCS community I would like to offer our congratulations and deep respect to all of the six winners!

The EATCS Award 2012 has been granted to Moshe Vardi for his decisive influence on the development of theoretical computer science, for his pre-eminent career as a distinguished researcher, and for his role as a most illustrious leader and disseminator. The laudation, also published in this issue of the bulletin, illustrates his distinguished scientific career. Moshe Vardi has made fundamental and lasting research contributions to the development of mathematical logic as a unifying foundational framework for modeling computational systems. His research has focused on applying and developing logic in computing, and has played a major role in our present understanding and use of logic in computing. Vardi has contributed to several areas of theoretical computer science: in particular, software and hardware verification, databases, complexity theory and distributed systems. The proposal has been made by our selection committee consisting of Leslie Ann Goldberg, Friedhelm Meyer auf der Heide and Eugenio Moggi (chair), and it has been unanimously approved by the EATCS Council members. On behalf of the whole EATCS community I would like to offer our congratulations to Moshe for this well-deserved award!

The Presburger Award Committee 2012, consisting of Monika Henzinger, Antonin Kucera and Stefano Leonardi (chair) has

unanimously decided to propose Venkatesan Guruswami and Mihai Patrascu as joint recipients of the 2012 EATCS Presburger Award for young scientists. This excellent proposal has been approved by the EATCS Council members. The laudation, also published in this issue of the bulletin, illustrates that these two outstanding young scientists, both working in different fields at different stages of their career, are two of the most impressive researchers in theoretical computer science of their generation and ideal recipients for the Presburger Award. Venkatesan Guruswami has contributed cornerstone results to the theory of list decoding of error correcting codes. His work culminated in a publication that settles one of the most salient theoretical open problems in the theory of communication since Shannon's invention of error correcting codes in 1949. Mihai Patrascu has contributed fundamental results on lower bounds for data structures. His work has broken through many old barriers on fundamental data structure problems, not only revitalizing but also revolutionizing a field that was almost silent for over a decade. All our congratulations go out to Venkatesan and Mihai!

Please note that all three prizes will be presented in a ceremony that will take place during ICALP 2012 in Warwick. On behalf of the EATCS, I would like to offer our sincere thanks to all members of the Award Committees for their work and excellent choices.

The program of ICALP 2012 is now ready and can be viewed in detail on the corresponding website http://www2.warwick.ac.uk/fac/cross_fac/dimap/icalp2012. Once again it is an excellent program, due to both the high quality of the contributed papers and the large number of scientific events, like the Turing Talk *given by David Harel, the presentation of the Awards mentioned above and the five invited talks that will be given this year by Gilles Dowek, Kohei Honda, Stefano Leonardi, Daniel A. Spielman and Berthold Vöcking. Moreover, on Wednesday afternoon there will be a* Turing Excursion *combined with a visit of Bletchley Park which promises to be another memorable highlight. We are all convinced that the ICALP conference chaired by Kurt Mehlhorn (track A), Andrew Pitts (track B) and Roger Wattenhofer (track C) will again be a great success. The conference will be preceded by three satellite workshops: APAC (Workshop on Applications of Parameterized Algorithms and Complexity), CL&C (Fourth International Workshop on Classical Logic and Computation), and WRAWN (Third Workshop on Realistic models for Algorithms in Wireless Networks). The Conference Chair, Artur Czumaj, together with his team is doing an excellent job and we are looking forward to meeting you in Warwick. So, hurry up and register now for this extraordinary event!*

In Warwick, you will have also the opportunity to see the first Call for Papers for ICALP 2013 with the indication of the program committees and the invited speakers. In this context, I would like to take this opportunity to invite you to attend the EATCS General Assembly which

will take place on Tuesday evening during the ICALP week in Warwick and where you will be informed in detail about the EATCS activities within the last year and the plans for the next year.

Burkhard Monien, Paderborn
June 2012

Letter from the Bulletin Editor

Dear Reader,

Welcome to the June 2012 issue of the Bulletin of the EATCS.

I have some news to report. Starting from next issue, Edgar Chávez will be in charge of informing us on the novelties from Latin America. Chapter of the EATCS. I want to thank Edgar Chávez for accepting to take care of this section and express my gratitude to Alfredo Viola for their great contribution and for the effort devoted to the Bulletin for so many years. My warmest thanks to you.

Regarding the present issue, apart from the interesting Columns contributions, our reporters bring you news on EATCS and TCS activities all over the world. Please, read them carefully!

I trust that you will find a lot of interesting material to read in this issue.

María Serna, Barcelona
June 2012

The EATCS Award2012

Laudatio for Moshe Vardi

Moshe Vardi is an extremely active and productive researcher. His work so far has been very influential, certainly in science, but also in dissemination and policy matters. We are pleased to recognise these contributions with an EATCS Award.

Research Contributions

Vardi has made fundamental and lasting research contributions to the development of mathematical logic as a unifying foundational framework for modeling computational systems. His research has focused on applying and developing logic in computing, and has played a major role in our present understanding and use of logic in computing. Vardi has contributed to several areas of Computer Science, in particular: software and hardware verification; databases; complexity theory; and distributed systems.

Automata-theoretic approach to design verification. Vardi has demonstrated that questions about correctness of hardware and software designs can be reduced to algorithmic questions about finite automata on infinitary input structures (infinite words or infinite trees). Carrying out this approach required advances in both automata theory and the theory of program logics. This connection brought a wealth of new techniques to the theory of program logics, and the new application revived the theory of automata on infinitary inputs.

His 1986 paper, "An automata-theoretic approach to automatic program verification", has over 1350 citations, it won an IBM Outstanding Innovation Award in 1989, an ACM-EATCS Goedel Prize in 2000, a LICS Test-of-Time Award in 2006, and an ACM Kanellakis Award for Theory and Practice in 2006.

This line of work is the basis of several academic and industrial automated verification tools, such as the model checker SPIN that won the 2001 ACM's

Software System award. The work has also influenced emerging specification languages, such as the IEEE Standard Property Specification Language (PSL) for writing requirements of hardware designs, and the manner in which specifications are checked by industrial model checkers.

Theory of database queries. Database management systems have evolved from unsophisticated databases, which are essentially structured collections of data, toward *smart* databases possessing deductive capabilities.

Vardi has investigated the theory of database queries, with a focus on the trade-off between expressiveness and computational complexity. His research laid the foundations in the following areas: integrity constraints, complexity of query evaluation, querying incomplete information, database updates, universal-relation interfaces, and database logic programming.

In his 1982 paper "The complexity of relational query languages", Vardi showed that there are two fundamentally different ways to measure the complexity of queries, referred to as *data complexity* and *expression complexity*, a classification that is today widely accepted. This paper has over 1150 citations.

His 1998 paper, "Conjunctive-query containment and constraint satisfaction", which exhibited a deep connection between conjunctive-query evaluation and constraint-satisfaction solving, received an ACM PODS Mendelzon Test-of-Time Award in 2008. In 2008 Vardi received the ACM Edgar F. Codd Innovations Award, the top recognition for database research accomplishments.

Descriptive complexity theory. Computational Complexity focuses on classifying computational problems according to their inherent difficulty in terms of resource (such as time or space) requirements.

Descriptive complexity theory is a branch of computational complexity theory that characterizes complexity classes by the type of logic needed to express them. An example is a classical result, proved by Vardi (and independently by Immerman) in 1982, that characterizes the complexity class PTIME is terms of first-order logic enriched with the fixpoint operator. (This work won an IBM Outstanding Innovation Award in 1992.)

His 1993 paper, "Monotone monadic SNP and constraint satisfaction", used logic, graph theory, and algebra, to study the computational complexity of constraint-satisfaction problems. The 1998 paper, "The computational structure of monotone monadic SNP and constraint satisfaction: a study through Datalog and group theory", has more than 500 citations. Together, these papers form the basis for work on the complexity of constraint satisfaction.

Knowledge in multi-agent systems. Reasoning about knowledge has applications in such diverse fields as economics, linguistics, artificial intelligence and computer science. In a distributed system a process may need to know whether other processes know that a message has been lost.

Together with his collaborators, Vardi developed an extensive theory of reasoning about knowledge. This work focuses on using reasoning about knowledge to design, analyze and verify the correctness of multi-agent systems. The work provides good formal models of knowledge that are appropriate for multiple applications. This work won an IBM Outstanding Innovation Award in 1987. The book entitled "Reasoning about Knowledge" by Fagin, Moses, Halpern and Vardi, first published by MIT Press in 1995, is now considered a classic, with over 2900 citations.

In additional to fundamental contributions, Vardi also shaped the field by pointing out promising research directions.

Scientific Dissemination and Impact

Vardi's scientific output is impressive and very influential.

- He is author or co-author of over **400 publications**, including **2 books**: "Reasoning about Knowledge" and "Finite Model Theory and its Applications".
- He has more than **20000 citations** and an **h-index above 75**.
- He has a huge number of collaborations, e.g. on publications he has almost **200 co-authors**.

Leadership and Service

Vardi has a long service record and a strong leadership in the field, in particular:

- He has played a central role in the IEEE Symposium on Logic in Computer Science (**LICS**), for which he served as General Chair for several years, and he has been the major force behind the creation of the Federated Logic Conferences (**FLoC**).
- As Editor-in-Chief, he has performed admirably in rethinking and leading **Communications of the ACM**. This service to the whole community is commendable in its own right, but his vision led to important changes, like the introduction of the excellent new section on research highlights.

Conclusions. For all the above reasons, the EATCS awards Committee unanimously decided to give the EATCS award to Professor Moshe Vardi.

The EATCS awards Committee 2012
Leslie Ann Goldberg
Friedhelm Meyer auf der Heide
Eugenio Moggi (chair)

T G¨ P 2012

L

E K , C H. P , T R ,
É T , N N A R

The Gödel Prize 2012 for outstanding papers in Theoretical Computer Science is awarded jointly to the following three papers:

E K C H. P
Worst-case equilibria
Computer Science Review, 3(2): 65-69, 2009.

T R É T
How Bad Is Selfish Routing?
Journal of the ACM, 49(2): 236-259, 2002.

N N A R
Algorithmic Mechanism Design
Games and Economic Behavior 35: 166-196, 2001.

These three papers, appearing in their conference versions around the turn of the millennium, contributed highly influential concepts and results that laid the foundation for an explosive growth in algorithmic game theory, a transdisciplinary combination of the theory of algorithms and the theory of games that has greatly enriched both fields.

All three papers aimed to improve our understanding of the behavior of the internet and other complex computational systems, when users and service providers in those systems act selfishly. The paper by Koutsoupias and Papadimitriou introduced what is today known as the *price of anarchy*, the first quantitative measure

of the degree of inefficiency of equilibria in games, formally defined as the worst-case ratio between the social cost of a Nash equilibrium and the optimal social cost. The paper by Roughgarden and Tardos revealed the power and depth of the *price of anarchy* concept, providing striking and remarkably complete results on the relationship between centralized optimization and *selfish routing* in network traffic. Finally, the paper by Nisan and Ronen introduced a whole new range of applications of the theory of mechanism design within computer science, in the process also transforming and significantly enriching the theory of mechanism design by introducing algorithmic resource bounds as well as notions of approximate optimality.

Sanjeev Arora, Princeton
Josep Díaz, Universitat Politecnica de Catalunya
Giuseppe Italiano, UniversitÃă di Roma Tor Vergata
Mogens Nielsen, Aarhus University
Daniel Spielman, Yale University
Eli Upfal, Brown University (chair)

T P A 2012

L

V G M P

The Presburger Award Committee 2012, consisting of Monika Henzinger, Antonin Kucera, and Stefano Leonardi (chair) has unanimously decided to propose

V G (Carnegie Mellon University, Pittsburgh) and
M P (AT&T Labs, New York)

as joint recipients of the 2012 EATCS Presburger Award for young scientists.

Venkatesan Guruswami, born in 1976, has contributed cornerstone results to the theory of list decoding of error correcting codes. His work culminated in a publication with his former student Artri Rudra that gave constructions of error-correcting codes with a list-decoding algorithm that achieve minimum possible redundancy.

It settles one of the most salient theoretical open problem in the theory of communication since Shannon's invention of error correcting codes in 1949.

The first major result of Venkatesan Guruswami in the field was given in his PhD dissertation where together with his advisor Madhu Sudan he showed how to list-decode Reed-Solomon codes beyond the traditional bound (1-R)/2. On the way to his more recent results, he developed, jointly with Piotr Indyk, new constructions of error correcting codes that allowed list decoding in nearly linear time. All these results were possible by establishing novel unexpected deep connections between computational complexity, coding theory, randomness and computation. Venkatesan Guruswami also contributed fundamental results to the theory of inapproximability and probabilistically checkable proofs.

Mihai Patrascu, born in 1982, has contributed fundamental results on lower bounds for data structures.

Despite his very young age, Mihai Patrascu's work has broken through many old barriers on fundamental data structure problems, not only revitalizing but also revolutionizing a field that was almost silent for over a decade.

The first cornerstone result was the logarithmic lower bound for dynamic search trees in the cell probe model published at SODA 2004 together with his advisor Erik Demaine. This result was followed at STOC 2004 with the logarithmic lower bound for dynamic trees, thus matching the upper bound of Sleator and Tarjan from 1983.

(Mihai Patrascu is the non-alphabetic first author in both papers.)

Key to these and later progresses was the deep information-theoretic understanding of computation developed in Mihai's work, which views the source of hardness in many problems as the same fundamental barrier in the representation and movement of information. In his work at FOCS 2008, Mihai Patrascu gave a simple common proof for many of his previous cell-probe lower bounds, connecting problems ranging from computational geometry to graph algorithms, by reduction from a well-identified hard computational core.

The committee has recommended the EATCS Council to share the 2012 Presburger Award between Venkatasan Guruswami and Mihai Patrascu. These two outstanding young scientists, both working in different fields at a different stage of their career, fully deserve the award. This decision has been approved by the EATCS Council.

Key to these and later progresses was the deep information-theoretic understanding of computation developed in Mihai's work, which views the source of hardness in many problems as the same fundamental barrier in the representation and movement of information. In his work at FOCS 2008, Mihai Patrascu gave a simple common proof for many of his previous cell-probe lower bounds, connecting problems ranging from computational geometry to graph algorithms, by reduction from a well-identified hard computational core.

The committee has recommended the EATCS Council to share the 2012 Presburger Award between Venkatasan Guruswami and Mihai Patrascu. These two outstanding young scientists, both working in different fields at a different stage of their career, fully deserve the award. This decision has been approved by the EATCS Council.

Report from the Japanese Chapter

R. Uehara (JAIST)

EATCS-JP/LA Workshop on TCS and Presentation Awards

The tenth *EATCS/LA Workshop on Theoretical Computer Science* was held at Research Institute of Mathematical Sciences, Kyoto University, January 30 to February 1, 2012. **Dr. Akio Fujiyoshi** (Ibaraki University) who presented the following paper, was selected at the tenth EATCS/LA Presentation Award.

> *Minimum connected spanning subgraph problem with label selection and its application to chemical structural OCR* by Akio Fujiyoshi (Ibaraki University) and Masakazu Suzuki (Kyushu University).

The award will be given him at the Summer LA Symposium held in July 2012.

>From this time, we establish another presentation award, named "EATCS/LA Student Presentation Award" to encourage students. **Mr. Tamotsu Kobayashi** (Saitama University) who presented the following paper, was selected at the first EATCS/LA Student Presentation Award.

> *Minimum Enclosing Rectangle with Fixed Aspect Ratio* by Tamotsu Kobayashi and Takashi Horiyama (Saitama University).

The award has been given him at the last day, February 1, 2012.

Congratulations!

This workshop is jointly organized with *LA*, Japanese association of theoretical computer scientists. Its purpose is to give a place for discussing topics on all aspects of theoretical computer science. That is, this workshop is an unrefereed meeting. All submissions are accepted for the presentation. There should be no problem of presenting these papers in refereed conferences and/or journals. We hold it twice a year (January/February, and July/August). If you have a chance, I recommend you to attend it. You can find the program of the last workshop in Appendix of this report.

New Web page of EATCS Japan Chapter

The web page of EATCS Japan Chapter was on `http://www.misojiro.t.u-tokyo.ac.jp/EATCS-J/index.html`, but it has been moved to `http://www.jaist.ac.jp/~uehara/EATCS-J/`. Please update your links.

Appendix: Program of EATCS-JP/LA workshop on TCS (January 30 to February 1, 2012)

In the following program, each [Sx] means student talk, while [x] means ordinary talk (student talks are shorter). Each "**" indicates a student speaker, and "*" indicates just a speaker. Talks are given in the following order:

[1] On a Faster Algorithm for Counting Perfect Matchings
**Taisuke Izumi, Tadashi Wadayama (Nagoya Institute of Technology)*

[2] Formula Decomposition into Ternary Majorities
**Kenya Ueno (Kyoto University)*

[3] A Satisfiability Algorithm for Formulas over the Full Binary Basis
*Kazuhisa Seto, *Suguru Tamaki (Kyoto University)*

[4] Enumerating Separating Families of Bipartitions
***Takahisa Toda (Kyoto University), Ivo Vigan (The City University of New York)*

[5] Hybrid-Automata-Theoretic Verification of CPU-DRP Reconfigurable Systems
***Ryo Yanase, gaoying, Shota Minami, Satoshi Yamane (Kanazawa University)*

[6] Formal Verification of Probabilistic Timed System Based on an Abstraction Refinement
***Takaya Shimizu, Atsushi Morimoto, Satoshi Yamane (Kanazawa University)*

[7] On limit cycle of composited cellular automata
**Toshikazu Ishida (kyushu sangyo university), Shuichi Inokuchi (Kyushu University)*

[8] Classification of spherical tilings by congruent quadrangles
**Yohji Akama (Mathematical Institute, Tohoku University), Yudai Sakano (JICA), Kosuke Nakamura (Department of Chemistry, Tohoku University)*

[S1] Heuristic Algorithms for Rectilinear Block Packing
***Yannan Hu, Hideki Hashimoto, Shinji Imahori, Mutsunori Yagiura (Nagoya University)*

[S2] Heuristic algorithm for a vertex pricing problem
***Sakaki Nakamura, Akiyoshi Shioura (Graduate School of Information Sciences, Tohoku University)*

[S3] A Randomized Algorithm for Finding Frequent Elements in Streams Using $O(\log \log N)$ Space
***Masatora Ogata (Kyushu University), Yukiko Yamauchi (Kyushu University), Shuji Kijima, Masafumi Yamashita (Kyushu University)*

[S4] An algorithm for the Hamiltonian circuit problem on bipartite distance-hereditary graphs
***Masahide Takasuka, Tomio Hirata (Nagoya University)*

[9] Earley's parsing algirithm and Petri net controlled grammars
**Tashin Nishida (Department of Information Sciences, Toyama Prefectural University)*

[10] The relationship between language classes in terms of insertion and locality
**Kaoru Fujioka (Kyushu University)*

[11] A Polynomial-Time Algorithm for Checking the Equivalence of Deterministic Restricted One-Counter Transducers Which Accept by Final State
**Mitsuo Wakatsuki (The University of Electro-Communications), Kazushi Seino (Toshiba Solutions Corporation), Etsuji Tomita, Tetsuro Nishino (The University of Electro-Communications)*

[12] Efficient Reduction of Square Factors in Strings
Peter Leupold (Universitat Rovira i Virgili)

[13] Decremental construction of seaweed bijections
Yoshifumi Sakai (Tohoku University)

[14] Minimum Enclosing Rectangle with Fixed Aspect Ratio
Tamotsu Kobayashi, Takashi Horiyama (Saitama University)

[15] Complexity of Smooth Ordinary Differential Equations
Hiroyuki OTA (University of Tokyo), Akitoshi Kawamura (University of Tokyo), Martin Ziegler, Carsten Rösnick (Technische Universität Darmstadt)

[16] On QMA Protocols with Two Short Quantum Proofs
Francois Le Gall (The University of Tokyo), Shota Nakagawa, Harumichi Nishimura (Osaka Prefecture University)

[17] Limiting Negations in Probabilistic Circuits
Hiroki Morizumi (Shimane University)

[18] Greedy Algorithms for Multi-Queue Buffer Management Policies with Class Segregation
Toshiya Itoh, Seiji Yoshimoto (Tokyo Institute of Technology)

[19] Memory Efficient Path Finding Algorithm
Tatsuya Imai (Tokyo Institute of Technology)

[20] An Efficient Algorithm for General-purpose Computation on GPU
Hidetoki Tanaka, Osamu Watanabe (Tokyo Institute of Technology)

[21] Probabilistic stabilization under probabilistic schedulers
Yukiko Yamauchi (Kyushu University), Sébastien Tixeuil (Paris 6), Masafumi Yamashita (Kyushu University)

[S5] Hierarchy of reversible logic elements with memory
Yuuta Mukai, Kenichi Morita (Hiroshima University, Graduate School of Engineering)

[S6] A Study of Selection Method of Separate Points Set for mm-GNAT
Pingfang Xie (Graduate School of Sciences, Tokai University), Kensuke Onishi (School of Sciences, Tokai University)

[S7] Automata inspired by biochemical reaction
Fumiya Okubo (Waseda University), Satoshi Kobayashi (University of Electro-Communications), Takashi Yokomori (Waseda University)

[S8] Complexity and winning ways of trick taking games
Kenichiro Nakai, Yasuhiko Takenaga (Department of Communication Engineering and Informatics, Graduate School of Informatics and Engineering, The University of Electro-Communications)

[S9] On the compressibility of concatenated sequence
Toshihiko Yusa (Tokyo Institute of Technology)

[S10] A graph class of unit disk graphs with chain-like structure
Hayashi Takashi, Kino Toru, Kuwabara Yuto, Nagasawa Ryosuke, Shibata Yuka, Yamazaki Koichi (Gunma University Department of Computer Science)

[22] The Matroid Intersection Problem with Priority Constraints
Naoyuki Kamiyama (Kyushu University)

[23] Minimum connected spanning subgraph problem with label selection and its application to chemical structural OCR
Akio Fujiyoshi (Ibaraki University), Masakazu Suzuki (Kyushu University)

[24] Row/column operation of tables with the octgrid model
*Shinji Koka (Nihon University), *Takaaki Goto (UEC), Kensei Tsuchida (Toyo University), Tetsuro Nishino (UEC), Takeo Yaku (Nihon University)*

[25] On the base-line location problem for the maximum weight region decomposable into base-monotone shapes
*Takashi Horiyama (Saitama University), Takehiro Ito (Tohoku University), Hirotaka Ono (Kyushu University), *Yota Otachi (Tohoku University), Ryuhei Uehara (Japan Advanced Institute of Science and Technology), Takeaki Uno (National Institute of Informatics)*

[26] Cover time of multiplex random walks on random graphs
***Yusuke Hosaka, Yukiko Yamauchi, Shuji Kijima (Department of Informatics Kyushu University), Hirotaka Ono (Department of Economic Engineering Kyushu University), Masafumi Yamashita (Department of Informatics Kyushu University)*

[27] Approximating Steiner Tree and Tree Cover in Directed Graphs
***Hibi Tomoya, Fujito Toshihiro (Toyohashi University of Technology)*

[28] On the Unit-length Embedding of Graphs on a Square Grid
***Kenji Takada, Kazuyuki Amano (Gunma University)*

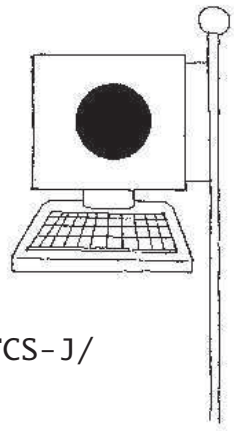
[S11] Pattern Formation by Asynchronous Mobile Robots
***Nao Fujinaga, Yukiko Yamauchi, Shuji Kijima, Masafumi Yamashita (Graduate School of ISEE, Kyushu University)*

[S12] Hiding an Image into Different Images
***Yuko Moriyama, Tomomi Matsui (Chuo University)*

[S13] Optimization Model for Mosaic Art Construction
***Ryo Nakatsubo, Tomomi Matsui (Chuo University)*

[S14] Efficient Sampling Method for Monte Carlo Tree Search Problem
***Kazuki Teraoka, Kohei Hatano, Eiji Takimoto, Masayuki Takeda (Kyushu University)*

The Japanese Chapter



Chair:	Osamu Watanabe
V.Chair:	Kazuhisa Makino
Secretary:	Ryuhei Uehara
email:	eatcs-jp@is.titech.ac.jp
URL:	http://www.jaist.ac.jp/~uehara/EATCS-J/

News from Latin America

by

Alfredo Viola

Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Casilla de Correo 16120, Distrito 6, Montevideo, Uruguay
viola@fing.edu.uy

In this issue I present the report of LATIN 2012 by Igor Shparlinski, and the announcement of SPIRE 2012. At the end I present a list of events in Theoretical Computer Science to be held in Latin America in the following months.

This is my last column. I thank very much to all the collaborators who have given me most of the information needed to write this column. I also thank very much to all the researchers who have written reports of several important scientific events held in our region. The next editor is Edgar Chávez.

Report of LATIN 2012 (by Igor Shparlinski)

LATIN 2012 was held during 16-20 March in Arequipa, a very nice and friendly town about 1000 km South-East of Lima. The conference location, Universidad Católica San Pablo, was a 10 minutes walk from the town center which allowed to interleave conference attendance with occasional escapes to town with all it could offer in historical, cultural and gastronomical aspects.

The program was very smoothly run by the PC Chair David Fernández-Baca and local organizers. Certainly regular lubrication with Pisco Sour was a contributing factor as well.

LATIN 2012 received 153 submissions from authors from 42 different countries. Out of these, 55 papers (i.e. 36%) were accepted after very carefully reviewing (by at least 3, typically by 4, and in one case by 7, referees). The accepted

papers were published by Springer in LNCS, vol. 7256. Overall, the program was well balanced, although I personally would like to see more papers addressing algebraic and number theoretic problems and algorithms.

An underlying motif behind many talks of this conference was dedication to the memory of Philippe Flajolet, who passed away 13 months before the conference, but continues to influence and motivate many different generations of researchers in discrete mathematics and theoretical computer science.

This LATIN 2012 also established a new award, namely, the Imre Simon Test-of-Time award (to be given to the most influential LATIN paper published at least ten years prior to the current conference). The selection committee (Ian Munro (chair), Sergio Rajsbaum, and Yoshiharu Kohayakawa). gave this award to a LATIN 2000 paper by Michael A. Bender and Martin Farach-Colton: "The LCA Problem Revisited".

LATIN 2012 also took part in the world-wide commemoration of the Alan Turing Year. The core of the celebration were the invited talks by Scott Aaronson and Martin Davis. Besides these talks, there were four more invited talks by Luc Devroye, Marcos Kiwi, Kirk Pruhs and Dana Randall, all of exceptionally high quality.

SPIRE 2012

SPIRE 2012 is the 19th edition of the International Symposium on String Processing and Information Retrieval. It will be held in Cartagena de Indias, Colombia from 21-25 October 2012. The conference will be organized by the Information Technology Research Group of the Universidad Autónoma de Bucaramanga.

The scope of the SPIRE series of symposia includes not only fundamental algorithms in string processing and information retrieval, but also SP and IR techniques as applied to areas such as computational biology, DNA sequencing, and Web mining. Given its interdisciplinary nature, SPIRE offers a unique opportunity for researchers from these different areas to meet and network.

For more information you can visit `http://catic.unab.edu.co/spire/`.

Regional Events

- October 7 - 10, 2012, Santiago, Chile: Second International Conference on Cryptology and Information Security in Latin America (LATINCRYPT 2012). `http://2012.latincrypt.org/`.

- October 21 - 25, 2012, Cartagena, Colombia: International Symposium on String Processing and Information Retrieval (SPIRE 2012).

  `http://catic.unab.edu.co/spire/`.

News from New Zealand

BY

C. S. Calude

Department of Computer Science, University of Auckland
Auckland, New Zealand
cristian@cs.auckland.ac.nz

1 Scientific and Community News

0. The latest CDMTCS research reports are (http://www.cs.auckland.ac.nz/staff-cgi-bin/mjd/secondcgi.pl):

417. R. Nicolescu and H. Wu. New Solutions for Disjoint Paths in P Systems. 03/2012

418. J. Hertel. Inductive Complexity of Goodstein's Theorem. 04/2012

419. L. Staiger. A Correspondence Principle for Exact Constructive Dimension. 04/2012

420. M. McKubre-Jordens and R. Sainudiin (eds.). Construmath South 2012. 04/2012

2 A Dialogue with Yuri Gurevich about Mathematics, Computer Science and Life

Yuri Gurevich is well-known to the readers of this Bulletin. He is a Principal Researcher at Microsoft Research, where he founded a group on Foundations of Software Engineering, and a Professor Emeritus at the University of Michigan. His name is most closely associated with abstract state machines but he is known also for his work in logic, complexity theory and software engineering. The Gurevich-Harrington Forgetful Determinacy Theorem is a classical result in game theory. Yuri Gurevich is an ACM Fellow, a Guggenheim Fellow, and a member of Academia Europaea; he obtained honorary doctorates from Hasselt University in Belgium and Ural State University in Russia.

Cristian Calude: Your background is in mathematics: MSc (1962), PhD (under P. G. Kontorovich, 1964) and Dr of Math (a post-PhD degree in Russia), all at Ural State University. Please reminisce about those years.

Yuri Gurevich: I grew up in Chelyabinsk, an industrial city in the Urals, Russia, and was in the first generation of my family to get systematic education. In 1957, after ten boring years in elementary + middle + high school, I enrolled in the local Polytechnic. I enjoyed student life, but I couldn't draw well, and I hated memorizing things. In the middle of the second year, one math prof advised me to transfer — and wrote a recommendation letter — to the Math Dept of the Ural State University in Ekaterinburg (called Sverdlovsk at the time), about 200 km to the north of Chelyabinsk. One of the Math Dept profs there examined me, and I joined the class of 1962, on the condition that I pass all the math exams taken during the last 1.5 years by my new classmates.

The Math Dept, formally the Dept of Mathematics and Mechanics, was demanding. Typically only a quarter of a class graduated after the five years of study. I did my first little research in classical analysis, with Prof. V.K. Ivanov, the best known Ekaterinburg mathematician. Ivanov was a good man but a busy one, the "prorector" of science. He advised me to go to computational math, because of its potential, or to join an active seminar. "You need interaction," he told me. Computational math seemed pedestrian to me at the time, and I joined the group-theory seminar of Prof. P.G. Kontorovich, the most active and competitive seminar in the Dept, with many enthusiastic participants and a list of open problems prominently posted on the wall. In my 1962 diploma thesis (article #1 at my website[1]) I solved the second problem on the problem list.

[1]Here and below, references #*n* are to the Annotated Articles list at `http://research.microsoft.com/~gurevich/annotated.htm`

CC: P.G. Kontorovich, did he win a Nobel prize in economics?

YG: No, the Nobel prize winner was L.V. Kantorovich. But my Kontorovich was remarkable in his own way. He went from an orphanage to founding the Ekaterinburg algebra school that is active to this very day. His humor was legendary, and he knew seemingly all the languages. Once I found him reading some text and complaining that he understands the text but does not recognize the language it is written in. It turned out that the language was Esperanto, forbidden as a "product of bourgeois internationalism and cosmopolitanism" in the USSR.

Maybe I can use this occasion to say a few words about Ural State University. Compared to other Soviet institutions, my alma mater (at least the hard sciences part of it) was a rare oasis of good will. Senior professors, like Ivanov and Kontorovich, created an atmosphere of decency. Even our philosophical seminars, a necessary fixture in Soviet universities, were different. Typically a philosophical seminar would be devoted to the study of the latest documents of the Central Committee of the Communist Party. The philosophical seminar of our Math Dept was devoted — surprise! — to philosophy, more exactly to the philosophical aspects of mathematics and mechanics. Later in my career, I spoke there about logic.

But I am getting ahead of myself. Upon getting my university diploma, I wanted to do math research at a university or the Academy of Sciences which offered better conditions. Conveniently the famous Steklov Math Institute of the Academy of Sciences opened a branch in Ekaterinburg and was hiring, and I applied there. But my chances were slim to none.

CC: Why? You probably were one of the best students or even the best student of your class.

YG: I might have been but Steklov was *Judenfrei*. Even Ural State University had limitations. They accepted me only as a PhD student by correspondence, but they hired me also as a lecturer. It actually worked well for me. I taught about 20 hours a week and did my math. Today it sounds exhausting to me, but at the time I enjoyed it all and had time left to hang out with my dissident friends. I remember even feeling somewhat guilty for being paid to have fun.

CC: What does it mean "PhD student by correspondence"?

YG: This is for people who have regular jobs. They may correspond with the university by mail.

CC: How did you move to mathematical logic? Did you study it at Ural State University?

YG: No, mathematical logic wasn't taught there. In fact there were few math logicians in the whole USSR. Formal (as opposite to dialectical) logic had hard time in the USSR. However things were improving during the 1960s. Kleene's "Introduction to Metamathematics" was translated into Russian, and I got it as a

birthday present in May 1962. I studied it and fell in love with logic. But what could an algebraist do in logic?

In the 1962–63 winter, a guest lecturer from Novosibirsk told us that a Polish student of Alfred Tarski, called Wanda Szmielew, proved the decidability of the first-order theory of abelian groups. A natural problem arose whether the first-order theory of ordered abelian groups is decidable. Szmielew and Tarski announced the decidability of that theory but then withdrew their claim. I worked on the problem. A big part of it was to understand when two ordered abelian groups have the same first-order properties. After a long chain of incremental advances, I proved that the theory is indeed decidable (#3). That became my PhD thesis which I defended in the spring of 1964 in Novosibirsk.

CC: Why Novosibirsk? Ural State University is not in Novosibirsk.

YG: By Soviet rules, you could defend your thesis in a science area X only at an institution with sufficient expertise in X. My choice was restricted to Moscow, Leningrad and Novosibirsk. Because of Maltsev's "Algebra and Logic" seminar, Novosibirsk was the best fit for me.

The 1964-65 academic year I was teaching at a new Krasnoyarsk State University in Siberia. By the way the word "State" in the names of Soviet universities meant simply "of the Soviet state". In the middle of that academic year I attended an algebraic winter school near Ekaterinburg. There I met a third-year Ural State University student Zoe, and I returned to Krasnoyarsk with a wife. We sought to move back to Ekaterinburg, and Ural State University accommodated us; the 1965-66 academic year I was already teaching there. My obsession with logic was contagious, and the logic seminar attracted the brightest students. During the winter breaks, we would rent a little house in the country to study but also to ski, play charades, etc.

CC: It sounds like scientific life in Soviet Union was similar to that in the West.

YG: It was similar, at least where hard sciences were concerned. But there were important differences. We were poorer. For example, Ural State University had no foreign currency, and western books and journals were not available in the library. More importantly, the totalitarian state was never far away. Here is an incident from one of those winter schools. One morning I woke up to much noise in another room, with none of my roommates in my room. I went to investigate. Two boys, surrounded by all the other students, were arguing whether there was state anti-Semitism in the USSR. Now all the eyes were upon me. What could I say? The safe lie of denial was out of the question, but publicly accusing the state of anti-Semitism was too dangerous, especially for a teacher. The chances were that there was an informant present. I spoke and spoke trying to humor my audience. I used whatever parables and jokes occurred to me leaving it up to the students to interpret things. Eventually passions subsided, and the attention

deviated to other topics. And I remember wishing to be able speak my mind safely.

But science and life interacted also independently of politics. Upon our return to Ekaterinburg, I had a bad motorcycle accident. In the hospital, they sewed me up but inadvertently infected me with hepatitis. As a result, I was quarantined for a month. No visitors were allowed in, and there were few books to read there. I used the time to think about the classical decision problem — classify infinite fragments of first-order predicate logic, given by restrictions on quantifier prefixes and the vocabulary, into decidable (for satisfiability) and undecidable. The problem attracted the attention of great logicians including Gödel, and there had been much progress in the early 1960s. If only one could prove that the $\forall\exists\forall\exists^*$ fragment with one binary relation is undecidable, the classification would be complete. The $\forall\exists\forall\exists^*$ problem was uniquely appropriate to my confinement. While the decision problem for ordered abelian groups required a long sustained effort and a long sequence of lemmas, each building upon the previous ones, the $\forall\exists\forall\exists^*$ problem seemed to require just a clever combinatorial trick. It was like jumping over a barrier. You give it a try and you fall, then another try and another fall, over and over again. Indeed, by the end of my quarantine, I got lucky and jumped over that barrier. The fame of the problem helped me to defend my Dr. of Math thesis later, in 1968.

CC: What is the Dr. of Math degree for? The Russian system of academic degrees seems different from that in English-speaking countries.

YG: It is different. The first Russian postgraduate academic degree, an equivalent of PhD, is Candidate of Science, and the second is Doctor of Science. Here "Science" is a variable to be replaced with "Mathematics", "Physics", etc. The Dr. of Science degree was a big deal at the time. If you taught at a university, the degree was a necessary and, in practice, sufficient condition for getting a full professorship. All academic degrees in Russia were — and are — subject to approval by the Central Attestation Committee of Russia.

CC: This was and continue to be also the system in Romania: nowadays, this Committee includes also Romanians from diaspora.

YG: The system is supposed to impose some standards but of course it can be abused.

CC: Did you go to Novosibirsk to defend your Dr. of Math thesis.

YG: No, the atmosphere in Novosibirsk changed for the worse, and a "Jewish dissertation" had little chance there. My dissertation had also a large algebraic component and thus qualified as algebraic. I defended it in Ekaterinburg, and the degree was eventually approved by the Central Attestation Committee.

CC: Your scientific activity splits into three periods: Soviet (up to 1973), Israeli (1974–1981), and American (since 1982). Let's visit them in that order.

YG: During the Soviet period I worked primarily on two subjects. One was related to the classical decision problem. The complete classification mentioned above comprised nine minimal undecidable classes and three maximal decidable ones. I wanted to understand whether there was an a priori reason that the classification resulted in a finite table. It turned out that indeed there was a rather general reason. That encouraged me to work on the extensions of the classification to first-order logic with equality or function symbols or both. I made a good progress, and the Institute of Philosophy of the Russian Academy of Sciences asked me to write a book on the subject. I write too slow to produce a book, but I wrote a survey. It was withdrawn from publication upon our emigration from the USSR. Later the survey became the core of the 1997 Springer book "The Classical Decision Problem" by Egon Börger, Erich Grädel and myself.

The other subject was the decidability of algebraic theories. In particular, I continued my work on ordered abelian groups. It bothered me that theorems in the literature on the subject were not first-order; they were mostly in terms of so-called convex subgroups. I extended my analysis to the variant of the monadic second-order theory of ordered abelian groups where the set variables ranged over convex subgroups. Somewhat miraculously, the decision procedure not only survived but simplified. The extended theory (and its easy further extensions) accounted for virtually all theorems in the literature. My attempts to publish these results in the USSR were unsuccessful (which is a separate story) but I published them after my departure (#25).

I also did some applied work. In my later undergraduate years, I worked at the university computing center. Later I worked with the transportation industry on linking railway transportation to trucks. All that work influenced me and changed my attitude on pure vs. applied science. You may have heard about a mathematician working on a difficult four-legged table problem. He generalized the problem to n-legged tables and solved the cases $n \leq 2$, the case $n = \infty$ and the case of sufficiently large n. In the process he advanced his career but the original problem remained open. That's pure science ☺

CC: Now tell me about the Israeli period.

YG: That period started with a touch of drama, or comedy. The first few months we lived in Jerusalem and studied Hebrew. During my first trip to Hebrew University, I met a young logician, Saharon Shelah. "Do you have an open problem," he asked me. I told him my conjecture that the $\exists^*\forall\exists^*$ fragment of first-order logic with equality, one unary function and infinitely many unary relations is decidable for satisfiability. When I saw him again, a week or two later, he told me that he confirmed my conjecture. I smiled: "Tell me about it." He did. I could not follow his explanation, partially because my Hebrew was still insufficient and my English nonexistent, but I realized that he had all the intuition that led me to the conjec-

ture and more. I was stunned. The first Israeli mathematician that I had a serious discussion with confirmed my conjecture. Maybe I should not seek a university position in Israel. I asked Shelah whether he had an open problem. He gave me his paper on the monadic second-order theory of the real line; it was submitted to Annals of Mathematics and had many open conjectures.

The paper was full with original ideas, but it was difficult to read. It took me months just to understand the paper. After a year or so of hard work, I confirmed or refuted most of Shelah's conjectures. He was most kind; as he proofread his paper, he added footnotes announcing my results. The incident resulted in a fruitful collaboration with Shelah on monadic (second-order) theories. Survey #64 reflects a large initial segment of the results of the monadic project.

CC: Give me some flavor of that work.

YG: Shelah conjectured that countability is not definable in the monadic second-order theory MT($\mathcal{R}$) of the real line $\mathcal{R}$ with just the order relation (and no addition or multiplication). In this connection I thought of the known and unsuccessful attempts to define countability in measure-theoretic terms. Of course sets of Lebesgue measure zero can be uncountable, but also sets of universal measure zero (defined by Hausdorff) can be uncountable, and sets of strong measure zero (defined by Borel) can be uncountable under the continuum hypothesis. I expected the conjecture to be true but it turned out, somewhat surprisingly, that countability was definable in MT($\mathcal{R}$) under the continuum hypothesis. The construction built heavily on the methods developed by Shelah in his original paper.

One of the main results in Shelah's original paper was the undecidability of MT($\mathcal{R}$). The proof was a clever interpretation of first-order arithmetic in MT($\mathcal{R}$). In #57, Shelah and I interpreted second-order arithmetic in MT($\mathcal{R}$). Later, in apparent contradiction with these results, we discovered that first-order arithmetic, let alone second-order arithmetic, cannot be interpreted in MT($\mathcal{R}$) (#79). A closer examination of Shelah's original reduction revealed that it (and our generalization of it) went beyond the standard model-theoretic notion of interpretability. And there was an interesting connection to set theory. If W is a model of ZFC, let W' be the model of ZFC resulting from the extension of W with a Cohen real, a real number that does not exist in W. Paul Cohen discovered a technique, *forcing*, that allows one to do things like that. Think of W as the current set-theoretic world, and of W' as the next world. Our reduction in #57 was a reduction of the next-world second-order arithmetic to the current-world MT($\mathcal{R}$).

CC: Not too many mathematicians or computer scientists have a theorem bearing their name. Tell me about the Gurevich-Harrington Forgetful Determinacy Theorem and how did you arrive at it.

YG: The 1980–81 academic year was a logic year at Hebrew University. Both Leo Harrington and I were there and proved the theorem independently; we talked

about that, and I volunteered to write the theorem up for publication. I do not know Leo's motivation. On my side, laziness played a role. In 1969, Michael Rabin used nondeterministic finite automata on infinite (colored) trees to prove the decidability of S2S, the monadic second-order theory of two successor relations. I understood the proof except for the complementation lemma according to which, for every tree automaton A, there is a complementary tree automaton that accepts exactly the trees that A doesn't. I kept thinking about the lemma but was reluctant to go through the difficult proof. And one day it occurred to me that it all, not only the complementation lemma but the whole paper of Rabin, was really about games. Things simplify (and become amenable to new useful generalizations) if you see them that way. For the games in question, the players can restrict themselves to "forgetful" strategies so that, at every point, the players need to remember only boundedly many bits about the history of the current play. Even finite automata are able to execute forgetful strategies; hence Rabin's result.

CC: Eventually you moved to the United States and to computer science. How did that happen?

YG: I had been contemplating more applied research already at the end of my Russian period but the Jerusalem logic seminar enthralled me. In spite of solving some high-profile logic problems, I was really a logic ignoramus. The seminar allowed me to learn cutting-edge logic developments. It was so much more efficient and so much more fun to learn things from seminar presentations than by reading papers. It was in Israel that I really became a logician, thanks to the logic seminar and joint work with Shelah. When the monadic project with Shelah began to wind down, I applied to computer science departments at some Israeli and US universities. All offers came from the US. I accepted a good offer from the University of Michigan, and in the summer of 1982 we moved to Ann Arbor, Michigan. There was another reason to choose the University of Michigan. Andreas Blass, the logician, was there, albeit in the Math Dept. Andreas and I have been actively collaborating ever since.

CC: Tell me about your work in finite model theory.

YG: Let me restrict myself to just one little story. At my first computer science conference, I heard a presentation by Moshe Vardi. He applied the interpolation theorem of first-order logic to relational databases viewed as first-order structures. I asked him whether his databases can be infinite, and he said yes. But naturally databases are finite of course. I looked into the issue. As I suspected, most classical theorems of first-order logic, including the interpolation theorem, fail in the finite case (#60). I had a sense of déjà vu. First-order logic wasn't right for ordered abelian groups, and it wasn't right for finite structures in the computer science context (#74).

Later on, a realization came that real databases are not necessarily finite af-

ter all. For a simple example, consider a salary database of some organization. The organization may use a popular database-query language SQL to query the salary database. In addition to relational-algebra operations, SQL has so-called grouping and aggregation operations. This allows the organization to compute various statistics over the database, e.g. the average salary and the total salary expense of the organization. Note that the average salary may not occur in the database and, ignoring degenerate cases, the total salary surely does not occur. Thus the database gives us a function from the employees to numbers, say rational numbers, and has rational arithmetic in the background. In that sense, it is not truly finite. To formalize this phenomenon of finite foreground and infinite background, Erich Grädel and I introduced *metafinite* structures (#109). The metafinite phenomenon is not restricted to databases. The states of an algorithm often are metafinite. Most classical theorems of first-order logic, including the interpolation theorem, fail in the metafinite case.

CC: Finite model theory has intimate relations with computational complexity but your complexity work went beyond that.

YG: It did. In particular I worked on the average-case reduction theory pioneered by Leonid Levin. Consider NP complete problems equipped with probability distributions on the instances. Some such problems turn out to be easy on average but others remain complete even for the average case. Proving such average-case completeness results is difficult, and the reason is this. While the range of a worst-case reduction may consist of very esoteric and unrepresentative instances of the target problem, the range of an average-case reduction should be of non-negligible probability. A popular article #85 argues in favor of an alternative, based on the average-case complexity, to the P=?NP question. Consider a game between Challenger and Solver where Challenger repeatedly picks instances of a given NP problem (with a fixed probability distribution), and Solver solves them. The idea is to measure Solver's time in terms of Challenger's rather than in terms of the instance size. It may take a long time to produce hard instances.

CC: Tell me about your work on abstract state machines. In particular what motivated it?

YG: Right upon starting at Michigan, I volunteered to teach "Introduction to Computer Science with Pascal" to computer science majors. The Dept chair did not like the idea ("We hired you to teach theory.") but agreed that I teach the course once. Preparing that course was instructive. I had not realized how much I fell behind in programming technology. At Ural State University, I programmed on the naked machine (01 for addition, 02 for substraction, etc.), and Pascal seemed advanced. The troubling part was that Pascal wasn't sufficiently documented. The interpreter on my Macintosh and the compiler on the university mainframe often disagreed on whether a given program is legal. Which, if either, of them was

right? What was I supposed to tell my 250 or so students? That was scary and brought home the problem of the semantics of programming languages.

In this connection, I studied denotational and algebraic semantics but found them wanting. It seemed infeasible to use them to specify the "dirty parts" of software. The celebrated declarativeness of denotational and algebraic specifications did not impress me. The advancers of the computer revolution weren't shy to program, specify and reason imperatively. There is a persistent confusion between declarative and high-level. Declarative specifications tend to be high-level, and executable specifications tend to involve unnecessary details. However I saw no reason why high-level specifications cannot be imperative and executable, amenable to testing and experimentation.

By Turing's thesis, every algorithm can be simulated by an appropriate Turing machine. Are Turing machines executable? In principle yes but of course this may be impractical. A bigger problem is that Turing machines work on the level of single bits. Are there more general state machines that specify algorithms on their natural abstraction level? Maybe that was too much to ask. But if yes then the reward would be high, for theory and practice. It would open a road to formalizing the notion of algorithms. On the practical level, it would enable us to specify software on whatever abstraction level is desired.

It was that line of thought that led me eventually to abstract state machines (ASMs). By the ASM thesis, every algorithm can be faithfully simulated by an ASM. We attempted to verify the thesis, which led to practical applications. There was also theoretical advances. The notion of sequential algorithms was formalized in #141; this formalization was used later by Nachum Dershowitz and myself to derive Turing's thesis from first principles (#188). Parallel and interactive algorithms were also formalized (#162).

CC: How did you get attracted to Microsoft?

YG: I was convinced that the ASM approach was more practical than other formal methods but all methods work on small examples, and my attempts to find an industrial partner were unsuccessful. In the summer of 1998, I visited Microsoft Research (MSR), in Redmond, WA, by an invitation of their crypto group. On that occasion I volunteered an ASM lecture. The lecture went rather well. There were many good questions. One of the MSR directors, Jim Kajiya, asked particularly astute and pointed questions. He said that he was surprised to see a formal specification method that seemed scalable. He proposed me to start a new MSR group on foundations of software engineering, and I jumped at the opportunity. The atmosphere and conditions at MSR are great, and the geographical area is spectacular. But what attracted me most was of course the opportunity to apply ASMs.

CC: Did it work? Could you apply ASMs at Microsoft?

YG: It was tough. I was lucky to hire the right people, and we built a tool, Spec Explorer, that facilitated writing executable specifications and playing with them. In particular, one could test the conformance between a spec and implementation. Spec Explorer was kept compatible with the Microsoft technology stack which consumed a lot of time and effort. The tech transfer was the biggest challenge. It is relatively easy to "sell" an incremental improvement to product groups. But Spec Explorer required learning and training, and product groups are busy. For a while we had only a few courageous groups here and there using Spec Explorer with our help. At a certain point, the European Union required from Microsoft high-level executable specifications of numerous communication protocols. The Windows division took over Spec Explorer and used it extensively and successfully.

CC: How applied is your work at Microsoft now? Do you use some theoretical results you proved as a "blue-sky researcher"?

YG: When Spec Explorer left MSR, I spend a couple of years catching up with theoretical work but then I returned to applications. Microsoft is an engineering place, and you catch the bug and want to influence technology. From time to time, I do internal consulting, developing efficient algorithms for various purposes. But my main current occupation is with Distributed Knowledge Authorization Language (DKAL). With the advent of cloud computing, a policy-management problem arises. In a brick-and-mortar setting, many policies may be unwritten. Clerks learn them from their peers. If they don't know a policy, they know whom to ask. In the cloud, the clerks disappear. The policies have to be handled automatically. The most challenging aspect is how to handle the interaction of policies, especially in federated scenarios where there is no central authority. DKAL was created to deal with such problems. The DKAL project has a large logic component so my logic expertise is useful.

CC: If you could dream about the year 3012, which result or concept would you like to see still "alive"?

YG: Hmm. "It's tough to make predictions, especially about the future," said Yogi Berra, the famous American baseball player and a philosopher of a kind. We live in quickly changing times. In the computer industry, long-term refers to just a few years ahead. It is an interesting question to what extent the future is predictable, even probabilistically. Let me just express the hope that the humanity will survive till 3012 and that the scientific method will survive as well. It may seem that the second is obvious given the first, but it is not necessarily so. Lucio Russo convincingly argues in "The Forgotten Revolution: How Science Was Born in 300 BC and Why it Had to Be Reborn" (2004) that the scientific method was not invented but reinvented by Galileo, Newton and their contemporaries, that science was discovered in the Hellenistic period and then was forgotten.

CC: How do you see the relevance of theoretical computer science for the com-

puter technology?

YG: Theory made weighty contributions to computer technology. Think of Alan Turing, John von Neumann, modern cryptography. The search technology that made Google rich is based on clever algorithms. One important theoretical contribution is for some reason less known to theorists than it deserves; I searched for it in vain in computation theory books. It is the 1965 discovery of LR(k) languages by Donald Knuth: "A language can be generated by an LR(k) grammar if and only if it is context-free and deterministic, if and only if it can be generated by an LR(1) grammar." LR(k) grammars can be parsed in time essentially proportional to the length of string, and their discovery revolutionized compiler construction.

But it is hard to influence computer technology by advancing theory, especially if the result is a non-incremental change in technology. "Nothing is more difficult than to introduce a new order," writes Niccolo Machiavelli in *The Prince*, "Because the innovator has for enemies all those who have done well under the old conditions and lukewarm defenders in those who may do well under the new." I lifted this quotation from a 2006 book "The Change Function: Why Some Technologies Take Off and Others Crash and Burn." The author, Pip Coburn, argues that the chances of adoption of a new disruptive technology is given by

$$\frac{\text{pain of the crises}}{\text{pain of adoption}}$$

To achieve successful technology transfer starting from just a theoretical advance is harder yet (though one may get lucky).

CC: Many thanks.

The Computational Complexity Column

BY

V. Arvind

Institute of Mathematical Sciences, CIT Campus, Taramani
Chennai 600113, India
arvind@imsc.res.in
http://www.imsc.res.in/~arvind

Ever since Reingold's deterministic logspace algorithm [66] for undirected graph reachability, logspace algorithms for various combinatorial problems have been discovered and it is now a flourishing area of research. Notable examples include special cases of directed graph reachability and planar graph isomorphism [23].

In this interesting article, Johannes Köbler, Sebastian Kuhnert and Oleg Verbitsky discuss the structural properties of interval graphs and other technical ingredients that go into their recent logspace isomorphism algorithm for interval graphs, along with some generalizations and new directions.

Bulletin of the EATCS no 107, pp. 43–71, June 2012

Around and Beyond the Isomorphism Problem for Interval Graphs

Johannes Köbler Sebastian Kuhnert* Oleg Verbitsky†
Humboldt-Universität zu Berlin, Institut für Informatik
{koebler,kuhnert,verbitsk}@informatik.hu-berlin.de

Abstract

The class of problems solvable in logarithmic space has recently replenished with the isomorphism testing for interval graphs. We discuss this result, prospects of its extension to larger classes of graphs, and related issues such as constructing canonical models of intersection graphs and solving the Star System Problem for restricted classes of graphs.

1 Introduction

Graph Isomorphism (GI for short) is the problem of determining whether or not two given graphs are isomorphic. The problem is in the class NP, but its complexity status is open since decades; see, e.g., the surveys [64, 31, 77, 4, 48]. Structural complexity theory provides good evidence showing that GI is hardly NP-complete; see the monograph [51]. The best known algorithm for GI, worked out by Babai, Luks, and Zemlyachenko [6], has moderately exponential running time $2^{O(\sqrt{n \log n})}$. Here and throughout, n denotes the number of vertices in an input graph. The best known lower bound is also surprisingly weak. Currently we do not even know if GI is P-hard under logspace reductions. Torán [71] shows that the problem is at least as hard as computing the determinant of an integer matrix (which in terms of complexity classes implies DET-hardness.)

In view of the fact that the general graph isomorphism problem has so far resisted all efforts to solve it more efficiently, it is natural to investigate its restrictions to particular classes of graphs or to reconsider the problem in other computational paradigms. An example of research in the latter direction is the search for

*Supported by DFG grant KO 1053/7–1.
†Supported by DFG grant VE 652/1–1. On leave from the Institute for Applied Problems of Mechanics and Mathematics, Lviv, Ukraine.

parameters making GI fixed-parameter tractable [46, 28, 76, 70, 52, 68]. An interesting open problem in this area is whether or not GI is fixed-parameter tractable with respect to tree-width [45].

GI for particular classes of graphs has a rich literature. A systematic overview can be found in the monograph [69]. Like in the theory of NP-completeness, two cases can be distinguished: *isomorphism-complete* graph classes, for which the problem remains as hard as in general, and *isomorphism-tractable* graph classes, for which it is solvable in polynomial time. As another resemblance to the theory of NP-completeness, a dichotomic phenomenon can be observed: just a few classes of graphs are discussed in the literature for which neither isomorphism-completeness nor polynomial-time solvability is known; the most prominent examples are the classes of circular-arc and trapezoid graphs (see [41, 21] for discussions of the former and [69, 75] for the latter).

Well-known examples for isomorphism-complete classes include bipartite and chordal graphs; see [13] for a comprehensive list of other basic examples and [7, 8, 75, 74] for some more advanced results.

A very powerful tractability result is recently obtained by Grohe and Marx [36] who showed that GI is solvable in polynomial time for each class of graphs excluding a fixed topological subgraph. This includes graphs of bounded vertex degree and graphs excluding a fixed minor. The polynomial-time algorithm by Luks [58] for the former case is used in [36] as a subroutine. An earlier polynomial-time algorithm for the latter case was designed by Ponomarenko [63]; see also [35]. Furthermore, examples of minor-free classes include graphs embeddable into a fixed surface (earlier polynomial-time algorithms are due to [29, 60, 34]) and graphs of bounded tree-width (an earlier polynomial-time algorithm is due to [12]).

The tractable cases of GI admit a finer classification through the computational concepts of polylogarithmic parallel time or logarithmic space (logspace for short). The first, and very important, logspace isomorphism algorithm was designed by Lindell for trees [56].

Let L denote the class of recognition problems solvable in logspace. Recall the hierarchy of low-complexity classes:

$$\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{LOGCFL} \subseteq \mathsf{AC}^1 \subseteq \mathsf{TC}^1 \subseteq \mathsf{NC}^2, \qquad \mathsf{NL} \subseteq \mathsf{DET} \subseteq \mathsf{TC}^1.$$

Note that L occupies a lower position than DET. Thus, Lindell's algorithm for trees, together with Torán's lower DET-bound for the general isomorphism problem, implies that isomorphism of trees is strictly easier than isomorphism of all graphs unless, for instance, NL = L. Somewhat surprisingly, the same conclusion holds for a number of much broader classes of graphs, in particular, for planar and interval graphs.

A graph is interval if its vertices can be assigned to intervals such that two vertices are adjacent if and only if their intervals have non-empty intersection.

Interval graphs have received persistent interest over the decades, finding applications (amongst others) in scheduling and computational biology; see e.g. [33]. Recognition of interval graphs played for example a role in establishing the linear structure of DNA (Benzer [10]).

Classifying classes of graphs as isomorphism-complete or polynomial-time solvable, an interesting phenomenon occurs: Once a particular isomorphism problem is put in P, it can often be put also in NC and, even more, in L. Examples of such a double jump are given by two classical classes of graphs, namely planar graphs (polynomial-time algorithm by Hopcroft and Tarjan [39], parallel AC^1 algorithm by Miller and Reif [61], logspace algorithm by Datta et al. [23]; see also survey [72]) and interval graphs (linear-time algorithm by Lueker and Booth [57], parallel AC^2 algorithm by Klein [47], logspace algorithm by Köbler et al. [49]). For graphs with bounded tree-width, the transition from P (Bodlaender [12]) to TC^1 was made by Grohe and Verbitsky [37], while the membership of this problem in L remains open. An important step towards this goal was made by Das, Torán, and Wagner [22] who put the problem in LOGCFL. A logspace isomorphism algorithm is known in the particular case of k-trees (Arvind et al. [2]). The "new wave" of logspace results on GI includes also the isomorphism test in [24] for graphs excluding one of the Kuratowski graphs K_5 and $K_{3,3}$ as a minor.

Note that in all cases where the isomorphism problem is solvable in logspace, we actually have an L-completeness result. For trees it was obtained by Jenner et al. [43].

A linear-time algorithm is also known for the isomorphism problem of planar graphs (Hopcroft and Wong [40]). It should be stressed that linear-time bounds are formally incomparable with L or NC bounds. On the other hand, the membership of a computational problem in L implies the existence of logarithmic time parallel algorithms for this problem (and then the next, practically important task is to minimize the number of processors in such an algorithm).

The remaining part of this survey is organized as follows. In Section 2 we establish several useful connections between graphs and hypergraphs. In Section 3 we describe recent logspace algorithms for computing canonical representations for interval graphs, proper interval graphs and some special classes of circular-arc graphs. We conclude this survey with a study of the Star System Problem and its connections to isomorphism testing. In Section 4 we observe that some known polynomial-time tractable cases of the Star System Problem can even be solved in logspace.

2 Graphs and hypergraphs

Recall that a *hypergraph* is a pair $(V, \mathcal{H})$, where V is a set of vertices and $\mathcal{H}$ is a family of subsets of V, called *hyperedges*. A graph is a hypergraph whose hyperedges are all of size 2. We will use the same notation $\mathcal{H}$ to denote a hypergraph and its hyperedge set; this causes no ambiguity if V has no *isolated* vertex (i.e., a vertex that is not contained in any hyperedge). The vertex set V of $\mathcal{H}$ will be denoted by $V(\mathcal{H})$. An *isomorphism* from a hypergraph $\mathcal{H}$ to a hypergraph $\mathcal{K}$ is a bijection $\phi\colon V(\mathcal{H}) \to V(\mathcal{K})$ such that $X \in \mathcal{H}$ if and only if $\phi(X) \in \mathcal{K}$ for every $X \subseteq V(\mathcal{H})$. We allow multiple hyperedges; therefore, ϕ must also respect the multiplicity of every hyperedge X.

Hypergraphs can be used to represent certain graphs in a succinct way. For example, we can associate with a hypergraph $\mathcal{H}$ the *intersection graph* $\mathbb{I}(\mathcal{H})$ on vertex set $\mathcal{H}$ where $X \in \mathcal{H}$ and $Y \in \mathcal{H}$ are adjacent if and only if they have a non-empty intersection. We call a hypergraph *connected* if its intersection graph is connected. Of course,

$$\mathcal{H} \cong \mathcal{K} \implies \mathbb{I}(\mathcal{H}) \cong \mathbb{I}(\mathcal{K}), \tag{1}$$

but the converse implication does not hold in general.

Another graph that can be derived from a hypergraph $\mathcal{H}$ is the *incidence graph* $I(\mathcal{H})$. This is a colored bipartite graph with the class of red vertices $V(\mathcal{H})$, the class of blue vertices $\mathcal{H}$, and edges between all $v \in V(\mathcal{H})$ and $X \in \mathcal{H}$ such that $v \in X$. A hyperedge X of multiplicity k contributes k blue vertices in $I(\mathcal{H})$ (with the same adjacency pattern to the red part of the graph). In contrast to the intersection graph, the incidence graph contains full information about the underlying hypergraph, as we have

$$\mathcal{H} \cong \mathcal{K} \iff I(\mathcal{H}) \cong I(\mathcal{K}). \tag{2}$$

This equivalence reduces testing isomorphism of hypergraphs with n vertices and m hyperedges to testing isomorphism of colored graphs with $n+m$ vertices (where colors can, in fact, be removed by using standard gadgets, for example, by connecting all red vertices to an auxiliary triangle). Although hypergraph isomorphism reduces to GI, in many cases it is preferable to solve it directly; see [59, 5, 3] for the currently best algorithms. Though these algorithms have an exponential running time, it turns out that some interesting cases of GI can be solved efficiently by reducing them to the isomorphism problem for related hypergraphs.

An inclusion-maximal clique in a graph G will be called *maxclique*. The *bundle hypergraph* $\mathcal{B}(G)$ has one node for each maxclique of G, and for each vertex v of G a hyperedge B_v consisting of all maxcliques that contain v. We call B_v the *(maxclique) bundle* of v. Since two vertices are adjacent if and only if they are

contained in a common maxclique, G is isomorphic to the intersection graph of the bundle hypergraph $\mathbb{I}(\mathcal{B}(G))$. Hence, (1) implies that

$$G \cong H \iff \mathcal{B}(G) \cong \mathcal{B}(H). \tag{3}$$

Unlike (2), the equivalence (3) does not yield an efficient reduction in general (because a graph can have up to $3^{n/3}$ maxcliques [62]). However, it does in the case of interval graphs; see Section 3.1.

We notice that the bundle hypergraph $\mathcal{B}(G)$ is the dual of the clique hypergraph of G. The *clique hypergraph* $\mathcal{C}(G)$ of a graph G has the same vertex set as G and the maxcliques of G as its hyperedges. The *dual* of a hypergraph $\mathcal{H}$ is the hypergraph $\mathcal{H}^* = \{v^* : v \in V(\mathcal{H})\}$ on vertex set $\mathcal{H}$, where $v^* = \{X \in \mathcal{H} : v \in X\}$ consists of all hyperedges in $\mathcal{H}$ containing v. Thus, taking the dual of a hypergraph corresponds to transposing its incidence matrix.

Twins in a hypergraph are two vertices such that every hyperedge contains either both or none of them. A hyperedge $X \in \mathcal{H}$ of multiplicity k contributes k twin vertices in the dual hypergraph $\mathcal{H}^*$. Conversely, if v and u are twins in $\mathcal{H}$, then $v^* = u^*$, and therefore, any class of k twins in $\mathcal{H}$ contributes a hyperedge of multiplicity k in $\mathcal{H}^*$. Clearly, the duals of isomorphic hypergraphs are again isomorphic. Since the two hypergraphs $\mathcal{H}$ and $(\mathcal{H}^*)^*$ are isomorphic via the mapping $x \mapsto x^*$, it follows that the converse implication is also true, implying that

$$\mathcal{H} \cong \mathcal{K} \iff \mathcal{H}^* \cong \mathcal{K}^*. \tag{4}$$

Other useful hypergraphs that can be associated with a graph G are the *open* and *closed neighborhood hypergraphs*, denoted by $\mathcal{N}(G)$ and $\mathcal{N}[G]$, respectively. The *open neighborhood* of a vertex v in G consists of all vertices adjacent to v and is denoted by $N(v)$, whereas the vertex set $N[v] = N(v) \cup \{v\}$ is called the *closed neighborhood* of v. Both hypergraphs $\mathcal{N}(G)$ and $\mathcal{N}[G]$ have the same vertex set as G and the open (resp. closed) neighborhoods of these vertices as hyperedges, i.e., $\mathcal{N}[G] = \{N[v]\}_{v \in V(G)}$ and $\mathcal{N}(G) = \{N(v)\}_{v \in V(G)}$. Note that in contrast to $\mathcal{N}[G]$, which never contains isolated vertices, $\mathcal{N}(G)$ inherits all isolated vertices from G.

If $N[u] = N[v]$, we call the vertices u and v *twins in the graph* G. Note that u and v are twins in G if and only if they are twins in $\mathcal{N}[G]$. Note also that the closed neighborhoods of twins are equal and form a hyperedge of multiplicity greater than one in the hypergraph $\mathcal{N}[G]$. Twins in a graph are always adjacent and their bundles $B_u = B_v$ coincide, implying that the hyperedge $B_u = B_v$ in $\mathcal{B}(G)$ is also of multiplicity greater than one. Of course,

$$G \cong H \implies \mathcal{N}[G] \cong \mathcal{N}[H]. \tag{5}$$

The converse implication is not true in general; for an example see Section 4.2. However, it holds true (and is useful) for proper interval graphs; see Corollary 4.4.

The applicability of the relationship $\mathcal{N}[G] \cong \mathcal{N}[H]$ vs. $G \cong H$ (stated in the language of matrices) for isomorphism testing for particular graph classes was discovered and exploited by Chen [16, 17, 18].

In more generality we will discuss the conditions under which implication (5) can be reversed in Section 4.

3 Canonical representations for intersection graphs

We call a hypergraph $\mathcal{H}$ an *intersection model* of a graph G, if G is isomorphic to the intersection graph $\mathbb{I}(\mathcal{H})$. Any isomorphism from G to $\mathbb{I}(\mathcal{H})$ is called a *representation* of G by an intersection model. Every graph possesses an intersection model since, as mentioned above,

$$G \cong \mathbb{I}(\mathcal{B}(G)) \tag{6}$$

via the isomorphism $v \mapsto B_v$. When we put natural restrictions to intersection models, we obtain special classes of intersection graphs. Classical examples are interval graphs (having intervals on a line as their intersection models), circular-arc graphs (arcs on a circle), circle graphs (chords of a circle), permutation graphs (segments with endpoints in two opposite parallel lines), and trapezoid graphs (trapezoids with sides in two opposite parallel lines).

In order to represent interval graphs and circular-arc graphs by intersection models it is more convenient to use integer intervals and arcs on a discrete cycle. We refer to these intersection models as *interval models* and *arc models*, respectively. It is not hard to see that this convention does not affect the resulting graph classes.

The *canonical representation problem* for a class C of intersection graphs is defined as follows: For a given graph G, either compute a representation ρ_G of G by an appropriate intersection model (if $G \in C$) or determine that no such model exists (if $G \notin C$). Moreover, it is required that isomorphic graphs $G \cong H$ in C receive identical intersection models $\rho_G(G) = \rho_H(H)$. For a specified algorithm, we call its output ρ_G on input $G \in C$ a *canonical representation* of G and the resulting model $\rho_G(G)$ a *canonical model* of G. Note that such an algorithm simultaneously solves both the recognition (even model construction) and the isomorphism (even canonical labeling) problems for C.

We quickly recall the *canonical labeling problem* for a graph class C. Given a graph $G \in C$ with n vertices, we have to compute a map $\lambda_G \colon V(G) \to \{1, \ldots, n\}$ so that the graph $\lambda_G(G)$, the image of G under λ_G on the vertex set $\{1, \ldots, n\}$, is the same for isomorphic input graphs. Equivalently, for any graph $G \in C$ we have to compute an isomorphism λ_G from G to a graph $G^\star$ so that

$$G \cong H \implies G^\star = H^\star.$$

In fact, the condition $V(G^\star) = \{1, \ldots, n\}$ requires no special care as the vertices of $G^\star$ can be sorted and renamed. We say that λ_G is a *canonical labeling* and $\lambda_G(G)$ is a *canonical form* of G.

Note the similarity between the pairs of notions *canonical labeling/canonical form* and *canonical representation/canonical model* for a class of intersection graphs. Obviously, the former can be obtained from the latter by taking the intersection graph of the canonical model.

3.1 Interval graphs

In this section we describe the logspace algorithm of [49] that computes a canonical interval model for any given interval graph G. The algorithm first transforms G into its bundle hypergraph $\mathcal{B}(G)$ over the vertex set consisting of all maxcliques of G. The maxcliques of G can be found in logspace by applying the following lemma.

Lemma 3.1 (Laubner [54])**.** *Every maxclique C of an interval graph G contains vertices u and v such that $C = N[u] \cap N[v]$.*

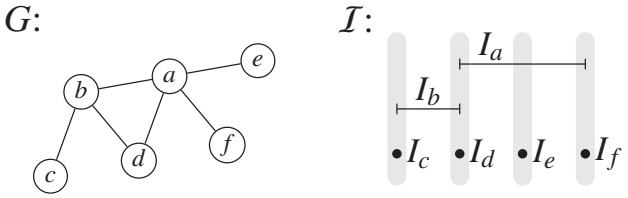
For adjacent vertices u and v in an arbitrary graph holds: If $N[u] \cap N[v]$ is a clique, it is maximal. Lemma 3.1 shows that any maxclique in an interval graph is of this kind and, hence, can be represented by a pair of vertices u and v (that are adjacent and satisfy the condition that $N[u] \cap N[v]$ is a clique). An explicit representation of the bundle hypergraph $\mathcal{B}(G)$ of G can be computed in logspace by listing, for each bundle B_v, the maxcliques that contain v.

The following lemma shows that the bundle hypergraph $\mathcal{B}(G)$ is indeed a good starting point for constructing an interval model for a given graph G. We call an interval model $\mathcal{I}$ of G *minimal* if G admits no interval model that has fewer points than $\mathcal{I}$.

Lemma 3.2 ([49, Lemma 2.3])**.** *Every minimal interval model $\mathcal{I}$ of an interval graph G is isomorphic to the bundle hypergraph $\mathcal{B}(G)$.*

Lemma 3.2 implies that the minimal interval model of G is unique up to hypergraph isomorphism, and any such model can be obtained from the bundle hypergraph $\mathcal{B}(G)$ by renaming its vertices to integers; Fig. 1 shows an example.

Given a hypergraph $\mathcal{H}$, call a linear order $<$ on $V(\mathcal{H})$ *interval* if every hyperedge of $\mathcal{H}$ forms an interval w.r.t. $<$. If $\mathcal{H}$ admits an interval order, then it is called an *interval hypergraph*. Equivalently, an interval hypergraph is a hypergraph isomorphic to a system of intervals of integers, which is then called an *interval model* of this hypergraph. For an interval order $<$ of $\mathcal{H}$, let $r_<(v)$ denote the rank of a vertex $v \in V(\mathcal{H})$ w.r.t. $<$, and let $\mathcal{H}^<$ denote the image of $\mathcal{H}$ under the map $r_<$.



$\mathcal{B}(G)$	$\{b,c\}$	$\{a,b,d\}$	$\{a,e\}$	$\{a,f\}$
B_a	0	1	1	1
B_b	1	1	0	0
B_c	1	0	0	0
B_d	0	1	0	0
B_e	0	0	1	0
B_f	0	0	0	1

Figure 1: An interval graph G, a minimal interval model $\mathcal{I}$ of G, and the bundle hypergraph $\mathcal{B}(G)$ of G. The latter is given by its incidence matrix with columns indexed by vertices (i.e., maxcliques) and rows indexed by hyperedges (i.e., bundles).

Clearly, $<$ is an interval order of $\mathcal{H}$ if and only if $\mathcal{H}^<$ is an interval model of $\mathcal{H}$ on the segment of integers $\{1,\ldots,n\}$.

A binary matrix has the *consecutive-ones property* if its columns can be permuted so that in each row the ones are consecutive. Viewing the matrix as incidence matrix of a hypergraph shows that testing for the consecutive-ones property is equivalent to recognizing interval hypergraphs. Dom [26] surveys algorithmic aspects of the consecutive-ones property.

The following theorem is an immediate consequence of Lemma 3.2 and the general relation (6).

Theorem 3.3 (cf. [33, Theorems 8.1 and 8.3])**.** *G is an interval graph if and only if $\mathcal{B}(G)$ is an interval hypergraph.*

Hence, in order to decide whether G is interval, it suffices to check whether the bundle hypergraph $\mathcal{B}(G)$ is interval.

Moreover, from any interval ordering $<$ of $\mathcal{B}(G)$, we an easily construct an interval representation ρ_G. For any vertex $v \in V(G)$, define $\rho_G(v) = r_<(B_v)$. Since the mapping $v \mapsto B_v$ is a graph isomorphism from G to $\mathbb{I}(\mathcal{B}(G))$ and $r_<$ is a hypergraph isomorphism from $\mathcal{B}(G)$ to the interval system $\mathcal{B}(G)^<$, the map ρ_G is an isomorphism from G to $\mathbb{I}(\mathcal{B}(G)^<)$. Hence, ρ_G is indeed an interval representation of G.

Since the bundle hypergraph $\mathcal{B}(G)$ and the map $v \mapsto B_v$ are constructible in logspace due to Lemma 3.1, it follows that ρ_G is computable in logspace, provided that an interval ordering $<_\mathcal{H}$ for a given interval hypergraph $\mathcal{H}$ is computable in logspace. Moreover, this reduction even gives a canonical interval representation ρ_G of G, if $<_\mathcal{H}$ is a *canonical ordering* of $\mathcal{H}$, meaning that $\mathcal{H}^{<_\mathcal{H}} = \mathcal{K}^{<_\mathcal{K}}$ whenever $\mathcal{H} \cong \mathcal{K}$ are isomorphic interval hypergraphs. Indeed, $G \cong H$ implies that $\mathcal{B}(G) \cong \mathcal{B}(H)$ and hence the resulting interval systems $\mathcal{B}(G)^{<_{\mathcal{B}(G)}}$ and $\mathcal{B}(H)^{<_{\mathcal{B}(H)}}$ are equal.

Lemma 3.4. *The canonical representation problem for interval graphs is reducible in logspace to the canonical ordering problem for interval hypergraphs.*

Computing an interval ordering for interval hypergraphs

In this subsection, we describe an algorithm for computing an interval ordering for a given interval hypergraph $\mathcal{H}$ (or detecting that $\mathcal{H}$ is not interval).

PQ-trees, introduced by Booth and Lueker [14], provide a succinct way to represent all possible interval orderings of an interval hypergraph $\mathcal{H}$. A *PQ-tree* for $\mathcal{H}$ is an ordered rooted tree. Its leaves are the vertices of $\mathcal{H}$ and its inner nodes are classified as either P- or Q-nodes. Clearly, the ordering of the tree induces a unique linear order on the leaves of the tree. It is possible to change the tree order of a PQ-tree according to the following rules. The children of a P-node can be reordered arbitrarily, while the ordering of the children of a Q-node can only be reversed. A tree order is *permissible* if it can be obtained from a combination of such reorderings. A PQ-tree represents the set of all linear orders on its leaves which are induced by a permissible tree order.

Booth and Lueker [14] showed that a PQ-tree encoding all interval orderings of a given interval hypergraph can be computed in linear time. Their algorithm starts with the PQ-tree T for the empty hypergraph (where all leaves are attached to a single P-node). Then it iteratively incorporates into T the restrictions caused by each hyperedge. Klein [47] reduced the number of iterations from linear to logarithmic by incorporating several hyperedges in one step. This results in a parallel AC^2 algorithm. In [49] it was shown that a PQ-tree for a given interval hypergraph $\mathcal{H}$ can even be computed in logspace. The key observation behind this is that the *overlap component tree* of $\mathcal{H}$ can be viewed as PQ-tree and is constructible in logspace. This tree comprises of slot nodes, which are interpreted as P-nodes, and overlap component nodes, which are interpreted as Q-nodes. To give its precise definition we need some more notation.

We say that two sets A and B *overlap* and write $A \between B$ if A and B have a nonempty intersection but neither of them includes the other. The *overlap graph* $\mathbb{O}(\mathcal{H})$ of a hypergraph $\mathcal{H}$ is the subgraph of the intersection graph $\mathbb{I}(\mathcal{H})$, where the vertices corresponding to the hyperedges A and B are adjacent if and only if they overlap. Of course, $\mathbb{O}(\mathcal{H})$ can be disconnected even if $\mathbb{I}(\mathcal{H})$ is connected. A subset $\mathcal{O}$ of the hyperedges of $\mathcal{H}$ spanning a connected component of $\mathbb{O}(\mathcal{H})$ will be referred to as an *overlap component* of $\mathcal{H}$. This is a subhypergraph of $\mathcal{H}$ and should not be confused with the corresponding induced subgraph of $\mathbb{O}(\mathcal{H})$. Note that a hyperedge of an overlap component inherits the multiplicity that it has in $\mathcal{H}$. In the case that $\mathbb{O}(\mathcal{H})$ is connected, $\mathcal{H}$ will be called an *overlap-connected hypergraph*.

Lemma 3.5 (Chen and Yesha [19])**.** *Let $\mathcal{H}$ be an interval hypergraph. If $\mathcal{H}$ is overlap-connected, then it has, up to permutation of twins and reversing, a unique interval ordering.*

Since the resulting interval model does not depend on the ordering of twins inside a slot, it follows that an overlap-connected interval hypergraph has at most two different interval models inside the range $\{1, \ldots, n\}$ and that these models are mirror symmetric to each other.

In fact, such a model can be constructed in logspace as follows. In a preprocessing step, compute a walk $X_1, \ldots, X_N$ in the overlap graph $\mathbb{O}(\mathcal{H})$ that visits every hyperedge of $\mathcal{H}$ at least once (this can be done in logspace using Reingold's universal exploration sequences [66]). Then iterate over the hyperedges X_i in this walk, computing an interval I_i for each X_i. Once the first interval $I_1 = [1, |X_i|]$ is fixed, the cardinality of $X_1 \cap X_2$ leaves only two possibilities for I_2 (resulting in reflected representations), and once I_{i-2} and I_{i-1} are fixed, I_i is uniquely determined; see Fig. 2. As only the two previous intervals have to be remembered, this computation is possible in logspace. In a post-processing step, verify that the result is indeed an interval model of $\mathcal{H}$ and shift the model into the range $\{1, \ldots, n\}$ if necessary.

A *slot* of $\mathcal{H}$ is an inclusion-maximal set S of twins, i.e., the slots are the equivalence classes of the twin relation.

If $\mathcal{O}$ and $\mathcal{O}'$ are different overlap components, then either every two hyperedges $A \in \mathcal{O}$ and $A' \in \mathcal{O}'$ are disjoint or all hyperedges of one of the two components are contained in a single slot of the other component. (This follows from the simple observation that the conditions $B \subset A$, $B \not\parallel B'$, and $\neg(B' \not\parallel A)$ imply $B' \subset A$.) This containment relation determines a tree-like decomposition of $\mathcal{H}$ into its overlap components. Specifically, let S be a slot of an overlap component $\mathcal{O}$ of $\mathcal{H}$. We say that an overlap component $\mathcal{Q}$ of $\mathcal{H}$ is *located at* slot S of $\mathcal{O}$ if $V(\mathcal{Q}) \subseteq S$ and there is no "intermediate" overlap component $\mathcal{O}' \neq \mathcal{O}$ such that $V(\mathcal{O}') \subseteq S$ and $\mathcal{Q}$ is contained in some slot of $\mathcal{O}'$. Furthermore, a vertex v of $\mathcal{H}$ is *located at* slot S of $\mathcal{O}$ if $v \in S$ and there is no overlap component $\mathcal{O}'$ located at slot S of $\mathcal{O}$ such that $v \in V(\mathcal{O}')$.

Now we are ready to give a precise definition of the overlap component tree of an interval hypergraph $\mathcal{H}$. We assume that $\mathcal{H}$ is connected: To ensure this, we add an additional hyperedge $B_0 = V(\mathcal{H})$ (this has no influence on the possible interval orderings of $\mathcal{H}$).

The nodes of the overlap component tree of $\mathcal{H}$ are the overlap components of $\mathcal{H}$, their slots, and the vertices of $\mathcal{H}$. Since a slot S of $\mathcal{O}$ may belong also to another overlap component, we denote the corresponding slot node by $S_\mathcal{O}$. The children of an overlap component node $\mathcal{O}$ are the slots of $\mathcal{O}$. The children of a slot node $S_\mathcal{O}$ are the vertices and the overlap components located at the slot S

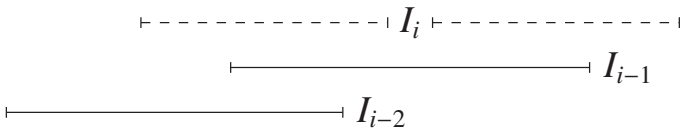


Figure 2: Proof of Lemma 3.5: Let $X_i \not\subseteq X_{i-1} \not\subseteq X_{i-2}$. If I_{i-1} is already determined, only two positions for I_i satisfy $|I_i| = |X_i|$ and $|I_{i-1} \cap I_i| = |X_{i-1} \cap X_i|$. If also I_{i-2} is given, at most one of them additionally satisfies $|I_{i-2} \cap I_i| = |X_{i-2} \cap X_i|$.

of $\mathcal{O}$. As $\mathcal{H}$ is connected, there is an overlap component $\mathcal{O}_0$ with $V(\mathcal{O}_0) = V(\mathcal{H})$. Thus, $\mathcal{O}_0$ is the root of the overlap component tree. An example for an overlap component tree can be found in Fig. 3.

Suppose that $\mathcal{H}$ is an interval hypergraph. To interpret its overlap component tree as PQ-tree, treat all slot nodes as P-nodes, and all overlap component nodes as Q-nodes. By Lemma 3.5, the slots of each overlap component can be ordered uniquely up to reversing; this defines the order on the children of Q-nodes. It is easy to verify that every rearrangement of this PQ-tree again induces an interval ordering of $\mathcal{H}$. Using the uniqueness given by Lemma 3.5 and a simple inductive argument on the depth of the overlap component tree, one can show that every interval representation of $\mathcal{H}$ can be obtained in this way (cf. [42]).

It is not hard to verify that the overlap component tree (and hence a PQ-tree) for a given interval hypergraph $\mathcal{H}$ can be computed in logspace. Thus, we can compute an interval ordering for $\mathcal{H}$ in logspace. In order to compute a canonical interval ordering for $\mathcal{H}$ we have to do some more work.

Of course, isomorphic interval hypergraphs have isomorphic overlap component trees (when considered as rooted trees but ignoring the order on the children). As shown in Fig. 3, the converse is not true, since the overlap component tree does not contain enough information on the underlying hypergraph. The idea is to reflect this missing information by coloring the nodes of the tree in such a way that the underlying hypergraph can be reconstructed up to isomorphism from the resulting tree. Roughly speaking, we will refine the tree in such a way that only rearrangements of the corresponding PQ-tree become isomorphic to the refined tree. Then selecting an interval ordering in a canonical way corresponds to selecting an isomorphic copy of the refined tree in a canonical way. For the latter task we can use Lindell's tree canonization algorithm.

Computing a canonical ordering for interval hypergraphs

In this subsection, we describe an algorithm for computing canonical orderings for interval hypergraphs. Together with Lemma 3.4 this gives us a logspace algorithm for computing a canonical representation for interval graphs.

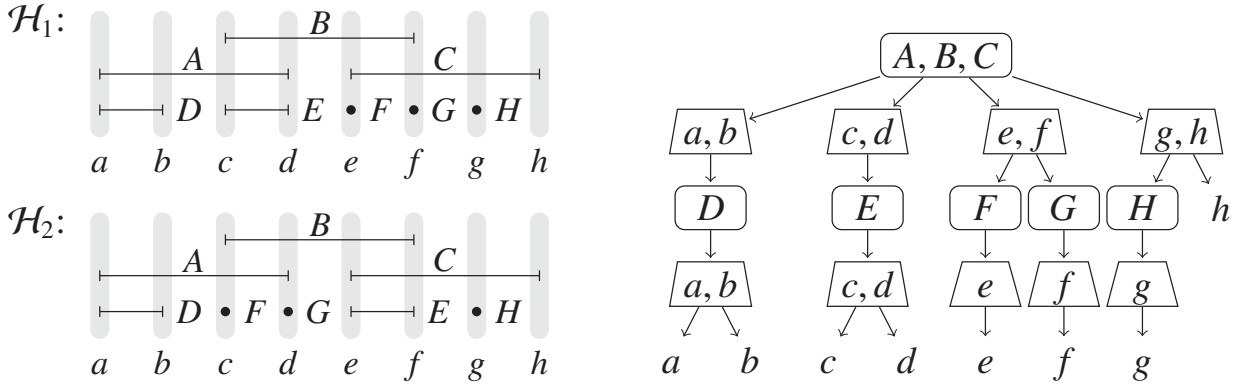


Figure 3: An interval hypergraph $\mathcal{H}_1$ and its overlap component tree. In the tree, the node of an overlap component O is given by listing the hyperedges in O; a slot node S_O is given by listing the vertices contained in S (we omit the reference to O as it is understood from the tree structure). The hypergraph $\mathcal{H}_2$ is not isomorphic to $\mathcal{H}_1$; yet both have isomorphic overlap component trees.

Theorem 3.6 ([49, Theorem 4.6])**.** *The canonical ordering problem for interval hypergraphs can be solved in logspace.*

Corollary 3.7. *The canonical representation problem for interval graphs is solvable in logspace.*

As explained above, we reduce the task of computing a canonical interval ordering for $\mathcal{H}$ to computing a canonical labeling of an associated tree $\mathbb{T}(\mathcal{H})$. This tree has the property that $\mathcal{H} \cong \mathcal{K}$ if and only if $\mathbb{T}(\mathcal{H}) \cong \mathbb{T}(\mathcal{K})$. To construct $\mathbb{T}(\mathcal{H})$ from the overlap component tree of $\mathcal{H}$, we first compute canonical interval models for each overlap component (using the lexicographically smaller of the at most two that are possible by Lemma 3.5) and assign these models as colors to the overlap component nodes. For an asymmetric overlap component, the chosen model already fixes the order of its slots, which can be enforced by assigning ascending colors to the slot nodes. For a symmetric overlap component with at most two slots, any ordering of its slots is fine; for one with more than two slots, we employ a small gadget to ensure that the order of its slots can only be reflected: Between the overlap component node O and its slots, we introduce three connector nodes lo_O, mi_O and hi_O. Fix an arbitrary interval order $<$ of O; it induces an order $<^*$ on the slots of O. For each slot S of O, denote its position from the left and right by

$$l_O(S) = |\{S' : S' \text{ is a slot of } O \text{ with } S' \leq^* S\}|$$
$$r_O(S) = |\{S' : S' \text{ is a slot of } O \text{ with } S' \geq^* S\}|$$

A slot node S_O becomes a child of lo_O if $l_O(S) < r_O(S)$, a child of mi_O if $l_O(S) = r_O(S)$, and a child of hi_O if $l_O(S) > r_O(S)$. (Choosing a different interval

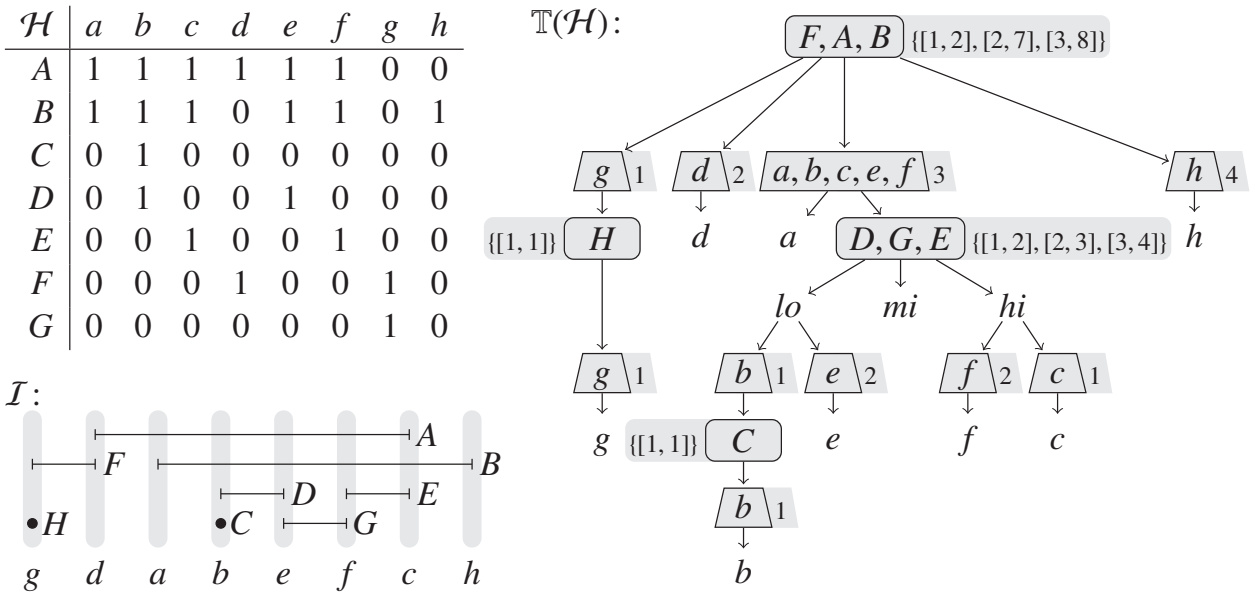


Figure 4: An interval hypergraph $\mathcal{H}$ and its tree representation $\mathbb{T}(\mathcal{H})$. Gray areas in $\mathbb{T}(\mathcal{H})$ indicate the color of overlap components and their slots. An overlap component O is represented by listing the hyperedges in O (sorted, for the reader's convenience, by their corresponding intervals in the canon of O). We omit the references from slot and connector nodes to their overlap components as they are understood from the tree structure. The interval system $\mathcal{I} \cong \mathcal{H}$ can be derived from $\mathbb{T}(\mathcal{H})$ by reading the vertex nodes from left to right.

ordering for O would only result in exchanging the nodes lo_O and hi_O, yielding an isomorphic tree.) Fig. 4 shows an example for a tree representation $\mathbb{T}(\mathcal{H})$. As indicated above, this construction ensures that a canonical labeling of $\mathbb{T}(\mathcal{H})$ specifies a canonical rearrangement of the PQ-tree, which in turn determines a canonical interval order of $\mathcal{H}$ (see [49] for details).

3.2 Proper interval graphs

An intersection model $\mathcal{H}$ is *proper* if the sets in $\mathcal{H}$ are pairwise incomparable by inclusion. G is called a *proper interval graph* if it has a proper interval model. In this section, we describe a logspace algorithm for computing a canonical proper interval model for a given proper interval graph.

As a consequence of the following theorem, we can reduce the problem of recognizing proper interval graphs to the problem of recognizing interval hypergraphs.

Theorem 3.8 (Roberts [67, 27])**.** *G is a proper interval graph if and only if $\mathcal{N}[G]$ is an interval hypergraph.*

Theorem 3.8 does not provide us with an appropriate interval model for a proper interval graph, since $\mathcal{N}[G]$ need not even be an intersection model for G. However, it is possible to convert an interval ordering of $\mathcal{N}[G]$ into a proper interval model for G, if G is a proper interval graph. This is done via a tight interval model for G.

An interval system is *tight* if the intervals have the following property: whenever $A = [a^-, a^+]$ includes $B = [b^-, b^+]$, we have $a^- = b^-$ or $a^+ = b^+$. It is not hard to see that any tight interval model for a graph G can be converted to a proper interval model for G (cf. Tucker [73]): If several intervals start (resp. end) at the same point, introduce new points to extend the shorter intervals so that none is contained in the other anymore. In fact, this transformation is possible in logspace (see [50] for details). Thus, the following lemma provides us with a proper interval representation of G.

Lemma 3.9. *Given an interval ordering of $\mathcal{N}[G]$, a tight interval representation of G can be constructed in logspace.*

Proof. Given an interval ordering $<$ of $\mathcal{N}[G]$, for each vertex v of G we let v^- and v^+ denote the two endpoints of the interval $N[v] = [v^-, v^+]$ w.r.t. $<$. Define $\rho(v) = N^+[v] = [v, v^+]$. The map ρ is an interval representation of G. Indeed, if u and v are adjacent, either $u \in N^+[v]$ (if $v < u$) or $v \in N^+[u]$ (if $u < v$) holds. In either case $N^+[u] \cap N^+[v] \neq \emptyset$. If u and v are not adjacent, $u \notin N^+[v]$ and $v \notin N^+[u]$, which implies that the intervals $N^+[u]$ and $N^+[v]$ are disjoint. See Fig. 5 for an example.

Next we show that ρ is tight. Suppose that $N^+[u] \subseteq N^+[v]$. Since $v \in N[v^+]$, we have also $u \in N[v^+]$. Therefore, $N^+[u] = [u, u^+]$ contains v^+ and $u^+ = v^+$.

Finally, given G and an interval ordering $<$ of $\mathcal{N}[G]$, the map ρ can be easily computed in logspace. □

To obtain a canonical proper interval representation for the class of proper interval graphs, we can combine any canonical labeling for this class with the proper interval representation of the resulting canon. To compute a canonical labeling for proper interval graphs we can for example use the algorithm provided by Corollary 3.7 (which even works for all interval graphs). Alternatively, since we anyway have to compute an interval ordering of $\mathcal{N}[G]$, we can also make use of the following lemma.

Lemma 3.10 (cf. [25, Corollary 2.5])**.** *If G is a connected proper interval graph, then $\mathcal{N}[G]$ has, up to reversing and up to permutation of twins, a unique interval ordering.*

This result can also be derived from Lemma 3.5 and the fact that, for a connected proper interval graph G, the hypergraph $\mathcal{N}[G]\backslash\{V(G)\}$ is overlap-connected; see [49].

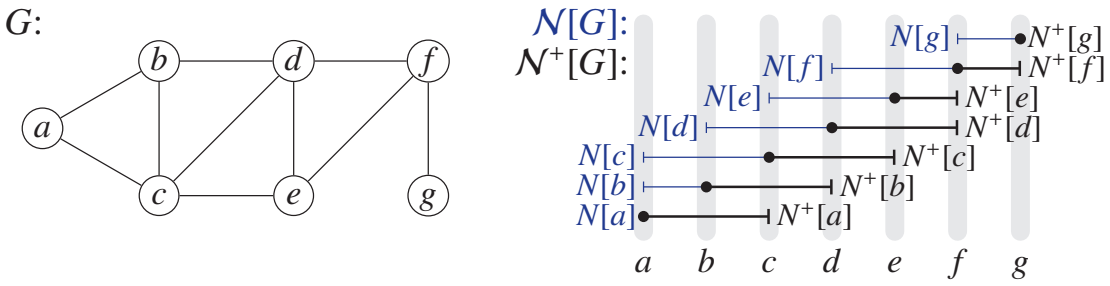


Figure 5: From an interval ordering of $\mathcal{N}[G]$ to a tight interval model $\mathcal{N}^+[G]$ of G.

As the interval order of $\mathcal{N}[G]$ can be computed in logspace by Theorem 3.6, Lemma 3.10 implies that we can easily compute canonical labelings for the connected components of a given proper interval graph G and combine them to a canonical labeling of the whole graph in a straightforward way.

This proves the following theorem.

Theorem 3.11 ([49, Theorem 6.3])**.** *The canonical representation problem for proper interval graphs can be solved in logspace.*

A graph is a *unit interval graph* if it has an interval model over rationals in which every interval has unit length. It is well known [67] that the class of proper interval graphs is equal to the class of unit interval graphs. Corneil et al. [20] show that unit interval representations of proper interval graphs can be computed in linear time. Based on their methods, it has been shown in [49] that this task can also be performed in logspace.

3.3 Circular-arc graphs

Though circular-arc graphs may at first glance appear close relatives of interval graphs, essential differences between the two classes are well known. For example, an interval graph has at most n maxcliques, and we used a succinct representation for each of them given by Lemma 3.1. For circular-arc graphs this is no longer possible, because these graphs can have exponentially many maxcliques; see Fig. 6 for an example. Note also that, unlike interval graphs, currently there is no characterization of the class of circular-arc graphs in terms of forbidden induced subgraphs; see [55] for an overview of circular-arc graphs and subclasses. These facts may serve as some excuse for the status of GI for circular-arc graphs staying open: Recently, Curtis et al. [21] published a counter-example to Hsu's algorithm [41], raising the following question.

Problem 3.12. *Is the isomorphism problem for circular-arc graphs in* P*?*

Furthermore, proper interval and proper circular-arc graphs also show structural distinctions. For example, while every proper interval graph is known to be

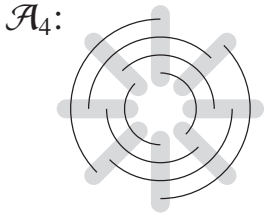


Figure 6: The complement graph G_m of m disjoint edges is circular-arc and has 2^m maxcliques. $\mathcal{A}_4$ is a circular-arc model for G_4.

representable by an intersection model consisting of unit intervals, the analogous statement for proper circular-arc graphs is not true. Another difference, very important in our context, lies in relationship to interval and circular-arc hypergraphs that we will explain shortly.

A *circular ordering* of a hypergraph $\mathcal{H}$ is a circular successor relation $\succ$ such that all hyperedges $X \in \mathcal{H}$ are consecutive points w.r.t. $\succ$. A hypergraph is *circular-arc* if it admits a circular ordering.

By Theorem 3.8, G is a proper interval graph if and only if $\mathcal{N}[G]$ is an interval hypergraph. The circular-arc world is more complex. While $\mathcal{N}[G]$ is a circular-arc hypergraph if G is a proper circular-arc graph, the converse is not always true. Proper circular-arc graphs are properly contained in the class of those graphs whose neighborhood hypergraphs are circular-arc. Graphs with this property are called *concave-round* by Bang-Jensen, Huang, and Yeo [9] and *Tucker graphs* by Chen [16]. The latter name is justified by Tucker's result [73] saying that all these graphs are circular-arc (even though not necessarily proper circular-arc). Fig. 7 shows a circular-arc graph that is not concave-round.

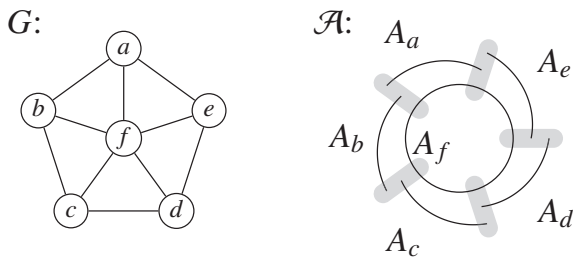
In the context of hypergraphs, however, the similarity between circular-arc and interval hypergraphs can be directly exploited, as first observed by Tucker [73]. For a circular-arc hypergraph $\mathcal{H}$ and a vertex $v \in V(\mathcal{H})$, define the hypergraph

$$\mathcal{H}_v = \{X : v \notin X \in \mathcal{H}\} \cup \{V(\mathcal{H}) \setminus X : v \in X \in \mathcal{H}\}.$$

This corresponds to complementing all hyperedges that contain v. Tucker observed that $\mathcal{H}_v$ is interval if and only if $\mathcal{H}$ is circular-arc. The following theorem is proved by iterating over all $v \in V(\mathcal{H})$ and distinguishing non-complemented and complemented hyperedges in $\mathcal{H}_v$ with two different colors.

Theorem 3.13 ([50])**.** *The canonical ordering problem for circular-arc hypergraphs can be solved in logspace.*

In [50] we use the algorithm of Theorem 3.13 as a starting point to design logspace algorithms for computing canonical proper circular-arc models of proper circular-arc graphs and canonical circular-arc models of concave-round graphs.



$\mathcal{N}[G]$	a	b	c	d	e	f
$N[a]$	1	1	0	0	1	1
$N[b]$	1	1	1	0	0	1
$N[c]$	0	1	1	1	0	1
$N[d]$	0	0	1	1	1	1
$N[e]$	1	0	0	1	1	1
$N[f]$	1	1	1	1	1	1

Figure 7: A circular-arc graph $G = \mathbb{I}(\mathcal{A})$ that is not concave-round: Its closed neighborhood hypergraph $\mathcal{N}[G]$ is not circular-arc.

4 Realizing Star Systems

The Star System Problem (to be abbreviated as *SSP*) consists in finding, for a given hypergraph $\mathcal{H}$, a graph G such that $\mathcal{H} = \mathcal{N}[G]$. We call any such graph G a *solution* to the SSP on input $\mathcal{H}$. Note that $\mathcal{H}$ can only have an SSP solution if $\mathcal{H}$ has an equal number of vertices and hyperedges. The terms *star* and *star system* are synonyms for the closed neighborhood of a vertex and the closed neighborhood hypergraph of a graph, respectively. The problem occurs in the literature also under the name *Closed Neighborhood Realization*. The question on the computational complexity of the SSP was posed by Sabidussi and Sós in the mid-70s. Shortly afterwards, Babai observed that the problem is at least as hard as GI; see [30] for a historical overview. Subsequently, Lalonde [53] showed that its decision version is in fact NP-complete.

In the complementary version of the SSP, called *co-SSP* here and also known as *Open Neighborhood Realization* problem in the literature, on input $\mathcal{H}$ we have to find a graph G with $\mathcal{N}(G) = \mathcal{H}$. Recall that the *complement* $\overline{G}$ *of a graph* G has the same vertex set as G, and two vertices are adjacent in $\overline{G}$ if and only if they are not in G. The *complement* $\overline{\mathcal{H}}$ *of a hypergraph* $\mathcal{H}$ also has the same vertex set as $\mathcal{H}$, but hyperedges complementing the hyperedges of $\mathcal{H}$, i.e., $X \in \overline{\mathcal{H}}$ if and only if $V(\mathcal{H}) \setminus X \in \mathcal{H}$. Now it is easy to verify that

$$\overline{\mathcal{N}[G]} = \mathcal{N}(\overline{G}) \text{ and } \overline{\mathcal{N}(G)} = \mathcal{N}[\overline{G}]. \tag{7}$$

Hence, finding for a given hypergraph $\mathcal{H}$ a graph G with $\mathcal{N}[G] = \mathcal{H}$ is equivalent to finding for $\overline{\mathcal{H}}$ a graph G' with $\mathcal{N}(G') = \overline{\mathcal{H}}$. Thus, the SSP and the co-SSP have the same complexity.

The following simple observation characterizes open neighborhood hypergraphs of bipartite graphs.

Lemma 4.1. *Suppose that G is a connected bipartite graph with vertex classes U and W. Then the open neighborhood hypergraph $\mathcal{N}(G)$ is split into two con-*

nected components $\mathcal{U}$ and $\mathcal{W}$, on the vertex sets U and W, respectively, such that $\mathcal{U} \cong \mathcal{W}^*$.

In the notation of the lemma, note that the incidence graphs $I(\mathcal{U})$ and $I(\mathcal{W})$ become isomorphic after interchanging the colors red and blue in one of them. Moreover, the uncolored versions of both $I(\mathcal{U})$ and $I(\mathcal{W})$ are isomorphic to G.

4.1 Case study: NP-hardness, GI-completeness, and efficient solvability

If we restrict the SSP to a particular graph class C, we only seek for a solution G in the class C. As mentioned above, the restriction of the SSP to C is equivalent to the co-SSP restricted to the co-class of C (consisting of the complements of all graphs in C).

Fomin et al. [30] study the restrictions of the SSP to H-free graphs, that is, to graph classes that are characterized by forbidding a single induced subgraph H. They show that the SSP restricted to H-free graphs remains NP-hard if H is a path or a cycle of at least 5 vertices, the claw graph, or any other graph obeying a set of conditions specified in [30].

Aigner and Triesch [1] showed that the SSP for co-bipartite graphs is equivalent to GI, provided that the bipartition of the vertices is given along with the input hypergraph $\mathcal{H}$. Boros et al. [15] observed that this remains true, if only $\mathcal{H}$ is given as input.

Theorem 4.2 ([1, 15])**.** *The SSP for co-bipartite graphs is equivalent to GI.*

Proof-sketch. We show the equivalent statement that the co-SSP for bipartite graphs is equivalent to GI. Recall that GI is equivalent with its restriction to connected graphs (because $G \cong H$ if and only if $\overline{G} \cong \overline{H}$, and if G is disconnected, then its complement $\overline{G}$ must be connected). Consider an even more general problem of deciding whether two connected hypergraphs $\mathcal{H}$ and $\mathcal{K}$ are isomorphic. By (4) and Lemma 4.1,

$$\mathcal{H} \cong \mathcal{K} \iff \mathcal{H}^* \cong \mathcal{K}^* \iff \mathcal{H} \cup \mathcal{K}^* = \mathcal{N}(G) \text{ for a bipartite graph } G,$$

and hence, the reduction $(\mathcal{H}, \mathcal{K}) \mapsto \mathcal{H} \cup \mathcal{K}^*$ shows that the co-SSP for bipartite graphs is at least as hard as (hyper)graph isomorphism.

In order to show a reduction in the other direction, assume first that $\mathcal{H}$ consists of two connected components $\mathcal{U}$ and $\mathcal{W}$. By Lemma 4.1,

$$\mathcal{H} = \mathcal{N}(G) \text{ for a bipartite graph } G \iff \mathcal{U} \cong \mathcal{W}^*,$$

giving the desired reduction of the co-SSP for bipartite graphs to hypergraph isomorphism. Moreover, we can also compute a solution G to the co-SSP instance $\mathcal{H}$, since G is isomorphic to the incidence graph $G' = I(\mathcal{U}) \cong I(\mathcal{W})$, where the red-blue coloring is disregarded. In order to compute G, it suffices to establish an isomorphism π from $\mathcal{N}(G')$ to $\mathcal{H}$ and take the image of G' under π.

In general, $\mathcal{H} = \mathcal{N}(G)$ for a bipartite G if and only if the components of $\mathcal{H}$ can be arranged into pairs $\mathcal{U}_1, \mathcal{W}_1, \ldots, \mathcal{U}_m, \mathcal{W}_m$ such that $\mathcal{U}_i \cong \mathcal{W}_i^*$. Thus, the co-SSP for bipartite graphs is no harder than GI. ∎

In several cases the SSP is known to be efficiently solvable. Polynomial-time algorithms are worked out for H-free graphs with H being a cycle or a path on at most 4 vertices (Fomin et al. [30]) and for bipartite graphs (Boros et al. [15]). In [50] we give a logspace solution for the SSP for proper circular-arc and concave-round graphs. An analysis of the algorithms in [30] for C_3- and C_4-free graphs shows that the SSP for these classes is also solvable in logspace, and the same holds true for the class of bipartite graphs.

C_3-free graphs and bipartite graphs. The approach of Fomin et al. [30] to C_3-free graphs is based on the following observation. If G is C_3-free, then for any pair of vertices u and v adjacent in G there are exactly two hyperedges X and Y in $\mathcal{N}[G]$ containing both u and v. Moreover, $N[v] \in \{X, Y\}$ and the assumption that $N[v] = X$ forces the equality $N[u] = Y$.

Let us show how to derive from here the logspace solvability of the SSP for C_3-free graphs. Since the composition of logspace computable functions is logspace computable, we can split the whole algorithm into a few steps, each doable in logspace. We can assume that the input hypergraph $\mathcal{H}$ is connected; otherwise we apply the procedure below to each of its components. We first construct an auxiliary graph F. The vertices of F are all pairs (v, X) such that $v \in X \in \mathcal{H}$. Two vertices (v, X) and (u, Y) are adjacent in F if and only if $v \neq u$, $X \neq Y$, and X and Y are the only two hyperedges of $\mathcal{H}$ containing both v and u.

Fix an arbitrary vertex v of $\mathcal{H}$. For each vertex of the form (v, X) of F, we now try to construct a vertex-hyperedge assignment A_X as follows. Assign X to v. To each other u we assign an Y such that (u, Y) is reachable from (v, X) along a path in F. At this step we use the Reingold reachability algorithm [66]. For some u, the choice of Y may be impossible or ambiguous.

For each successfully constructed one-to-one assignment A_X, we then try to construct a graph G_X by connecting each u with all other vertices in the assigned hyperedge Y. For each successfully constructed G_X, it remains to check if $\mathcal{N}[G_X] = \mathcal{H}$ and if G_X is C_3-free. We will succeed at least once, unless the SSP on $\mathcal{H}$ has no C_3-free solution. This completes the description of the algorithm.

Note a useful fact that follows from the above discussion: If a hypergraph $\mathcal{H}$

is connected, then for any hyperedge $X \in \mathcal{H}$ and vertex $v \in X$ there is at most one C_3-free graph G such that $\mathcal{H} = \mathcal{N}[G]$ and $X = N[v]$. Thus, the SSP on $\mathcal{H}$ has at most $\min_{X \in \mathcal{H}} |X|$ triangle-free solutions, and all of them can be computed in logspace. It readily follows that the SSP is solvable in logspace for any logspace recognizable class consisting of C_3-free graphs. In particular, this applies to the class of bipartite graphs.

C_4-free graphs. The algorithm of Fomin et al. [30] for C_4-free graphs is implementable in logspace in a straightforward way. It is based on the following argument.

Suppose that G is C_4-free. Given two vertices u and v in G, let $X_1, \ldots, X_t$ be all hyperedges in $\mathcal{N}[G]$ containing both u and v. If u and v are adjacent, then

$$2 \le \left|\bigcap_{i=1}^{t} X_i\right| \le t. \tag{8}$$

This follows from the observation that u and v have exactly $t-2$ common neighbors, and every vertex in $\bigcap_{i=1}^{t} X_i \subseteq N[u] \cap N[v]$ must be one of them or one of u and v.

If u and v are not adjacent, then

$$t = 0 \ \text{ or } \ \left|\bigcap_{i=1}^{t} X_i\right| \ge t + 2.$$

Indeed, in this case u and v have exactly t common neighbors. Let $t > 0$. By the assumption that G is C_4-free, these t vertices form a clique. Therefore, $\bigcap_{i=1}^{t} X_i$ contains all of them as well as u and v themselves.

Thus, the graph G can be reconstructed from the hypergraph $\mathcal{H} = \mathcal{N}[G]$ by joining two vertices u and v by an edge whenever the condition (8) is true for this pair. Solving the SSP on an input $\mathcal{H}$, we first construct G by this rule and then check if $\mathcal{H} = \mathcal{N}[G]$ and if G is C_4-free. In the case of failure, no solution among C_4-free graphs exists.

Proper interval graphs. As we will discuss in more detail in Section 4.2, the SSP for proper interval graphs is solvable in logspace because these graphs form a logspace-recognizable subclass of C_4-free graphs. We now outline a different argument exemplifying our approach from [50] to the SSP for the broader classes of proper circular-arc and concave-round graphs.

Three important ingredients of our argument already appeared in Section 3.1. By Theorem 3.8, G is a proper interval graph if and only if $\mathcal{N}[G]$ is an interval hypergraph, i.e., this hypergraph admits an interval order of its vertices. By Lemma 3.10, if G is, moreover, connected, then such an interval order is unique (up to reversing and up to permutation of twins). The interval order of $\mathcal{N}[G]$ can

be computed in logspace by Theorem 3.6. We now state another key element of our analysis. Given a linear order $<$ on a set V, we introduce the linear order $<^*$ on the set of all intervals in V by comparing the endpoints of intervals lexicographically w.r.t. $<$.

Lemma 4.3 (cf. [50, Lemma 5.8.1])**.** *Suppose that a graph G (and hence $\mathcal{N}[G]$) is twin-free. If $<$ is an interval order for $\mathcal{N}[G]$, then*

$$u < v \iff N[u] <^* N[v]. \tag{9}$$

Putting it together, we come to the following logspace algorithm for the SSP for proper interval graphs on input hypergraph $\mathcal{H}$. We first consider the case that $\mathcal{H}$ is twin-free.

Compute an interval order $<$ for $\mathcal{H}$. If this fails, no solution among proper interval graphs exists; otherwise, any solution will be surely a proper interval graph.

Next, sort the hyperedges of $\mathcal{H}$ according to the lexicographic order $<^*$. The equivalence (9) allows us to establish the v-to-$N[v]$ correspondence, that is, for each hyperedge $X \in \mathcal{H}$, to find a vertex v such that $N[v] = X$ (assuming that a solution G to the SSP on $\mathcal{H}$ exists).

Finally, we have to check that this correspondence really defines a graph, that is, whenever two vertices v and v' receive hyperedges X and X' as their neighborhoods, we have to check that $v \in X$ and that $v \in X'$ if and only if $v' \in X$. If this is not true, the SSP on input $\mathcal{H}$ has no solution.

The general case, when $\mathcal{H}$ may have twins, reduces to the twin-free case by considering the *quotient-hypergraph* $\mathcal{H}'$ w.r.t. the equivalence relation of being twins, where the vertices are the twin-classes of $\mathcal{H}$, and a set of twin-classes is a hyperedge in $\mathcal{H}'$ if and only if the union of these twin-classes is a hyperedge in $\mathcal{H}$.

4.2 Uniqueness of a solution

The argument employed in the proof of Theorem 4.2 leads us to the following observation: If we know that a graph is bipartite, then it is reconstructible from its open neighborhood hypergraph up to isomorphism. More precisely, if two graphs G and H are both bipartite, then the equality $\mathcal{N}(G) = \mathcal{N}(H)$ implies the isomorphism $G \cong H$. (See Fig. 8 below for an example of an hypergraph that is the open neighborhood hypergraph of a bipartite and of a non-bipartite graph.) Equivalently, if G and H are both co-bipartite, then

$$\mathcal{N}[G] = \mathcal{N}[H] \implies G \cong H. \tag{10}$$

In other words, any instance of the SSP for co-bipartite graphs has at most one solution up to isomorphism. The argument of Fomin et al. [30] presented above leads to the same conclusion in the case that both G and H are C_4-free. Moreover, in this case the equality $\mathcal{N}[G] = \mathcal{N}[H]$ even implies the equality $G = H$. For a smaller class of chordal graphs this was observed earlier by Harary and McKee [38]. Due to Boros et al. [15], the implication (10) is also known to be true if both G and H are bipartite.

Chen [16, 18] showed an even stronger fact for any concave-round graph G:

$$\text{for } \textit{any} \text{ graph } H, \quad \mathcal{N}[G] = \mathcal{N}[H] \implies G \cong H. \tag{11}$$

In other words, each concave-round graph is reconstructible from its closed neighborhood hypergraph up to isomorphism. Earlier such a reconstructibility result was shown for complements of forests by Aigner and Triesch [1].

Our treatment of the SSP for proper interval graphs reveals a fact that is yet stronger than (11).

Corollary 4.4. *Let G be a proper interval graph. Then, for any graph H,*

$$\mathcal{N}[G] = \mathcal{N}[H] \implies G = H.$$

In this stronger form, the reconstructibility from the closed neighborhood hypergraph was earlier known only for complete graphs; see Aigner and Triesch [1].

The implication (10) can be rephrased as the equivalence of the isomorphisms $G \cong H$ and $\mathcal{N}[G] \cong \mathcal{N}[H]$. This provides the shortest way to testing isomorphism of concave-round graphs in logspace, if we do not care of coming up with a canonical arc model. Given concave-round graphs G and H, it suffices to compute the canons of $\mathcal{N}[G]$ and $\mathcal{N}[H]$ by the algorithm of Theorem 3.13 and to check if they are equal.

Moreover, the implication (10) has important consequences for the SSP. In general, the logspace solvability of the SSP for a class of graphs $\mathcal{C}$ does still not imply the logspace solvability of the SSP for any subclass $\mathcal{C}'$ of $\mathcal{C}$. However, it does if $\mathcal{C}'$ is recognizable in logspace and (10) holds true for all G and H in $\mathcal{C}$. This observation applies to the classes of chordal, interval, and proper interval graphs, which are subclasses of C_4-free graphs. Each of these classes is recognizable in logspace by Reif [65] in combination with Reingold [66] or by methods of [49]. Therefore, the results of Fomin et al. [30] about C_4-free graphs imply that the SSP for the classes of chordal, interval, and proper interval graphs is solvable in logspace.

While the case of interval graphs is therewith efficiently solvable, note that the complexity status of the SSP for circular-arc graphs remains open.

Problem 4.5. *Is the SSP for circular-arc graphs solvable by a poly-time algorithm?*

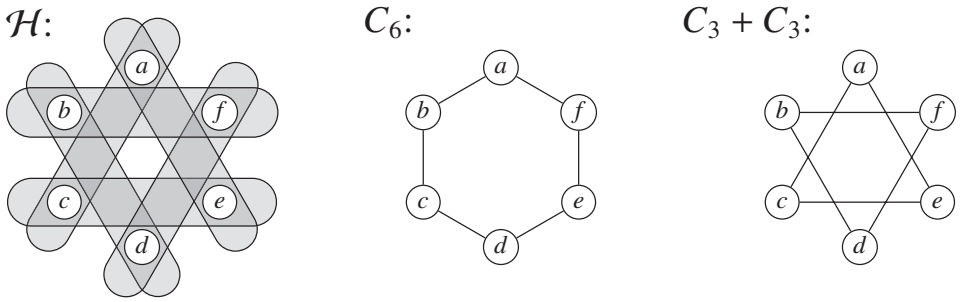


Figure 8: The open neighborhood hypergraph $\mathcal{H}$ of the two non-isomorphic graphs C_6 and $C_3 + C_3$.

By Corollary 4.4 and Theorem 3.8, the SSP on a given interval hypergraph has either none or exactly one solution. In general, the problem can have different solutions. For example, on a given set of 4 vertices we can draw a cycle C_4 in 3 different ways, and all three graphs will have the same closed neighborhood hypergraph. Moreover, the SSP can even have non-isomorphic solutions. This is especially easy to see after switching to the co-SSP. Fig. 8 shows an example of two non-isomorphic graphs with the same open neighborhood hypergraph.

This is an instance of the following general construction in Aigner and Triesch [1]. Given an arbitrary graph G, take two copies of its vertex set, $V = \{v_1, \ldots, v_n\}$ and $V' = \{v'_1, \ldots, v'_n\}$, and define two graphs on the $2n$ vertices. Let $G + G$ consist of two disjoint copies of G, one on V and the other on V'. Furthermore, let $G \times G$ be a bipartite graph with vertex classes V and V', where v_i and v'_j are adjacent if and only if v_i and v_j are adjacent in G. Then $\mathcal{N}(G+G) = \mathcal{N}(G\times G)$.

Call a hypergraph $\mathcal{H}$ *uniquely realizable* if there is a unique G such that $\mathcal{H} = \mathcal{N}[G]$, that is, the SSP has a unique solution on $\mathcal{H}$. Thus, any realizable interval hypergraph is uniquely realizable.

The recognition problem of uniquely realizable hypergraphs belongs to the complexity class US (abbreviated from *Unique Solution*) introduced by Blass and Gurevich [11].

Problem 4.6. *Is the unique realizability problem* US*-complete?*

A related hardness result is obtained by Aigner and Triesch [1]: Given a connected bipartite graph G, deciding whether or not $\mathcal{N}(G) = \mathcal{N}(H)$ for some $H \ncong G$ is NP-complete.

References

[1] M. Aigner and E. Triesch. Reconstructing a graph from its neighborhood lists. *Combinatorics, Probability & Computing*, 2:103–113, 1993.

[2] V. Arvind, B. Das, J. Köbler, and S. Kuhnert. The isomorphism problem for k-trees is complete for logspace. *Information and Computation*, 217:1–11, 2012.

[3] V. Arvind, B. Das, J. Köbler, and S. Toda. Colored hypergraph isomorphism is fixed parameter tractable. In *Proc. 30th FSTTCS*, volume 8 of *LIPIcs*, pages 327–337, Dagstuhl, 2010. Leibniz-Zentrum für Informatik.

[4] V. Arvind and J. Torán. Isomorphism testing: Perspective and open problems. *Bulletin of the EATCS*, 86:66–84, 2005.

[5] L. Babai and P. Codenotti. Isomorhism of hypergraphs of low rank in moderately exponential time. In *Proc. of the 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 667–676. IEEE Computer Society, 2008.

[6] L. Babai and E. M. Luks. Canonical labeling of graphs. In *Proceedings of the 15-th Annual ACM Symposium on Theory of Computing*, pages 171–183, 1983.

[7] L. Babel and S. Olariu. On the isomorphism of graphs with few P_4s. In M. Nagl, editor, *Proceedings of the 21st International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 1017 of *LNCS*, pages 24–36. Springer, 1995.

[8] L. Babel, I. N. Ponomarenko, and G. Tinhofer. The isomorphism problem for directed path graphs and for rooted directed path graphs. *J. Algorithms*, 21(3):542–564, 1996.

[9] J. Bang-Jensen, J. Huang, and A. Yeo. Convex-round and concave-round graphs. *SIAM J. Discrete Math.*, 13(2):179–193, 2000.

[10] S. Benzer. On the topology of the genetic fine structure. *Proceedings of the National Academy of Sciences of the United States of America*, 45(11):1607–1620, 1995.

[11] A. Blass and Y. Gurevich. On the unique satisfiability problem. *Information and Control*, 55(1–3):80–88, 1982.

[12] H. L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *J. Algorithms*, 11(4):631–643, 1990.

[13] K. Booth and C. Colbourn. Problems polynomially equivalent to Graph Isomorphism. Technical Report CS-77-04, Comp. Sci. Dep., Univ. Waterloo, 1979.

[14] K. Booth and G. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976.

[15] E. Boros, V. Gurvich, and I. E. Zverovich. Neighborhood hypergraphs of bipartite graphs. *Journal of Graph Theory*, 58(1):69–95, 2008.

[16] L. Chen. Graph isomorphism and identification matrices: Parallel algorithms. *IEEE Trans. Parallel Distrib. Syst.*, 7(3):308–319, 1996.

[17] L. Chen. Graph isomorphism and identification matrices: Sequential algorithms. *J. Comput. Syst. Sci.*, 59(3):450–475, 1999.

[18] L. Chen. A selected tour of the theory of identification matrices. *Theor. Comput. Sci.*, 240(2):299–318, 2000.

[19] L. Chen and Y. Yesha. Parallel recognition of the consecutive ones property with applications. *J. Algorithms*, 12(3):375–392, 1991.

[20] D. G. Corneil, H. Kim, S. Natarajan, S. Olariu, and A. P. Sprague. Simple linear time recognition of unit interval graphs. *Inform. Proc. Lett.*, 55:99–104, 1995.

[21] A. Curtis, M. Lin, R. McConnell, Y. Nussbaum, F. Soulignac, J. Spinrad, and J. Szwarcfiter. Isomorphism of graph classes related to the circular-ones property. *E-print*, *http://arxiv.org/abs/1203.4822v1*, 2012.

[22] B. Das, J. Torán, and F. Wagner. Restricted space algorithms for isomorphism on bounded treewidth graphs. In J.-Y. Marion and T. Schwentick, editors, *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science*, volume 5 of *LIPIcs*, pages 227–238, Dagstuhl, 2010. Leibniz-Zentrum für Informatik.

[23] S. Datta, N. Limaye, P. Nimbhorkar, T. Thierauf, and F. Wagner. Planar Graph Isomorphism is in Log-Space. In *Proceedings of the 24th Annual IEEE Conference on Computational Complexity*, pages 203–214. IEEE Computer Society, 2009.

[24] S. Datta, P. Nimbhorkar, T. Thierauf, and F. Wagner. Graph Isomorphism for $K_{3,3}$-free and K_5-free graphs is in Log-Space. In R. Kannan and K. N. Kumar, editors, *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4 of *LIPIcs*, pages 145–156, Dagstuhl, 2009. Leibniz-Zentrum für Informatik.

[25] X. Deng, P. Hell, and J. Huang. Linear-time representation algorithms for proper circular-arc graphs and proper interval graphs. *SIAM J. Comput.*, 25(2):390–403, 1996.

[26] M. Dom. Algorithimic aspects of the consecutive-ones property. *Bulletin of the EATCS*, 98:27–59, 2009.

[27] P. Duchet. Classical perfect graphs. An introduction with emphasis on triangulated and interval graphs. Perfect graphs, Ann. Discrete Math. 21, 67–96 (1984)., 1984.

[28] S. Evdokimov and I. N. Ponomarenko. Isomorphism of coloured graphs with slowly increasing multiplicity of jordan blocks. *Combinatorica*, 19(3):321–333, 1999.

[29] I. Filotti and J. N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus (working paper). In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 236–243, 1980.

[30] F. V. Fomin, J. Kratochvíl, D. Lokshtanov, F. Mancini, and J. A. Telle. On the complexity of reconstructing H-free graphs from their Star Systems. *Journal of Graph Theory*, 68(2):113–124, 2011.

[31] G. Gati. Further annotated bibliography on the isomorphism disease. *J. Graph Theory*, 3:95–109, 1979.

[32] F. Gavril. Algorithms on circular-arc graphs. *Networks*, 4:357–369, 1974.

[33] M. C. Golumbic. *Algorithmic graph theory and perfect graphs. 2nd ed.* Amsterdam: Elsevier, 2004.

[34] M. Grohe. Isomorphism testing for embeddable graphs through definability. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 63–72, 2000.

[35] M. Grohe. From polynomial time queries to graph structure theory. *Commun. ACM*, 54(6):104–112, 2011.

[36] M. Grohe and D. Marx. Structure theorem and isomorphism test for graphs with excluded topological subgraphs. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing*, 2012. To appear. Preprint at *http://arxiv.org/abs/1111.1109.*

[37] M. Grohe and O. Verbitsky. Testing graph isomorphism in parallel by playing a game. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming, Part I*, volume 4051 of *LNCS*, pages 3–14. Springer, 2006.

[38] F. Harary and T. A. McKee. The square of a chordal graph. *Discrete Mathematics*, 128(1–3):165–172, 1994.

[39] J. Hopcroft and R. Tarjan. A V^2 algorithm for determining isomorphism of planar graphs. *Inf. Process. Lett.*, 1:32–34, 1971.

[40] J. E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing*, pages 172–184, 1974.

[41] W.-L. Hsu. $O(m \cdot n)$ isomorphism algorithms for circular-arc graphs and circle graphs. In R. Kannan and W. R. Pulleyblank, editors, *Proceedings of the 1st Integer Programming and Combinatorial Optimization Conference*, pages 297–311. University of Waterloo Press, 1990.

[42] W.-L. Hsu and R. M. McConnell. PC trees and circular-ones arrangements. *Theoretical Computer Science*, 296(1):99–116, 3 2003.

[43] B. Jenner, J. Köbler, P. McKenzie, and J. Torán. Completeness results for graph isomorphism. *J. Comput. Syst. Sci.*, 66(3):549–566, 2003.

[44] B. L. Joeris, M. C. Lin, R. M. McConnell, J. P. Spinrad, and J. L. Szwarcfiter. Linear time recognition of Helly circular-arc models and graphs. *Algorithmica*, 59(2):215–239, 2 2011.

[45] K.-I. Kawarabayashi and B. Mohar. Graph and map isomorphism and all polyhedral embeddings in linear time. In *Proc. of the 40th Ann. ACM Symp. on Theory of Computing*, pages 471–480, 2008.

[46] M. Klawe, D. Corneil, and A. Proskurowski. Isomorphism testing in hookup classes. *SIAM J. Algebraic Discrete Methods*, 3:260–274, 1982.

[47] P. N. Klein. Efficient parallel algorithms for chordal graphs. *SIAM J. Comput.*, 25(4):797–827, 1996.

[48] J. Köbler. On graph isomorphism for restricted graph classes. In A. Beckmann, U. Berger, B. Löwe, and J. V. Tucker, editors, *Logical Approaches to Computational Barriers, Proceedings of the 2nd Conference on Computability in Europe*, volume 3988 of *LNCS*, pages 241–256. Springer, 2006.

[49] J. Köbler, S. Kuhnert, B. Laubner, and O. Verbitsky. Interval graphs: Canonical representations in Logspace. *SIAM J. on Computing*, 40(5):1292–1315, 2011.

[50] J. Köbler, S. Kuhnert, and O. Verbitsky. Solving the canonical representation and star system problem for proper circular-arc graphs in logspace. *E-print*, *http://arxiv.org/abs/1202.4406*, 2012.

[51] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Progress in Theoretical Computer Science. Birkhäuser, 1993.

[52] S. Kratsch and P. Schweitzer. Isomorphism for graphs of bounded feedback vertex set number. In H. Kaplan, editor, *Proc. of the 12th Scandinavian Symposium and Workshops on Algorithm Theory*, volume 6139 of *LNCS*, pages 81–92. Springer, 2010.

[53] F. Lalonde. Le probleme d'etoiles pour graphes est NP-complet. *Discrete Mathematics*, 33(3):271–280, 1981.

[54] B. Laubner. Capturing polynomial time on interval graphs. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science*, 2010.

[55] M. C. Lin and J. L. Szwarcfiter. Characterizations and recognition of circular-arc graphs and subclasses: A survey. *Discrete Mathematics*, 309(18):5618–5635, 2009.

[56] S. Lindell. A logspace algorithm for tree canonization. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 400–404, 1992.

[57] G. Lueker and K. Booth. A linear time algorithm for deciding interval graph isomorphism. *J. ACM*, 26(2):183–195, 1979.

[58] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25(1):42–65, 1982.

[59] E. M. Luks. Hypergraph isomorphism and structural equivalence of boolean functions. In J. S. Vitter, L. L. Larmore, and F. T. Leighton, editors, *Proc. of the 31st Ann. ACM Symposium on Theory of Computing*, pages 652–658. ACM, 1999.

[60] G. L. Miller. Isomorphism testing for graphs of bounded genus. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 225–235, 1980.

[61] G. L. Miller and J. H. Reif. Parallel tree contraction. Part 2: Further applications. *SIAM J. Comput.*, 20(6):1128–1147, 1991.

[62] J. Moon and L. Moser. On cliques in graphs. *Isr. J. Math.*, 3:23–28, 1965.

[63] I. Ponomarenko. The isomorphism problem for classes of graphs that are invariant with respect to contraction. In *Zap. Nauchn. Sem. Leningrad. Otdel. Mat. Inst. Steklov., 174 (Teor. Slozhn. Vychisl. 3)*, pages 147–177. LOMI, 1988. Translation from Russian in *J. Soviet Math.* 55(2):1621–1643 (1991).

[64] R. C. Read and D. G. Corneil. The graph isomorphism disease. *J. Graph Theory*, 1:339–363, 1977.

[65] J. Reif. Symmetric complementation. *J. ACM*, 31(2):401–421, 1984.

[66] O. Reingold. Undirected connectivity in log-space. *J. ACM*, 55(4), 2008.

[67] F. Roberts. Indifference graphs. Proof Tech. Graph Theory, Proc. 2nd Ann Arbor Graph Theory Conf. 1968, 139-146 (1969)., 1969.

[68] P. Schweitzer. Isomorphism of (mis)labeled graphs. In *Proc. of the 19th Ann. European Symposium on Algorithms*, volume 6942 of *LNCS*, pages 370–381. Springer, 2011.

[69] J. Spinrad. *Efficient graph representations*. Number 19 in Field Institute Monographs. AMS, 2003.

[70] S. Toda. Computing automorphism groups of chordal graphs whose simplicial components are of small size. *IEICE Transactions*, 89-D(8):2388–2401, 2006.

[71] J. Torán. On the hardness of graph isomorphism. *SIAM J. Comput.*, 33(5):1093–1108, 2004.

[72] J. Torán and F. Wagner. The complexity of planar graph isomorphism. *Bulletin of the EATCS*, 97:60–82, 2009.

[73] A. Tucker. Matrix characterizations of circular-arc graphs. *Pac. J. Math.*, 39:535–545, 1971.

[74] R. Uehara. Simple geometrical intersection graphs. In S.-I. Nakano and M. S. Rahman, editors, *Proceedings of the 2nd International Workshop on Algorithms and Computation*, volume 4921 of *LNCS*, pages 25–33. Springer, 2008.

[75] R. Uehara, S. Toda, and T. Nagoya. Graph isomorphism completeness for chordal bipartite graphs and strongly chordal graphs. *Discrete Applied Mathematics*, 145(3):479–482, 2005.

[76] K. Yamazaki, H. L. Bodlaender, B. de Fluiter, and D. M. Thilikos. Isomorphism for graphs of bounded distance width. *Algorithmica*, 24(2):105–127, 1999.

[77] V. Zemlyachenko, N. Kornienko, and R. Tyshkevich. Graph isomorphism problem. *J. Sov. Math.*, 29:1426–1481, 1985.

T D C C

P F

Department of Computer Science, University of Crete
P.O. Box 2208 GR-714 09 Heraklion, Crete, Greece
and
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
N. Plastira 100. Vassilika Vouton
GR-700 13 Heraklion, Crete, Greece
faturu@csd.uoc.gr

Programming with SpecTM

Tim Harris
Microsoft Research
tharris@microsoft.com

Aleksandar Dragojević
I&C, EPFL
aleksandar.dragojevic@epfl.ch

Abstract

In this paper we examine the use of "mini" transactions. An implementation of mini-transactions supports small sequences of memory accesses as atomic transactions (perhaps 1–4 accesses). When building a shared memory data structure using mini transactions, the programmer must either stay within the limits of a single mini transaction, or split the operation across a series

Bulletin of the EATCS no 107, pp. 72–94, June 2012

of mini transactions. Mini transactions therefore provide a greater degree of atomicity than single-word compare and swap (CAS), but they do not provide the full features of a general-purpose transactional memory (TM). We illustrate how hashtables and skip lists can be built over the SpecTM API for mini transactions. We discuss the advantages and disadvantages of the SpecTM API over a general-purpose TM. To address some of these limitations, we discuss techniques for integrating SpecTM with a general-purpose STM.

1 Introduction

In this paper we examine the use of "mini" transactions to implement shared-memory hashtables and skip lists. In this approach, an operation on a data structure is split across a series of short transactions, rather than using a single transaction encompassing the entire operation. There are several reasons for studying the use of mini transactions:

First, practical implementations of hardware transactional memory (HTM [17]) limit the size of transactions. Some proposals, such as AMD ASF [5], provide a guarantee that a transaction that follows various programming rules will be able to commit eventually if it accesses only 1–4 locations. Recent HTM implementations do not provide such a guarantee [7, 22]. However, irrespective of whether or not a guarantee is given, it is likely that shorter transactions will be more likely to commit than longer ones.

The second reason for studying mini transactions is that, in recent work, we showed that implementations of data structures using mini transactions can perform well [10]. We showed how parts of the implementation could be specialized in cases such as transactions that access a small fixed number of locations. Our preliminary results suggested that data structures built using this system were much faster and more scalable than those built using a general-purpose STM system (we used one based on SwissTM [9] and the STM described by Spear *et al.* [25]).

Finally, Attiya showed recently that, under various assumptions, a series of short transactions can be more efficient than a single long transaction [1]. Attiya's work derived lower bounds on the operations that need to be performed by an STM implementation, given assumptions about the progress-properties and safety-properties that the STM should satisfy.

In Section 2 we review the SpecTM API and describe, in outline, the implementation techniques that we use. We designed the SpecTM API to let us streamline much of a traditional STM system's book-keeping—the result is an API that is more cumbersome to use than a general-purpose STM, but which still provides

```
// Transaction management operations
void Tx_Start(TX_RECORD *t);
void Tx_Abort(TX_RECORD *t);
bool Tx_Commit(TX_RECORD *t);
bool Tx_Validate(TX_RECORD *t);

// Data access operations
Ptr Tx_Read(TX_RECORD *t, Ptr *addr);
void Tx_Write(TX_RECORD *t, Ptr *addr, Ptr val);
```

Figure 1: A traditional STM interface (pseudo-code).

the key abstraction of multi-word atomic memory accesses.

In Sections 3–4 we illustrate the use of SpecTM in practice in implementing a hashtable and a skip list. We take an informal approach in this paper. We aim to illustrate the use of SpecTM through examples, but we do not attempt to quantify exactly how much easier SpecTM is to use than single-word CAS, or exactly how much more difficult SpecTM is to use than a general-purpose TM. Based on these examples, we discuss the difficulties that we have encountered in using the SpecTM API (Section 5).

In Section 6 we discuss how SpecTM can be integrated with general-purpose TM. An advantage of such integration is that SpecTM can be used to accelerate the common cases in a data structure's implementation. General-purpose TM can be used as a fall-back for cases whose performance is not critical, or for cases which appear impractical to split into mini transactions. The main challenge is integrating the conflict detection algorithms used in different TM systems—some techniques from existing software-hardware hybrid TM systems can be applied.

2 Programming Models

Figure 1 sketches the kind of interface typically exposed by general-purpose STM systems. There are operations to start transactions, abort transactions, and to commit them. If `Tx_Commit` returns `true` then we say that the transaction has "succeeded", and its effects appear to take place atomically at some point during its execution. Otherwise, `Tx_Commit` returns `false`, we say that the transaction has "failed", and the transaction's effects are not made visible to other threads. There is a validation operation to detect whether or not a transaction has already experienced a conflict. There are operations to read a memory location, and to update a memory location with a new value. (For brevity we focus on an interface in which all of the locations read and written contain pointers.)

Note that, throughout the paper, we study implementations that provide only weak isolation [3], meaning that there is no conflict detection between transactional and non-transactional memory accesses. This is sufficient to implement

programming models that distinguish between transactional and non-transaction locations (e.g., STM-Haskell [14]). Alternative programming models are often proposed, such as ones that allow privatization idioms. Existing mechanisms could be used to support such models over the basic systems we describe here (Harris *et al.* survey many techniques [15]).

Although TM interfaces such as Figure 1 are widespread, they can be seen as one option in a broader set of alternative abstractions for writing atomic operations on shared memory. One possible characterization of these abstractions is in terms of the following properties:

- *Size*—Are operations of unbounded size permitted, or is there a maximum size? General-purpose TM is unbounded, as are multi-word compare-and-swap operations (CASN). "Strong" LL/SC is unbounded. A CAS operation has a bound of 1. Practical implementations of LL/SC have a bound of 1. DCAS has a bound of 2.

- *Dynamic access*—Can the locations to access be selected dynamically: i.e., selecting the next location to access based on the values seen in previous locations. Recent STM systems support dynamic accesses. CAS, DCAS, and CASN, support only static accesses—that is, the entire set of locations to access is supplied as a parameter to the operation. Shavit and Touitou's original STM supported only static accesses [24].

- *Inconsistency hidden*—Does the programmer have to consider the possibility of seeing a mutually inconsistent view of a set of locations? Alternatively, does the abstraction provide a property such as opacity [12] or TMS1 [8] that precludes this? HTM designs typically hide inconsistency. Many STM designs do, whereas others do not. The question does not arise with many CAS, DCAS, and CASN abstractions which provide only a success/failure response, rather than a snapshot of the locations accessed.

- *Fallback required*—Can the programmer use the abstraction without needing to write alternative code using a different abstraction? Best-effort HTM systems do not guarantee that any transaction will ever commit successfully (even if the transaction is short and does not experience contention). Consequently, an alternative code path is needed—e.g., based on locking, or based on STM. Typical STM systems do not need a fallback code path.

Figure 2 compares the properties of various practical implementations of programming abstractions along these axes. Concretely, for HTM, we consider a best-effort system. Note that, the size in this table is listed as unbounded (because the API does not prevent unbounded-size transactions from being written), but a

	CAS	DCAS	CASN	STM	Best-effort HTM	SpecTM
Size	1	2	*n*	*n*	*n*	4
Dynamic access	n/a	Static	Static	Dynamic	Dynamic	Dynamic
Inconsistency hidden	n/a	n/a	n/a	Usually	Yes	No
Fallback required	No	No	No	No	Yes	No

Figure 2: Comparison of typical implementations of different abstractions.

fallback is required because a best-effort implementation is not required to be able to run any specific size of transaction successfully. Figure 2 also characterizes the behavior of our SpecTM system for writing mini transactions.

Unlike general-purpose STM, SpecTM supports only a limited number of memory accesses within a single transaction (4, in the current implementation). Unlike CASN, it provides a dynamic interface.

Unlike many STM implementations, SpecTM exposes inconsistent views of memory to the programmer. To prevent possible problems that could result from execution with inconsistent reads, SpecTM includes functions that allow the programmer to explicitly check for the inconsistencies if needed. Furthermore, some implementation strategies, such as write-locking on reads in short read-write transactions, might guarantee consistency of reads for subsets of the SpecTM API [10].

Unlike best-effort HTM, or bounded-sized HTM, a program using SpecTM does not require a fallback path.

2.1 SpecTM

Our earlier paper expands on the rationale for designing SpecTM, and for providing this combination of features [10]. In outline, the overriding goal is to help us build high-performance implementations on current multi-socket multi-core shared-memory machines. Figure 3 shows the current SpecTM API:

Transactionally-managed locations are held in `TmPtr` structures. Section 2.2 discusses how different SpecTM implementations use different concrete representations for a `TmPtr`. However, from the programmer's viewpoint, a `TmPtr` encapsulates a pointer-typed value.

The `Tx_Single_*` functions perform transactions that access a single location: either read, write, or compare-and-swap. These accesses synchronize correctly with concurrent SpecTM transactions. Using a separate interface for these operations allows the implementation to optimize this frequent special case (e.g., avoiding initializing a transaction record).

The `Tx_RW_R*` operations are used for transactions that read from a series of locations, and then write new values to them all. `Tx_RW_R1` starts a trans-

```
// Single read/write/CAS transactions:
Ptr Tx_Single_Read(TmPtr *addr);
void Tx_Single_Write(TmPtr *addr, Ptr newVal);
Ptr Tx_Single_CAS(TmPtr *addr, Ptr oldVal, Ptr newVal);

// Read-write short transactions:
Ptr Tx_RW_R1(TX_RECORD *t, TmPtr *addr_1);
Ptr Tx_RW_R2(TX_RECORD *t, TmPtr *addr_2);
...
bool Tx_RW_1_Is_Valid(TX_RECORD *t);
bool Tx_RW_2_Is_Valid(TX_RECORD *t);
...
void Tx_RW_1_Commit(TX_RECORD *t, Ptr val1);
void Tx_RW_2_Commit(TX_RECORD *t,
                    Ptr val_1, Ptr val_2);
...
void Tx_RW_1_Abort(TX_RECORD *t);
void Tx_RW_2_Abort(TX_RECORD *t);
...
// Read-only short transactions:
Ptr Tx_RO_R1(TX_RECORD *t, TmPtr *addr_1);
Ptr Tx_RO_R2(TX_RECORD *t, TmPtr *addr_2);
...
bool Tx_RO_1_Is_Valid(TX_RECORD *t);
bool Tx_RO_2_Is_Valid(TX_RECORD *t);
...
void Tx_RO_1_Abort(TX_RECORD *t);
void Tx_RO_2_Abort(TX_RECORD *t);
...
// Commit combined read-only & read-write transactions:
bool Tx_RO_1_RW_1_Commit(TX_RECORD *t, Ptr val1);
bool Tx_RO_1_RW_2_Commit(TX_RECORD *t,
                         Ptr val_1, Ptr val_2);
...
// Upgrade a location from RO to RW:
bool Tx_Upgrade_RO_1_To_RW_2(TX_RECORD *t);
...
```

Figure 3: SpecTM API for short transactions (pseudo-code).

action, and performs its first read. `Tx_RW_R2` performs its second read, and so on. Using explicit sequence numbers on the operations avoids the need for the SpecTM implementation to track the current size of the read-write set. The `Tx_RW_*_Is_Valid` functions validate a transaction that has performed the specified number of reads. The `Tx_RW_*_Commit` functions commit such a transaction, taking the new values to store at each of the locations accessed (e.g., taking 2 values in a 2-word transaction). This API forces all of a transaction's writes to be deferred until its commit point (allowing the implementation to be streamlined because a read does not need to consult a log of preceding writes).

The `Tx_RO_*` operations manage read-only transactions, in a similar manner to read-write transactions. A single transaction may mix the `Tx_RO_*` operations for the locations that it only reads, and the `Tx_RW_*` operations for the locations that it both reads and writes. The two sets of locations must be disjoint. A set of commit functions with names such as `Tx_RO_`*x*`_RW_`*y*`_Commit` is provided to commit these transactions: *x* refers to the number of locations read, and *y* to the number of locations written. As with the `Tx_RW_*_Commit` functions, the values

to write are supplied to the commit operation.

Finally, if a transaction wishes to "upgrade" a location from read-only access to read-write access, then the function `Tx_Upgrade_RO_`*x*`_To_RW_`*y* function indicates that index *x* amongst the transaction's existing reads has been upgraded to form index *y* in its writes—*x* may be any of the locations read previously, and *y* must be the next write index. Aside from locations that are upgraded, each `Tx_RW_R*` call and `Tx_RO_R*` call must access a distinct address.

To illustrate the use of these operations, a double-compare single-swap operation can be implemented as follows:

```
bool DCSS(TmPtr *a1, TmPtr *a2,
          Ptr o1, Ptr o2, Ptr n1) {
 TX_RECORD t;
restart:
  if (Tx_RO_R1(&t, a1) == o1 &&
      Tx_RO_R2(&t, a2) == o2 &&
      Tx_Upgrade_RO_1_To_RW_1(&t)) {
    if (Tx_RO_2_RW_1_Commit(&t, n1)) return true;
  } else if (Tx_RO_2_Is_Valid(&t)) return false;
  goto restart;
}
```

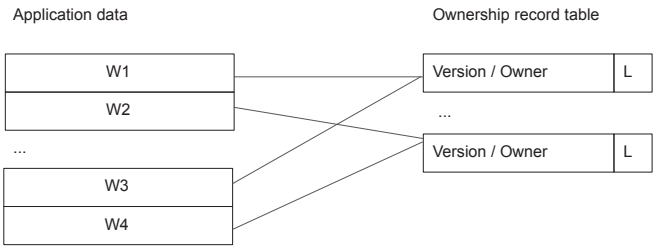
The `DCSS` function reads from the two locations supplied (`a1`, `a2`). If the values match those expected (`o1`, `o2`), then the first access is upgraded to a read-write access, and the new value (`n1`) written during commit. The transaction is repeated until either the commit succeeds, or a valid mismatch is seen.

2.2 SpecTM Implementations

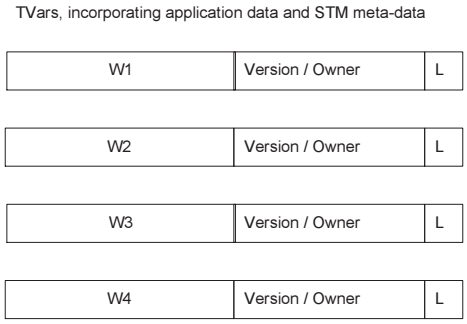
We have built three variants of SpecTM. They each implement the interface in Figure 3, but they differ in how a `TmPtr` is represented:

SpecTM-ORec. SpecTM-ORec follows the design of many general-purpose STMs in using a table of "ownership records" (ORecs) to hold the meta-data used by the STM system. A hash function maps heap addresses onto slots in a fixed-sized table of ORecs. This approach allows the STM's meta-data to be kept completely separate from the application's data. A `TmPtr` contains simply an ordinary pointer: the application's data structures do not need to be modified. A downside of this approach is that each transactional load will touch two cache lines: one to load the data, and a second to load the meta-data. Figure 4(a) illustrates this structure.

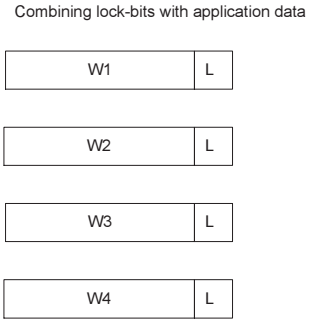
SpecTM-TVar. SpecTM-TVar follows the approach of STM-Haskell [14] by limiting the pointers passed to the STM functions to be references to specific "TVar" structures. Each `TmPtr` is a two-word TVar, holding a piece of STM meta-data alongside the piece of application data that it manages. With care, this allows



(a) SpecTM-ORec: Meta-data held in a table of ownership records (ORecs), indexed by a hash function.



(b) SpecTM-TVar: Meta-data co-located with application data in `TVar`s.



(c) SpecTM-LB: One lock bit of meta-data held in each data item.

Figure 4: Organization of STM meta-data in variants of SpecTM.

both words to be held on the same cache-line. However, it requires that an application's data structure be changed to accommodate the extra words. Figure 4(b) illustrates this scheme.

SpecTM-LB. SpecTM-LB reduces the meta-data used by the STM down to a single "lock bit" held in the *same* memory word as the application data that it

controls[1]. A `TmPtr` is a single word, within which this lock bit is held as the LSB. Figure 4(c) illustrates this structure. Using a single lock bit, in place of a TVar or ORec, reduces the memory consumption of the system.

However, in order for an algorithm using SpecTM-LB to be correct, the programmer must be careful about the structure of the transactions that are used: (*i*) at most one "normal" location can be accessed by `Tx_RO_*` operations, (*ii*) any number of locations can be accessed via `Tx_RW_*` operations, and (*iii*) any number of additional read-only locations can be accessed by `Tx_RO_*` operations provided that these locations satisfy a "non-repeated value" property (NRV).

In order to satisfy NRV, the program must ensure that any particular value `v` is stored into a each location `x` at most once between the start and end of any transaction. This might be satisfied if `v` contains a sequence number that is incremented on each store to `x` (and is large enough to prevent wrapping), or if `v` is a pointer to a dynamically allocated object that is placed in a data structure, and then de-allocated using a mechanism such as those of Michael [21] and Herlihy *et al.* [16]. (Forms of NRV property have been used to support multi-word atomic snapshot algorithms, and to avoid A-B-A problems in lock-free algorithms. We use the name NRV from Lev and Moir [19].)

Our earlier paper [10] examines the performance of these different implementations of SpecTM. For the data structures we have studied, SpecTM-LB performs best, then SpecTM-TVar, then SpecTM-ORec. We believe this primarily reflects the decreasing storage requirements, with SpecTM-LB requiring the least storage, and requiring the fewest cache lines to be accessed. Conversely, SpecTM-LB places the greatest burden on the programmer by requiring the programmer to ensure that the NRV property is satisfied.

In the next two sections we illustrate the use of SpecTM to implement a hashtable and a skip list (Sections 3–4). These two designs are both correct with all of the SpecTM implementations, including SpecTM-LB. Then, in Section 5 we discuss some of the limitations of programming using SpecTM. In Section 6 we discuss how some of these limitations can be overcome by integrating SpecTM with an general-purpose STM system.

3 Hashtable

In this section we illustrate the use of SpecTM to build a hash table. For brevity, we simplify the data structure to store only integer values and to provide `search`, `insert`, and `delete` operations. We also omit memory-management code from

[1]Note that the SpecTM-LB implementation is called "version-free" or "value-based" in our earlier paper [10].

the pseudo-code—many conventional techniques are available, as discussed in our earlier paper [10].

The overall approach is based on Fraser's design [11]. We assume a fixed size array of buckets, each of which is the head of a singly linked list of elements. One value is stored in each element. The lists can be of unbounded size. Consequently, depending on the loading of the hash table, we cannot rely on using a single SpecTM transaction to traverse an entire list.

Figure 5 provides pseudo-code. The hash table is implemented as an array of `TmPtr`s (line 3) that point to the sorted lists of nodes (line 5). Each node stores a single integer value and the `TmPtr` to the next node in the list. The hash table code internally uses an iterator that stores the address of the pointer to the current node and the address of the current node during the iteration (line 9).

We structure each list using the "mark bit" technique [13]. With this technique, we reserve a bit within the "next" pointer of each node in the list, and if this bit is set then the node itself is considered to have been deleted from the lists. Comparing the SpecTM implementation of the hash table with Fraser's original lock-free design, the main benefit from using SpecTM is in the handling of deletions. SpecTM lets us simplify deletion by using a 2-word transaction to atomically (*i*) mark a node as deleted, and (*ii*) unlink the node from the list. This atomicity avoids the need for concurrent traversals of the list to consider nodes that have been marked as deleted, but not yet excised from the list. In more detail:

The main internal function of the hashtable is the function for searching for a node with a specified value (line 14). The search function is invoked by all three public hashtable functions. The arguments of the function are used to pass the identifier of the node that is being searched for and the reference of the iterator used to return the position of the node. The search first locates the bucket the element belongs to and stores the address of the bucket list head into the iterator (line 15). Next, it traverses the bucket list by following the forward pointers of the nodes in the list (line 17). While traversing the list, the search does not need to consider marked nodes (line 18). The search can only access a marked node if it already held the reference to the node before the node was marked because the node gets marked and removed in the same transaction. This means that it is safe to just read through the marked pointers as they can only be marked by the concurrent remove operation. The traversal of the list stops when either the end of the list is reached or the element with the higher or equal value is found (line 19).

To check whether the hash table contains a particular value (line 25) it is enough to search for the value (line 27). If the search stops before reaching the end of the list and the element has the value that is being looked for then the function returns `true`. Otherwise, the element is not in the hash table and it returns `false`. There is no need to check whether the node returned by the search is marked or not. If the node is marked and the lookup returns `true` it simply means that it is

```
1  const int BUCKETS = 1024;
2  struct Hashtable {
3      TmPtr buckets[BUCKETS];
4  };
5  struct Node {
6      int id;
7      TmPtr next;
8  };
9  struct Iterator {
10     TmPtr *prev;
11     Ptr curr;
12 };
13
14 void Search(Hashtable *htable, int id, Iterator *it) {
15     it->prev = GetBucket(htable, id);
16     while(true) {
17         it->curr = Tx_Single_Read(it->prev);
18         it->curr = Unmark(it->curr);
19         if(it->curr == NULL || it->curr->id >= id) {
20             break;
21         }
22         it->prev = &(it->curr->next);
23     }
24 }
25 bool Contains(Hashtable *htable, int id) {
26     Iterator it;
27     Search(htable, id, &it);
28     return (it.curr != NULL && it.curr->id == id);
29 }
30 bool Add(Hashtable *htable, Node *node) {
31     Iterator it;
32 retry:
33     Search(htable, data->id, &it);
34     if(it.curr != NULL && it.curr->id == node->id) {
35         return false;
36     }
37     TmPtrWrite(&(node->next), it.curr);
38     if(Tx_Single_CAS(it.prev, it.curr, node) != it.curr) {
39         goto retry;
40     }
41     return true;
42 }
43 bool Remove(Hashtable *htable, int id) {
44     Iterator it;
45     TX_RECORD t;
46 retry:
47     Search(htable, id, &it);
48     if(it.curr == NULL || it.curr->id != id) {
49         return false;
50     }
51 retry_tx:
52     Ptr prevNext = Tx_RW_R1(&t, it.prev);
53     if(prevNext != it.curr) {
54         Tx_RW_1_Abort(&t);
55         goto retry;
56     }
57     TmPtr *nextPtr = &(it.curr->next);
58     Ptr nextVal = Tx_RW_R2(&t, nextPtr);
59     if(!Tx_RW_2_Is_Valid(&t)) {
60         goto retry_tx;{
61     }
62     Tx_RW_2_Commit(&t, nextVal, Mark(nextVal));
63     return true;
64 }
```

Figure 5: Hashtable algorithm using SpecTM.

linearized before the concurrent remove operation.

To add a new element (line 30), a search is first performed for the node with the same value as the element being added (line 33). If the node with the same value is found (line 34), the new element is not added to the hash table. If the element is not found, the iterator points at the successor of the new element. The next pointer of the new node is updated (line 37) and the SpecTM compare-and-swap transaction is executed to try to link the new element into the list (line 38). If the compare-and-swap does not succeed the whole operation is retried. There is no need to explicitly check whether the state of the nodes where the search stopped has changed. If it has, the compare-and-swap will fail and the operation will be restarted.

To remove a node with a particular value (line 43), a search is first performed to find the node to remove (line 47). If the node is not in the hash table, the remove returns `false` immediately, indicating that the operation could not be performed. If the element is found, a SpecTM transaction is executed to unlink the node that is being removed and to mark it atomically (lines 51–62). The transaction first re-reads the pointer to the node to remove (line 52) and checks whether it has changed. If it has (line 53), that means that the state of the nodes has changed since the search and the whole operation is restarted (line 55). Otherwise, the transaction reads the next pointer of the node that is being removed (line 58). If the transaction aborts at this point (line 59) the SpecTM transaction is restarted (line 60). If the read is successful, then the transaction can commit the new values of the previous and removed nodes' next pointers (line 62).

4 Skip List

The pseudo-code of the skip list algorithm is shown in Figure 6. Each skip list node stores an integer value, and an array of forward pointers. The array holds one pointer for each level of the skip list the node belongs to (line 2). The skip list is represented by a head node that points to the first node in each level of the list (line 7). To iterate along the list, a window of pointers for all skip list levels is used (line 10).

Similarly to the hashtable, we structure skip list using the "mark bit" technique, as in Fraser's lock-free skip list [11]. When a node is removed from the list, "mark bits" at all levels of the node are set, indicating that the node is deleted. Comparing the SpecTM implementation of the skip list with Fraser's original lock-free design, the main benefit from using SpecTM is in the handling of deletions. SpecTM lets us simplify deletion by using a transaction to atomically (*i*) mark a node as deleted, and (*ii*) unlink it from the list. Similarly to the hashtable, this atomicity avoids the need for concurrent traversals of the list to

```
1  const int MAX_LEVEL = 32;
2  struct Tower {
3    int id;
4    TmPtr next[MAX_LEVEL]
5    int lvl;
6  };
7  struct Skiplist {
8    Tower head;
9  };
10 struct Iterator {
11   Tower *prev[MAX_LEVEL];
12   Tower *next[MAX_LEVEL];
13 };
14
15 Tower *Skiplist::Search(int id, Iterator *it, int lvl) {
16   Tower *curr, *prev = &head;
17   while(--lvl >= 0) {
18     while(true) {
19       curr = Tx_Single_Read(&(prev->next[lvl]));
20       curr = Unmark(curr);
21       if(curr == NULL || curr->id >= id)
22         break;
23       prev = curr;
24     }
25     it->prev[lvl] = prev;
26     it->next[lvl] = curr;
27   }
28   return curr;
29 }
30 bool Skiplist::Add(Tower *data) {
31   Iterator it;
32   bool restartFlag;
33 restart:
34   int headLvl = PtrToInt(Tx_Single_Read(&head.lvl));
35   Tower *curr = Search(data->id, &it, headLvl);
36   if(curr != NULL && curr->id == id)
37     return false;
38   data->lvl = GetRandomLevel();
39   if(data->lvl == 1)
40     restartFlag = !AddLevelOne(data, &it))
41   else
42     restartFlag = !AddLevelN(data, &it);
43   if(restartFlag)
44     goto restart;
45   return true;
46 }
47 bool Skiplist::AddLevelOne(Tower *data, Iterator *it) {
48   TmPtrWrite(&(data->next[0]), it->next[0]);
49   return Tx_Single_CAS(&iter->prev[0]->next[0],
50     it->next[0], data) == it->next[0];
51 }
52 bool Skiplist::AddLevelN(Tower *data, Iterator *it) {
53   bool ret;
54   STM_START_TX(); // Using general-purpose STM
55   int headLvl = STM_READ_INT(&(head.lvl));
56   if(data->level > headLvl) {
57     STM_WRITE_INT(&(head.lvl), data->level);
58     for(int lvl = headLvl;lvl < data->level;lvl++) {
59       it->prev[lvl] = head;
60       it->next[lvl] = NULL;
61     }
62   }
63   for(int lvl = 0;lvl < data->level;lvl++) {
64     Ptr nxt = STM_READ_PTR(&win->prev[lvl]->next[lvl]);
65     if(nxt != it->next[lvl]) {
66       ret = false;
67       STM_ABORT_TX();
68     }
69     STM_WRITE_PTR(&(it->prev[lvl]->next[lvl]), data);
70     TmPtrWrite(&(data->next[lvl]), win->next[lvl]);
71   }
72   ret = true;
73   STM_END_TX();
74   return ret;
75 }
```


Figure 6: Skiplist implementation using SpecTM.

consider nodes that have been marked as deleted, but not yet excised from the list. The skip list algorithm is further simplified as the atomicity of insert and remove operations eliminates races between a concurrent insertion and removal of the same node: atomicity allows a node to be inserted/deleted at *all* levels as a single atomic action. In contrast, Fraser's lock-free skip list is further complicated by handling partially-inserted/partially-deleted nodes, and ensuring correctness when multiple operations are in progress on the same node at the same time. In more detail:

The function for searching the skip list (line 15) traverses the nodes by reading their forward pointers (line 19). It starts at the highest level in the skip list, moving successively lower whenever the level would skip over the integer being sought. The search function ignores deleted notes (line 20). The search terminates once it reaches the bottom level.

Adding a new node (line 30) starts with a search for the value being inserted (line 35). The skip list does not permit duplicate elements, so `false` is returned if the value is found (line 36) Otherwise, the search returns an iterator that can be used for the insertion. The level of the new node is generated randomly, with the probability of node being assigned a level l equal to $\frac{1}{2^l}$. The node is then inserted atomically into all of the lists up to this level. The nodes with level one are inserted using a short specialized transaction (lines 40) and the nodes with higher levels are inserted using an ordinary transaction (line 42). If the insertion does not succeed due to the concurrent changes to the skip list, the whole operation is restarted (line 44). Otherwise, the insert succeeds and `true` is returned to indicate its success.

Removals proceed in a similar manner to insertions. The node is first located using the search function. A single transaction is used to atomically mark the node at all levels, and to remove it from all of the lists it belongs too. Removal of nodes at level one is performed using a short specialized transaction, and the removal of nodes with higher levels is performed using ordinary transactions.

These insertion and removal operations typify the way we use SpecTM. The common cases are expressed using SpecTM transactions, and less frequent cases are expressed with more general, but slower, ordinary transactions. If developers see the need to further improve performance, they can further specialize the implementations.

5 Limitations of SpecTM

Broadly speaking, there are two difficulties when using SpecTM. First, there is the difficulty of writing an operation using short transactions, as opposed to using transactions of arbitrary length. This is an algorithmic problem, and we do

not yet understand the trade-offs very well. Aspects of this problem might be interesting to consider from a theoretical viewpoint, as well as in terms of ease-of-programming.

Second, there is the difficulty of expressing short transactions correctly using the SpecTM API: even if an algorithm has been decomposed correctly into a series of short transactions, there is a risk that the more complex SpecTM API will admit new kinds of error when writing mini transactions. The main difficulties we have encountered are:

Sequencing. SpecTM requires operations to be invoked in the correct sequence—e.g., `Tx_RW_R1` should be called before `Tx_RW_R2`, and the `Tx_RW_*_Commit` function that is called should match the number of data accesses that have been made. To detect sequencing bugs we need only track the size of the transaction's read-set and write-set, to ensure that the addresses in different elements in the sets are disjoint, and to check that an "upgrade" operation is performed at most once on any location.

Note that the `Tx_RO_*_Abort` functions exist solely to enable this form of sequencing check. These functions are empty in non-debug builds. However, when debugging, they delimit the boundaries between SpecTM transactions and their implementation resets the statistics maintained for sequence checks.

We have not yet built a tool for checking the use of the SpecTM API statically. However, a number of aspects of the design of SpecTM should help here. First, the correctness of a series of calls to the SpecTM API depends primarily on the names of the function being called, and on the set of functions that have previously been called. This means that a simple intra-procedural forwards data-flow analysis should be sufficient for tracking most usage. Note that this tool would not check the disjointness between the items in the read-set and write-set.

Validation. A more difficult aspect of the SpecTM API is the requirement to include calls to the `Tx_*_Is_Valid` functions whenever it is necessary to ensure that a transaction has seen a consistent view of memory. This dangers of working with inconsistent data have been reported in many earlier STM systems [15]. In part, these dangers led to the proposal for opacity as a correctness criteria for transactional memory. Approaches taken in earlier systems have included implicit validation as part of every transactional read [15], timestamp-based mechanisms to eliminate some of these validation steps [23, 27], along with static analyses to identify "safe" regions during which validation can be deferred [26]. For instance, if a thread performs a series of independent reads then validation may be deferred to the end of the sequence.

When programming using SpecTM, we rely on the programmer placing vali-

dation calls where necessary. Unfortunately, characterizing precisely what "where necessary" means is not straightforward. There are two broad options:

- First, we could place a very strong requirement on the part of the programmer, and permit *all* `Tx_RW_*` and `Tx_RO_*` operations to return *any* value. Validation must return `false` if the values returned by these operations do not represent a consistent view of memory.

  This definition allows a debug build to return "spurious" values which, in the absence of validation, are likely to lead to crashes—for instance, pointer-typed operations could probabilistically return addresses that are not mapped to valid memory. This definition is reminiscent of "catch fire" semantics for programs with data races (as in, for instance, the C/C++ memory model [4]). An advantage of this approach is that it provides a clear definition to the programmer of the behavior of invalid programs, and it makes it likely that crashes in debugging implementations of SpecTM will identify missing validation operations.

  Even with this definition, a programmer using SpecTM can still optimize the placement of validation calls. For instance, a single validation call may be used after a series of unrelated memory reads. Validation must be performed before de-referencing a pointer read from within a transaction, or before access an address computed from a value read.

- Second, we could define the semantics of `Tx_RW_*` and `Tx_RO_*` more strongly, and constrain exactly what they should do in the presence of invalidity—for instance, we might require that a value that was present in the location at some time in the past is returned, or we might require that a value present during the current transaction is returned.

  This second style of definition may allow the programmer to use slightly fewer validation operations, and hence obtain some performance improvement.

  A disadvantage of this approach is that the requirements on programmers are less clear, and it seems more difficult to build checking tools. The crux of the problem is distinguishing between cases in which validation has been missed accidentally, from cases where validation has been omitted deliberately to exploit a particularly subtle optimization. It is unclear how to distinguish these cases without a specification of the intended higher-level behavior of the program.

  There are many plausible definitions for the behavior of reads in invalid transactions—in contrast, the extreme approaches of "catch fire" and "opacity" are both relatively straightforward.

We currently take the "catch fire" approach, and have transactions in debug builds probabilistically appear invalid spuriously. We are not currently aware of any optimization opportunities that we are missing through this approach.

Non-repeating values (NRV). The final difficulty we highlight with SpecTM is the NRV property required of some locations in transactions in SpecTM-LB. In effect, NRV is shifting the responsibility of managing version numbers from the STM system to the programmer using the STM. A direct way to support NRV would be for each value to include a version number field that is incremented upon every write. This is typical of most STM algorithms, and SpecTM-LB's support for general NRV locations can be seen as a mechanism to exploit other forms of non-re-use, rather than just using a version number.

We do not currently have a good way to test that a program's use of memory satisfies NRV. The approaches that we have considered seem prohibitively costly, even for use in debugging builds. For instance, one could adapt the transactional write operations to maintain a history log for each location, and arrange that each SpecTM-LB transaction checks for re-use of values in these logs for the locations that it has read from. The cost of logging and checking is likely to be very high.

It might be possible to adapt this approach to perform checking probabilistically—either in terms of whether or not to log an update, or in terms of whether or not to perform checks at commit-time. It is not yet clear if these reduced checks would be sufficient to catch re-use. Equally, it is not clear if even the full checking regime would catch re-use bugs—it relies on spotting an occurrence of re-use, and so will not be useful for detecting problems that occur rarely. A further alternative would be to log additional information about the synchronization between threads, and to use this to spot "near miss" occasions of re-use, where a program contains a re-use bug, but where this is not witnessed by a SpecTM-LB transaction in a given run.

6 Integration with General-Purpose TM

From a pragmatic viewpoint, the main way in which we address the limitations in Section 5 is to enable inter-operation between transactions written with SpecTM and transactions written through a general-purpose TM interface. This reduces the amount of code that must be written using SpecTM: the programmer can use SpecTM to optimize performance-critical transactions, and use the general-purpose interface for code that is more complex.

Integration between SpecTM and a general-purpose TM can be implemented either by having both TM systems manage disjoint sets of memory locations, or by designing mechanisms to allow the two types of transaction to access the same

data correctly. In the former case, transactions written with SpecTM and with the general-purpose STM can be composed in a manner similar to *two-phase commit*: both transactions are first prepared for commit and, only after the prepare is successful, they are committed together. If one of the transactions aborts, then the other transaction gets aborted as well. In this manner the composite transaction is also atomic. With this approach, the general-purpose transaction can benefit from optimized implementations of data structures with SpecTM, but the data structures cannot be implemented with a mix of specialized and general-purpose transactions. The latter approach enables the programmer to have the specialized and general-purpose transaction access the same data concurrently. For instance, one could use a general-purpose transaction to re-size a hash table, while using SpecTM transactions for the common cases of data accesses. We focus on the latter form of inter-operability as it is more general and more interesting when implementing concurrent data structures.

In SpecTM we use two kinds of specialization to attempt to streamline the implementation: (*i*) the SpecTM API requires more work on the part of the software using STM, in an attempt to reduce the work needed within the SpecTM implementation, (*ii*) the representation of `TmPtr` structures in SpecTM-TVar and SpecTM-LB attempts to reduce the space occupied by the TM meta-data.

The different `TmPtr` representations introduce different constraints on integration between SpecTM and a general-purpose STM. Concretely, we use a general-purpose STM we refer to as "BaseTM". This uses similar techniques to SwissTM [9] and the STM described by Spear *et al.* [25].

SpecTM-ORec. Our SpecTM-ORec implementation uses the same protocol for managing the ORecs as the BaseTM system. No additional work is required, either on the SpecTM-ORec transactions, or on the BaseTM transactions.

SpecTM-TVar. SpecTM-TVar changes the way in which transactional data is represented. This prevents BaseTM from being used directly on the same data: the SpecTM transactions would be using ORecs held alongside the data in transactional variables in `TmPtr` structures (Figure 4(b)), whereas the BaseTM transactions would be using ORecs held in the usual ORec table (Figure 4(a)).

There are two main approaches for integrating SpecTM-TVar with general-purpose transactions.

Hybrid-TM-style. The first approach is to build on earlier techniques for hybrid HTM/STM systems [18, 6]. In Hybrid-TM models, the HTM is used to provide good performance, while an STM serves as a backup to handle situations where the HTM could not execute the transaction successfully. This approach may reduce the pressure on HTM implementations to provide features such as

long-running transactions, or conditional blocking.

Unfortunately, many Hybrid-TM techniques are not good fits for SpecTM. The problem is that the SpecTM-TVar transactions would be required to monitor the BaseTM transactions for conflicts (because the BaseTM transactions are unaware of the TVars being used by SpecTM-TVar). This additional monitoring would harm the performance of SpecTM transactions.

One hybrid technique that *is* practical to use is Lev *et al.*'s Phased TM system (PhTM, [20]). Instead of requiring hardware transactions to check for conflicts with concurrent software transactions, PhTM prevents HW and SW transactions from executing concurrently. The PhTM system maintains a counter of currently-executing SW transactions. Every HW transaction checks this counter, and if non-zero, the HW transaction aborts itself. Since the counter is also read inside the HW transaction, any subsequent modifications to this counter also trigger an abort. This approach reduces overheads for HW transactions, but it results in increased aborts (as discussed by Baugh *et al.* [2]): an overflow of even a single HW transaction aborts *all* other concurrently executing HW transactions.

Haskell-STM-style. An alternative to a phased-TM system is to adapt the BaseTM interface to use TVars. Unlike SpecTM, these general-purpose transactions can access an unbounded number of locations, and they do not need to provide sequence numbers on their accesses, or to distinguish read-only locations from read-write locations. However, as with SpecTM-TVar, all of a transaction's data accesses must be to locations with associated TVars. With this interface, the meta-data used by the STM system is the same as the meta-data used by SpecTM-TVar.

Whether or not this approach is practical will depend on the setting. It seems most palatable when writing new data structures using transactions: it requires the representation of the data to be able to be adapted to include TVars, and so it would be inappropriate for existing data structures, or those which are sometimes used inside transactions and sometimes used outside.

SpecTM-LB. SpecTM-LB uses only a single lock bit within each of the locations managed by the STM. The problem now is not that SpecTM's meta-data is in a different place to BaseTM's, but that the actual format of the meta-data is different. Again, two approaches are possible:

Phased-TM. As with SpecTM-TVar, we can use the techniques of Lev *et al.* to ensure that SpecTM-LB and BaseTM transactions do not run concurrently [20]. An advantage of this approach is that the only overhead on SpecTM-LB transactions is to ensure that execution is in a "SpecTM phase". A disadvantage is that, if even a single thread wishes to use general-purpose transactions, then *all* SpecTM-LB transactions must be prevented from running, irrespective of the data that they

are accessing.

Locked-by-Software. The intuition behind Phased-TM is that two different TM systems can co-exist without shared conflict detection mechanisms so long as they are separate in time. An alternative form of STM integration for SpecTM-LB is to keep the locations managed by SpecTM separate in space from those managed by BaseTM. That is, both kinds of STM can co-exist, so long as they are accessing disjoint sets of locations.

If lock bits can be reserved in all transactional data, then these can be used to control the separation between SpecTM-LB and BaseTM. If the bit is set then either (*i*) the location is currently part of a current SpecTM transaction's write set, or (*ii*) the location is currently owned for writing by BaseTM. If the bit is clear, then the location is available for reading by either kind of transaction.

This approach avoids intruding on the fast path of SpecTM-LB transactions that run and commit without conflict—all of the locations that they encounter will have the lock bit clear, and all of the additional work to integrate with BaseTM will be on the existing slow-path for when the lock bit is set. Conversely, BaseTM must ensure that it has ownership of all of the locations it is accessing by setting the lock bit before accessing them.

The main complexity with this use of the lock bit is how to arrange for BaseTM to release the lock bit in order to allow SpecTM-LB transactions to acquire it. Our current design is:

- BaseTM eagerly acquires the lock bit when executing a transaction.
- BaseTM releases the lock bit only when requested by a SpecTM-LB transaction that wishes to access the same location.
- If a SpecTM-LB transaction finds that the lock bit is held by a BaseTM transaction, then the thread running the SpecTM-LB transaction executes a "dummy" BaseTM transaction on the location. The dummy transaction synchronizes with other BaseTM transactions (ensuring no other writers are present) before releasing the lock bit back to SpecTM-LB's use.

This approach avoids repeated updates to the lock bit when a location is accessed by a series of BaseTM transactions.

7 Discussion

In this paper we have discussed the design of the SpecTM interface, illustrated its use in constructing shared memory data structures, and discussed some of

the shortcomings of SpecTM, along with techniques for integrating the different forms of SpecTM with general-purpose STM systems.

The limitations in Section 5 all reflect additional requirements that are placed on the programmer when using the SpecTM API rather than when using a traditional TM API. We believe that two of these limitations are relatively straightforward for the programmer to handle: The problem of sequencing the invocation of SpecTM operations correctly is straightforward to check dynamically, and appears relatively amenable to static analysis. The problem of calling validation functions correctly follows the existing problem of correctly sandboxing programs using TM systems without opacity in languages such as C/C++. In addition, in each of these two cases, it seems likely that straightforward testing tools could check that a program is constructed correctly, or that a compiler could target the SpecTM API correctly for programs whose workloads are suitable.

However, it is unclear if the additional performance benefits of exploiting the non-repeating value property (NRV) are sufficient for the additional complexity in using SpecTM-LB. This is the one setting in which we do not have a good checking tool (static or dynamic), and in which there seems to be a great risk of programmers making accidental errors in their use of SpecTM. In future work we would like to study this problem more closely—e.g., is it possible to provide a sufficiently expressive set of "NRV-safe" abstractions that guarantee that the NRV property will be satisfied, and is it possible to develop checking techniques that are sufficiently lightweight to be used in practice?

References

[1] Hagit Attiya. Invited paper: The inherent complexity of transactional memory and what to do about it. In *Distributed Computing and Networking*, volume 6522 of *Lecture Notes in Computer Science*, pages 1–11. 2011.

[2] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *ISCA '08: Proc. 35th Annual International Symposium on Computer Architecture*, pages 115–126, June 2008.

[3] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), November 2006.

[4] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08: Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, June 2008.

[5] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick

Marlier, and Etienne Riviere. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys '10: Proc. 5th ACM European Conference on Computer Systems*, April 2010.

[6] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *ASPLOS '06: Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, October 2006.

[7] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proc. 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 157–168, March 2009.

[8] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, pages 1–31, 2012.

[9] Aleksandar Dragojević, Rachid Guerraoui, and Michał Kapałka. Stretching transactional memory. In *PLDI '09: Proc. 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 155–165, June 2009.

[10] Aleksandar Dragojević and Tim Harris. STM in the small: trading generality for performance in software transactional memory. In *EuroSys '12: Proc. 7th European conference on Computer systems*, April 2012.

[11] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[12] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–184, February 2008.

[13] Tim Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC '01: Proc. 15th International Conference on Distributed Computing*, 2001.

[14] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton Jones. Composable memory transactions. In *PPoPP '05: Proc. ACM Symposium on Principles and Practice of Parallel Programming*, June 2005. A shorter version appeared in *CACM* 51(8):91–100, August 2008.

[15] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Morgan & Claypool, 2010.

[16] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *TOCS: ACM Transactions on Computer Systems*, 23(2):146–196, May 2005.

[17] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proc. 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[18] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, March 2006.

[19] Yossi Lev and Mark Moir. Lightweight parallel accumulators using C++ templates. In *IWMSE '11: Proc. 4th International Workshop on Multicore Software Engineering*, 2011.

[20] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd Workshop on Transactional Computing*, August 2007.

[21] Maged M. Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[22] James Reinders. Transactional synchronization in Haswell, February 2012. `http://software.intel.com/en-us/blogs`.

[23] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 284–298, September 2006.

[24] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proc. 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, August 1995.

[25] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 141–150, February 2009.

[26] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC '06: Proc. 20th International Symposium on Distributed Computing*, September 2006.

[27] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proc. International Symposium on Code Generation and Optimization*, pages 34–48, March 2007.

The Logic in Computer Science Column

by

Yuri Gurevich

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
gurevich@microsoft.com

Classes of Algorithms: Formalization and Comparison

Serge Grigorieff
LIAFA, CNRS & Université Paris 7
seg@liafa.jussieu.fr

Pierre Valarcher
LACL, Université Paris-Est
pierre.valarcher@u-pec.fr

Abstract

We discuss two questions about algorithmic completeness (in the operational sense). First, how to get a mathematical characterization of the classes of algorithms associated to the diverse computation models? Second, how to define a robust and satisfactory notion of primitive recursive algorithm? We propose solutions based on Gurevich's Abstract State Machines.

Bulletin of the EATCS no 107, pp. 95–127, June 2012

Contents

1 Introduction

• From Church Thesis to Gurevich' Sequential Thesis

Since Gurevich introduced Abstract State Machines (aka Evolving Algebras) in the 80's [24, 25], there is now a formal mathematical notion of small-step (i.e. non parallel) algorithm working in discrete time.

This answers a question raised by Turing in his celebrated 1936 paper:

> *The real question at issue is "What are the possible processes which can be carried out in computing a [real] number?" [38], page 20*

and Donald Knuth's concern for the question in his famous books [28]:

> *The notion of an algorithm is basic to all of computer programming, so we should begin with a careful analysis of this concept.*
>
> (The Art of Computer Programming, vol. 1, p. 1, §1.1 "Algorithms")

As a formalization of an intuitive notion, there can be no formal proof that ASMs (Abstract State Machines) truly formalize what they are intended to. Exactly as there can be no formal proof that the mathematical notion of recursive functions truly formalizes the intuitive notion of computable function.

Nevertheless, as is well-known, all the approaches to the notion of computability considered up to now have been proved to be equivalent to or included in that of recursive function. This carries water for the famous Church Thesis. The same holds with ASMs. The algorithms of all (small-step, discrete time) computation models considered up to now have been proved to be faithfully emulated by ASMs. This has lead to Gurevich' Sequential Thesis [25] which is to algorithms what is Church Thesis to computable functions.

• Classes of algorithms associated to computation models

In §2 we propose a formalization of classes of algorithms associated to computation models in terms of classes of ASMs defined by purely mathematical conditions on the ASM framework (namely, fixing the static background and the vocabulary of the dynamic foreground, cf. detailed explanations in §2) *with no condition on ASM programs except typing constraints.*

This last requirement may seem to be quite a challenge. Indeed, there is a small price to fulfill this requirement: computation models have to be closed under constant speedups. We consider that this last condition as a normalization which removes contingencies while preserving the core ideas of the computation model. Though the reader may not buy that such a closure property is a virtue in itself, we expect him to admit that it is a small price for a truly mathematical characterization.

• Algorithmic gaps in computation models

For a computation model, Turing completeness does not ensure algorithmic completeness: though all computable functions are obtained, maybe some algorithms are missing.

In §3.1 we recall how resource complexity may reveal algorithmic incompleteness. In §3.2, 3.3 we recall Colson's remarkable theorem about the limitations of primitive recursive definitions as algorithms for primitive recursive functions. This result is the motivation for Definition 5.2 in §5.

• Operational equivalence

In §4, we discuss operational equivalence of computation models, i.e. equality of the associated classes of algorithms.

First, in §4.2, we recall a recent result [20] about the operational completeness of lambda calculus. In §4.3, 4.4, we consider an imperative programming language for algorithms associated to (the inherently functional) Gödel system T.

• **What is a primitive recursive algorithm?**

The algorithmic gaps discovered by Colson (cf. §3.2) show that the obvious answer to that question is totally unsatisfactory. In §5, we recall the solution proposed in [39] and describes an imperative programming language $\text{Loop}_{\texttt{halt}}$ which is algorithmically complete with respect to the simplest natural class of primitive recursive algorithms APRA.

The rest of the paper is a dialog between the authors and Yuri Gurevich's imaginary student Quisani.

2 Computation models as simple classes of ASMs

2.1 The problem

Q: I heard that you showed that usual sequential-time non-parallel computation models correspond to very simple classes of Abstract State Machines.

A: Yes, this appears in [21, 23]. Recall that the Gurevich Sequential Thesis [25] ensures that any sequential time small step algorithm is modeled step by step by an ASM.

Q: "Small step algorithm": do you mean that a bounded amount of work is performed at each step?

A: Yes, this is what it means. The simplest way to associate a class of ASMs to a computation model is as follows. Consider all ASMs which model step by step the algorithms associated to the machines or programs of that computation model. Alas, this correspondence is not effective at all: in general, it is undecidable to test if an ASM models a particular algorithm.

Q: A usual phenomenon. Maybe you can take a subclass of that.

A: Yes. In fact, this is what is usually done. For each particular machine or program of the computation model, one devises a particular ASM which models step by step the associated algorithm. This gives a simple and effective class of ASMs. In particular, since it is effective, this class is necessarily a proper subclass of the previous largest one.

Q: Ok, but the obtained class has no intrinsic ASM characterization.

A: You are right. Indeed, given a computation model C, what we do is to define a class $\mathcal{M}$ of ASMs which has an intrinsic ASM definition and gives exactly the same algorithms as machines or programs in the computation model C.

2.2 Relaxing the time unit

A: But let us start with a word of caution. This approach does not work with the computation models as they are usually considered. We have to generalize them by letting the time unit be chosen arbitrarily.

Q: What does this mean?

A: Consider a Turing machine $\mathcal{M}$. It is usual to consider its instantaneous descriptions at times 0, 1, 2, ... up to time t if $\mathcal{M}$ halts at time t. Instead of that, fix some $k \geq 1$ and consider the instantaneous descriptions at times 0, k, $2k$, ... up to time $\lceil \frac{t}{k} \rceil k$ if $\mathcal{M}$ halts at time t. Looking at $\mathcal{M}$ this way, we get another machine $\mathcal{M}^{(k)}$ which "runs k times faster". The generalization of the model we consider is to add all $\mathcal{M}^{(k)}$'s, $k \geq 1$, for every $\mathcal{M}$.

Q: Relaxing the time unit... Is this generalization so benign? Don't you loose some important feature of a computation model?

A: Choosing a time or space unit carries some arbitrariness. An elementary action or piece of code can always be seen as a family of more atomic ones. On the opposite, some groups of several elementary actions or pieces of code can be considered as what is truly significant.

For instance, a transition of a Turing machine is considered as one computation step. But, you could view it as six computation steps corresponding to different actions of the machine: 1) read the scanned cell, 2) decide how to overwrite its contents, 3) overwrite the scanned cell, 4) decide how to move the head, 5) move the head, 6) change state.

Another example with space rather than time. It is common to say that computers work in base 2. But bits are usually grouped eight by eight into bytes. So computers can also be said to work in base 256.

We view the considered generalization of computation models as a normalization which removes contingencies and preserves the core features of the models.

Q: Hum... What is a contingency? What is a core feature?

A: Some examples will clarify our claim.

2.3 ASMs in a nutshell

A: But, first, let us briefly recall how Gurevich devised ASMs according to his analysis of the constituents of an algorithm. Let us consider only sequential-time small-step algorithms. Gurevich [25, 26] views an algorithm as a transition system and points four basic constituents.

(1) The data structures involved in the algorithm. They constitute the multisort domain of the ASM.

(2) The operations or relations over these data structures which are given for free: the so-called static framework. Observe that these operations and relations are typed. Together, the multisort domain and the associated static framework constitute the background of the computation.

(3) A dynamic vocabulary to name the elements and functions over the data structures which form the foreground of the computation. In the conventional programming, this role is played by the variables. The foreground is dynamic: it may vary at each step of the execution of the algorithm. This dynamic vocabulary is also typed.

A state of the algorithm is the static framework over the multisort domain, augmented with interpretations of the symbols of the dynamic vocabulary.

(4) The fourth ingredient of an algorithm is the program which determines the state change, i.e. the evolution of the dynamic foreground.

The run of an ASM halts if and when either some special value of a dynamic symbol is obtained or the ASM has reached a fixed point, so that two successive states are identical.

Q: So, the algorithm does not tell anything about how the values of static functions are obtained. This departs from the a priori intuition that algorithms tell everything about what they do.

A: Static functions have to be considered as oracles. The algorithm decomposes its global input/output action into a succession of atomic steps. These atomic steps are the limit of the algorithmic process: they use static functions totally ignoring their operationality.

Q: Thus, *algorithms are intrinsically oracular!*

A: Right. This is an essential feature.

For instance, with Turing machines, moving the head, reading the scanned symbol, changing state are static operations given for free. The machine knows how to do that though there is nothing about that in the transition table. However, as anyone can experience, if you want to write a program to emulate a Turing machine in any programming language, there will be lines of code devoted to these operations... As another example, with Blum-Shub-Smale machines [5], one can even check for free if a real number is zero or not. A test known to be non-computable since 1954 (Rice [34]) but given for free in the static framework!

Q: So, there is no absolute notion of algorithm.

A: Well, there is a smallest “effective" notion of algorithm: the one obtained by restricting to computable backgrounds. This means to restrict to computable data structures and to require all static functions to be computable.

Q: Going back to the brief overview of ASMs, what are ASM programs?

A: An ASM program is built with conditionals and updates. Updates are of the form $f(\tau_1, \ldots, \tau_k) := \tau$ where f is a dynamic symbol and $\tau_1, \ldots, \tau_k, \tau$ are any terms. These are in fact ground terms: there is no variable hence all terms are ground. The meaning of such an update is as follows: let $a_1, \ldots, a_k, a$ be the values of the τ_i and τ terms computed at step t; then at step $t + 1$, the new value of f at $a_1, \ldots, a_k$ is a.

Q: I remember that when I first learned about ASMs, I was really surprised that there is no explicit loop instruction in ASM programs.

A: It is the run of the ASM, obtained by applying again and again the ASM program, which constitutes kind of a meta loop.

2.4 Intrinsic ASM characterization of computation models

A: Let us go back to computation models and ASMs. Our idea is as follows:

> *Usual computation models are obtained by fixing the three first constituents of ASMs: namely, the data structures, the static framework and the dynamic vocabulary.* As for the fourth constituent, the ASM program, there is no constraint except for the type constraint in the syntactic construction of terms.

Q: Oh! You require the program to be subject to no constraint but typing restrictions. All emulating programs I ever saw code a lot of information about the emulated computation model: they are very particular programs! How is it possible not to discriminate among ASM programs? You are setting yourself such a high bar that it seems impossible to jump over it.

A: This is possible if you relax the time unit.

Q: How does this work for Turing machines?

A: Consider Turing machines with one tape infinite in both directions. There are three data types involved: an infinite set of states Q, an infinite set of letters A and the set $\mathbb{Z}$ of integers to denote the cells on the tape.

Q: Sorry, there are only finitely many states and letters!

A: Yes, but arbitrarily large. So let them be infinite sets. Finiteness will come as a side effect of the ASM program: being a finite object, it will mention only finitely many states and letters.

Q: Nice trick.

A: The static framework for one-tape Turing machines is as follows. Infinitely many nullary function symbols to denote the diverse states and letters and two unary function symbols to denote the successor and predecessor functions on $\mathbb{Z}$. That's all.

Q: Nullary function symbols? You mean constant symbols.

A: Yes. But we shall also deal with dynamic symbols and "dynamic constant" sounds quite awkward. This is why the term "nullary function" is preferred.

Q: So you consider an infinite static vocabulary.

A: Not a big deal. Any ASM program will use only a finite part of this infinite vocabulary.

The dynamic vocabulary consists of three symbols σ, π and γ. The two first ones are nullary function symbols of respective types Q and $\mathbb{Z}$, they are used to denote the current state and position of the head. The last one is a unary function symbol of type $\mathbb{Z} \to A$, it is used to denote the current contents of the tape.

Q: Ok, any Turing machine obviously corresponds to such an ASM. Now, given such an ASM, how do you get a Turing machine?

A: Look at what the program can express. You can update the current state σ to any new value. The position of the head π can also be updated to a new value. But this new value has to be defined by an ASM term of type $\mathbb{Z}$. Look at the static and dynamic vocabulary: there is only one way to get a term of type $\mathbb{Z}$, by applying the successor and predecessor functions to the dynamic constant π. Thus, you can move the head a bounded number of cells, say m, left or right. You can also update the dynamic function γ, henceforth modifying the contents of the current tape. But this can be done at a bounded number of places around the position of the head, say at distance $\leq n$. These bounds m, n depend on the length of terms in the ASM program. Thus, in one step of the ASM you get what is done in $m + 4n$ steps of an appropriate Turing machine.

Q: So this ASM simulates some Turing machine considered with a $m + 4n$ times larger time unit. This is why you extend the computation model by relaxing the time unit. By the way, why $4n$?

A: You may have to modify the contents of the n cells right of the head and the n cells left of the head. So, n steps to modify cells on one side of the head, n steps to move back the head and n steps to modify cells on the other side of the head. Finally, at most $n + m$ steps are necessary to move the head to the new scanned cell. With some no move transitions, this last phase requires exactly $n + m$ steps.

2.5 Slight background modification, big consequences

Q: A question. There is no static symbol for the integer 0.

A: No, and this is no accident. If you add a symbol 0 then the update $\pi := 0$ resets the head at cell 0 in one step. Thus, we would model "reset Turing machines". Which is another computation model.

Q: Right. Is it as easy for any computation model as it is for Turing machines?

2.6 ASM reserve and types with Cartesian product ranges

A: For multi-tape Turing machines, for RAMs, for grammars, for automata, etc. it does work with no real difficulty. But there is a problem with Kolmogorov-Uspensky machines and with Schönhage Storage Modification Machines. Their tapes are graphs which can grow.

Q: So you have to appeal to some "reserve set" to pick a new vertex.

A: Here comes the problem. When you pick a new element in the reserve, you have to do two actions: add the picked element to the graph and remove it from the reserve.

Q: Easy to tell in an ASM program. What is the problem?

A: The problem is that this is a constraint on programs: if the program adds a fresh node then it should also remove it from the reserve. But our approach puts no constraint on ASM programs except typing!

Q: How do you get around the problem?

A: We generalize typing. We want two simultaneous actions. In other words, we want to fire two functions at the same time. A solution is to introduce types with Cartesian product ranges. Instead of considering two functions $f : C \to D$ and $g : C \to E$ we consider a function $(f, g) : C \to D \times E$.

Q: Are you kidding? This is the same thing!

A: Not for the way we use it. First, given f and g separately, you can fire f and g on different arguments whereas if you fire (f, g) then you must fire f and g on the same argument. Second, there are far fewer possibilities of composition with a function of type $C \to D \times E$ than with two functions, one of type $C \to D$ and the other one of type $C \to E$.

Let us detail the case of Schönhage machines. The data structures are Q, A for states and letters, an abstract infinite set X for the nodes of the graph-tape (current nodes and "reserve" nodes) and the data structure $\mathcal{P}_{<\omega}(X)$ (the family of finite subsets of X) for the current set of nodes. There are static constants for states and

letters and a static function $new : \mathcal{P}_{<\omega}(X) \to X \times \mathcal{P}_{<\omega}(X)$ with Cartesian product range such that, for any $Z \in \mathcal{P}_{<\omega}(X)$, $new(Z)$ is of the form $(a, Z \cup \{a\})$ where $a \notin Z$. The dynamic symbols are σ of type Q for the current state, π of type X for the current position of the head, U of type $\mathcal{P}_{<\omega}(X)$ for the current set of nodes of the graph-tape, γ of type $X \to A$ for the labels of the nodes (with a default value for nodes not in U) and $f_1, \ldots, f_k$ of type $X \to X$ for the partial functions giving the nodes pointed by a given node.

Now, since U is the unique term of type $\mathcal{P}_{<\omega}(X)$, any use of *new* will involve U in an update $(\tau, U) := new(U)$ where τ is either π or a composition of the f_i's applied to π. Such an update simultaneously picks a fresh element from the reserve (which is the difference set $X \setminus U$) and removes it from the reserve, as wanted.

Q: In conventional programming languages like Java, one would not need such function pairs since a new object is created and then is worked upon during the same computation step.

A: Yes, this generalization of typing is done for ASM programming in order to fulfill our ASM characterization of computation models with no constraints on program except for the type constraints in the syntactic construction of terms.

Q: So the price to pay for this ASM characterization of computation models is
1) the extension of computation models by relaxing the time unit,
2) the generalization of typing.

A: Exactly. In our opinion, 1) is forgetting contingencies and 2) is reviving a forgotten (or underrated) feature of functions: coarity, i.e. the number of items constituting the output.

3 Turing completeness vs algorithmic completeness

Q: What are the tools to prove that the step-by-step behaviour of an algorithm is different from that of any algorithm in a given class of algorithms?

A: Tools to reveal operational gaps in computation models and programming languages, even in Turing complete ones?

3.1 Resource complexity theory reveals operational gaps

A: For sure, the simplest tool is computational complexity. For instance, some problems are much harder to solve with one-tape Turing machines than with two-tapes Turing machines. An example is palindrome recognition which requires

quadratic time with one tape Turing machine but is obviously solvable in linear time with two tapes.

Q: Yes, with one-tape there is no way but sweep back and forth through the tape to check that a first group of letter coincides with the last one, then a second group of letters next to the first one coincides with the penultimate one and so on. One can do this for groups of one, two or more letters but necessarily uniformly bounded groups. So we get time about $2n + 2(n-k) + 2(n-2k) + \ldots$, which is $\Omega(n^2)$ indeed.

A: Yet the formal proof that the idea works involves some technique. Crossing sequences in Hennie's original proof [27], or Kolmogorov complexity in Wolfgang J. Paul's very elegant proof [32].

By the way, this result has been extended to multidimensional one-tape Turing machines with a time lower bound $\Omega(n^2/\log(n))$, cf. [4].

Q: Such resource complexity lower bounds appear with a lot of different models. The more powerful the computation model, the more high-performing are the algorithms it implements to solve a fixed problem.

A: Sure. In ASM terms, your observation can be rephrased: with more powerful (static) background and richer dynamic vocabulary, ASM programs simulate step-by-step more sophisticated and high-performing algorithms.

Q: Is there such a substantial difference between the static backgrounds and dynamic vocabularies of one-tape and two-tape Turing machines?

A: Let us see. With two-tape Turing machines, there are two copies of $\mathbb{Z}$, say $\mathbb{Z}^{(\varepsilon)}$, $\varepsilon = 0, 1$, one for each tape, and on each copy, the successor and predecessor operations. This is where lies the difference: one or two copies of the structure $(\mathbb{Z}, x \mapsto x+1, x \mapsto x-1)$. And for each such copy $\mathbb{Z}^{(\varepsilon)}$ there is a symbol π_ε of type $\mathbb{Z}^{(\varepsilon)}$ for the position of the head, and a symbol $\gamma_\varepsilon : \mathbb{Z}^{(\varepsilon)} \to A$ for the current contents of the tape.

Q: Is this redundancy really necessary?

A: Yes! An example, different but similar to the above, illustrates the strength of duplication of structures in the ASM framework. Consider one counter machines. Modeled as ASMs, we have the sort Q for states, the sort $\mathbb{N}$ for contents of the counter, the static structure $(\mathbb{N}, x \mapsto x+1, x \mapsto x-1)$ and two dynamic constants: σ for the current state and γ for the current contents of the counter. Observe that γ has type $\mathbb{N}$. Consider now two-counter machines. The ASM model is analog with two copies $(\mathbb{N}^{(\varepsilon)}, x \mapsto x+1, x \mapsto x-1)$ and $\gamma^{(\varepsilon)}$.

Now, the duplication of $(\mathbb{N}, x \mapsto x+1, x \mapsto x-1)$ and γ witnesses a spectacular classical result: the halting problem is decidable for one-counter machines, but it is undecidable for two-counter machines!

3.2 Operational gaps and primitive recursive functions

Q: By the way, what about intentional behavior of programming languages? A subject initiated by Loïc Colson in the 1990's.

A: The primary goal of his research was to look at the algorithms associated to programs in some programming languages. The question was to find operational properties of such algorithms computing very simple and usual functions.

In particular, in [9], L. Colson considers the particular primitive recursive function *min* which computes the minimum of two non negative integers. Very elementary function, isn't it?

Q: Sure. A simple way to compute this function is to decrement x, y in parallel as long as none is 0. This runs in time $O(min(x, y))$.

A: The natural algorithm that you describe is based on a definition of the function *min* by the following equations:

$$min(0, y) = 0 \quad , \quad min(x, 0) = 0 \quad , \quad min(x + 1, y + 1) = min(x, y) + 1.$$

Though these equations constitute an inductive definition of *min*, this is not a primitive recursive definition since the induction proceeds with two variables simultaneously.

Colson looks at computations which use exclusively the equations involved in a primitive recursive definition of min. He proves that each such computation takes time $O(x)$ or $O(y)$ to compute $\min(x, y)$. When x and y are not of the same magnitude, this does not match the previous $O(min(x, y))$ time algorithm.

Q: What does it mean that a computation uses exclusively the equations involved in a primitive recursive definition?

A: The equations are considered as reduction rules and the computation is a succession of applications of these rewriting rules. Formally, Colson considers PR combinators which formalize primitive recursive definitions of functions.

(i) The basic PR combinators are $\mathtt{0}$, $\mathtt{S}$, $\pi_{k,i}$ (where $k, i \in \mathbb{N}$, $1 \leq i \leq k$). They have respective types $\mathbb{N}$, $\mathbb{N} \to \mathbb{N}$ and $\mathbb{N}^k \to \mathbb{N}$ and represent the integer 0, the successor function over $\mathbb{N}$ and the projections $\mathbb{N}^k \to \mathbb{N}$.

(ii) If c is a PR combinator of type $\mathbb{N}^k \to \mathbb{N}$ and $c_1, \ldots, c_k$ are PR combinators of type $\mathbb{N}^\ell \to \mathbb{N}$ then $\langle c; c_1, \ldots, c_k \rangle$ is a PR combinator of type $\mathbb{N}^\ell \to \mathbb{N}$. If $c, c_1, \ldots, c_k$ represent functions $g : \mathbb{N}^k \to \mathbb{N}$, $h_1, \ldots, h_k : \mathbb{N}^\ell \to \mathbb{N}$ then $\langle c; c_1, \ldots, c_k \rangle$ represents the function $\vec{x} = (x_1, \ldots, x_\ell) \mapsto g(h_1(\vec{x}), \ldots, h_k(\vec{x}))$.

(iii) Given two PR combinators c and d of respective types $\mathbb{N}^k \to \mathbb{N}$ and $\mathbb{N}^{k+2} \to \mathbb{N}$, we get another PR combinator $\mathtt{rec}_{c,d}$ of type $\mathbb{N}^k \to \mathbb{N}$. If c and d represent functions $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ then $\mathtt{rec}_{c,d}$ represents the function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ such that $f(0, \vec{y}) = g(\vec{y})$ and $f(x + 1, \vec{y}) = h(x, f(x, \vec{y}), \vec{y})$.

Q: Well, this describes PR combinators and their denotational semantics. But, how do you compute with a combinator?

A: To get the operational semantics, we define reduction rules:

(i) $(\pi_{k,i}; t_1, \ldots, t_k) \rightsquigarrow t_i$,

(ii) $\mathtt{rec}_{c,d}(\mathtt{0}, \vec{t}) \rightsquigarrow c(\vec{t})$, $\mathtt{rec}_{c,d}(\mathtt{S}(u), \vec{t}) \rightsquigarrow d(t, \mathtt{rec}_{c,d}(u, \vec{t}), \vec{t})$,

(iii) $c \rightsquigarrow c$ for every c,

(iv) if $u \rightsquigarrow u'$, $t_1 \rightsquigarrow t'_1, \ldots, t_k \rightsquigarrow t'_k$ then $\langle u; t_1, \ldots, t_k\rangle \rightsquigarrow \langle u'; t'_1, \ldots, t'_k\rangle$ and $\mathtt{rec}_{c,d}(u, t_1, \ldots, t_k) \rightsquigarrow \mathtt{rec}_{c,d}(u', t'_1, \ldots, t'_k)$.

To any combinator c representing a function $f : \mathbb{N}^k \to \mathbb{N}$, we associate a non-deterministic algorithm which works as follows: on input $(a_1, \ldots, a_k) \in \mathbb{N}^k$, iteratively apply reduction rules to the PR combinator $\langle c; S^{(a_1)}(\mathtt{0}), \ldots, S^{(a_k)}(\mathtt{0})\rangle$ (which has type $\mathbb{N}$) until no reduction rule applies. It is routine to check that we then get a PR combinator of the form $\mathtt{S}^{(n)}(\mathtt{0})$ where $n = f(a_1, \ldots, a_k)$.

Q: So, a confluence property holds for this calculus.

A: Yes. Now, a given PR combinator may contain many redexes, i.e. , there may be many sub-combinators to which a reduction can be applied. Thus, there are many computations to get the value of the function f represented by the PR combinator c on an input $(x_1, \ldots, x_k) \in \mathbb{N}^k$. In particular, there are computations corresponding to a call-by-name or a call-by-value strategy or any mixing, possibly a random one, of these strategies. What Colson proves is that every one of these computations takes time $O(x_1)$ or ... or $O(x_k)$.

3.3 Denotational semantics reveals operational gaps

Q: Could you recall the argument in Colson's proof?

A: Colson uses a non-standard denotational semantics, namely *lazy natural integers*. Formally, this model contains two kinds of objects: for every $n \in \mathbb{N}$, there is the integer $S^{(n)}(0)$ and the "in progress" integer $S^{(n)}(\perp)$ which stands for "being $\geq n$ ". There is also a limit element $S^{(\infty)}(\perp)$. The ordering relations are reduced to $S^{(n)}(\perp) \leq S^{(n+k)}(\perp) \leq S^{(\infty)}(\perp)$ and $S^{(n)}(\perp) \leq S^{(n+k)}(0)$.

Q: Yes, I remember. Colson processes the PR combinators on such lazy integers. And the core argument was an "ultimate obstinacy" result: on the pair $(S^{(\infty)}(\perp), S^{(\infty)}(\perp))$, the computation of *min* gets stuck, decrementing the same argument again and again. This prevents any alternation between arguments.

A: As it is, the proof can be viewed as non-constructive since it involves an infinite object $S^{(\infty)}(\perp)$. A constructive variant is given in [13].

Q: On lazy integers, what are the functions computed by PR combinators?

A: Looking on the sole true integers $S^{(n)}(0)$, PR combinators obviously compute all primitive recursive functions. Now, on the $S^{(n)}(\bot)$'s, they give exactly the subclass of primitive recursive functions obtained with definitions by inductions with null base case [39], i.e. inductions of the form

$$\begin{cases} f(\bot, S^{(y_1)}(\bot), \ldots) &= \bot \\ f(S^{(x+1)}(\bot), S^{(y_1)}(\bot), \ldots) &= h(S^{(x)}(\bot), f(S^{(x)}(\bot), S^{(y_1)}(\bot), \ldots), S^{(y_1)}(\bot), \ldots) \end{cases}.$$

Q: I see. The base case involves $\bot$, i.e. $S^{(0)}(\bot)$, as the induction input. Since $\bot$ means no information at all, when the induction input has value $\bot$, it is reasonable that the function gets value $\bot$.

How far have such arguments been pushed?

A: If the data is enriched with lists, the argument breaks down. In fact, René David [16] proves that the best algorithm for the *min* function can be obtained from a primitive recursive definition of *min* involving recursion over lists. Going to second-order primitive recursive definitions, i.e. working at level 1 of Gödel system T, one can also obtain the best algorithm for the *min* function, cf. [9]. Let us stress that this requires a call-by-name strategy. On the opposite side, [12] proves that call-by-value strategies for higher order primitive recursive definitions of *min*, (i.e. working at any level of Gödel system T) do not allow to get the best algorithm for *min*.

Q: Could you remind me of system T ?

A: Instead of considering primitive recursive definitions for the sole functions $\mathbb{N}^k \to \mathbb{N}$, extend such definitions to functionals. Formally, you consider a typed lambda calculus with types obtained from the basic type *nat* via the type constructor $A, B \mapsto (A \to B)$. Augment this calculus with

(1) constants $\mathtt{0}$, $\mathtt{S}$ and $\mathtt{rec}_{\alpha,\beta}$ (for integer zero, the successor function over $\mathbb{N}$ and the recursion operator defining functions of type $\alpha \to \beta$).

(2) Besides the usual beta reduction, consider new reduction rules for each $\mathtt{rec}_{\alpha,\beta}$ such that, for any terms t, u, n with respective types $\alpha \to \beta$, $\mathbb{N} \to ((\alpha \to \beta) \to (\alpha \to \beta))$ and $\mathbb{N}$,

$$\mathtt{rec}_{\alpha,\beta}\ t\ u\ 0 \rightsquigarrow t \quad , \quad \mathtt{rec}_{\alpha,\beta}\ t\ u\ (sn) \rightsquigarrow u\ n\ (\mathtt{rec}_{\alpha,\beta} t u n)$$

With the usual notations for functions, the above reductions correspond to a higher order definition by induction: $H(0) = t$, $H(n+1) = u(n, H(n))$ where H has type $\mathbb{N} \to (\alpha \to \beta)$.

Q: Please, forget the recursion combinators... What about an example expressed with higher order primitive recursive definitions of functionals?

A: Sure. Let us show how the best algorithm for *min* is second-order primitive recursive. Consider the following second-order primitive recursive definitions of

functionals $G : \mathbb{N}^{\mathbb{N}} \times \mathbb{N} \to \mathbb{N}$ and $F : \mathbb{N} \to \mathbb{N}^{\mathbb{N}}$:

$$\begin{cases} G(f,0) &= 0 \\ G(f,y+1) &= f(y)+1 \end{cases} \quad , \quad \begin{cases} F(0) &= \lambda y\,.\,0 \\ F(x+1) &= \lambda y\,.\,G(F(x),y) \end{cases}$$

These equations yield $\begin{cases} F(0)(y) &= 0 \\ F(x+1)(0) &= G(F(x),0) &= 0 \\ F(x+1)(y+1) &= G(F(x),y+1) &= F(x)(y)+1 \end{cases}$.

Letting $\varphi(x,y) = F(x)(y)$, we have $\begin{cases} \varphi(0,y) &= 0 \\ \varphi(x+1,0) &= 0 \\ \varphi(x+1,y+1) &= \varphi(x,y)+1 \end{cases}$.

These are exactly the equations for the best algorithm for the *min* function!

Q: Very nice definition. And beyond the *min* function?

A: Recall Stein's algorithm for the greatest common divisor of two integers. To take full benefit of the binary representation of integers in computers, Stein considers the following recursive equations for the gcd:

$$\begin{array}{lcll} \gcd(x,y) & = & 0 & \textit{if} \quad \textit{min}(x,y) = 0 \\ \gcd(2x,2y) & = & 2\gcd(x,y) & \textit{if} \quad \textit{min}(x,y) > 0 \\ \gcd(2x,2y+1) & = & \gcd(x,2y+1) & \textit{if} \quad x > 0 \\ \gcd(2x+1,2y) & = & \gcd(2x+1,y) & \textit{if} \quad y > 0 \\ \gcd(2x+1,2y+1) & = & \gcd(|x-y|,\min(2x+1,2y+1)) & \end{array}$$

Stein's algorithm has time complexity $O(\log(x) + \log(y))$. Moschovakis [31] proves that this complexity cannot be matched by any algorithm based on primitive recursive definitions of the gcd and using a call-by-value strategy. In fact, he gets an $O(x+y)$ lower bound for call-by-value computations even if addition, subtraction, division by 2, parity and order comparison are basic functions in such primitive recursive definitions.

4 Operational equivalence of classes of algorithms

4.1 A new problem

A: Besides all these new complexity results, a new problem emerged around the behavior of programs and especially functional programs. Instead of looking for missing algorithms in some computation model, we try to compare the families of algorithms associated to programming languages. Are they equal? Is one of them larger than the other? In this way, one can compare the operational strength of programming languages.

4.2 Lambda calculus and ASMs

Q: By the way, I remember Yuri told me that Kolmogorov-Uspensky machines and Schönhage Storage Modification Machines enriched with a pairing function encode all ASMs.

A: Yes. Such a pairing function allows to encode any data structure on the set of nodes, so that by adding a static framework on the set of nodes, you get any ASM static framework. And any dynamic vocabulary can be encoded by the dynamic successors of nodes in these models. You could also consider hyper-machines à la Kolmogorov-Uspensky or à la Schönhage: replace graph tapes by k-uniform hypergraph tapes for some $k \geq 3$.

Q: What is a k-uniform hypergraph?

A: An undirected graph can be seen as a pair (V,E) where V is a set of nodes and E is a collection of node sets of cardinality 2. A k-uniform hypergraph is a generalization where E is a collection of node sets of cardinality k.

Q: So we get two computation models which are operationally equivalent to ASMs. Well, adding a pairing function on the set of nodes seems to be an ad hoc device which was never considered seriously. Is there any other computation model which gives all sequential time small step algorithms?

A: Yes, lambda calculus. This is done in [20].

Q: How is that possible? Beta reduction is such a low level operation! Some consider it as an instruction on the level of an assembly language. Moreover, though all countable data structures can be encoded in lambda calculus, such encodings are time consuming: an unbounded number of beta reductions are necessary to mimic any basic operation in data structures.

A: Concerning your first objection, it is true that beta reduction is a very crude operation. But remember that the time unit is not really an intrinsic notion. In fact, the simulation of one step of an ASM corresponds to a succession of k beta reductions in lambda calculus for some fixed k which depends only on the simulated ASM.

As for the second objection, there is a fairness argument: the static framework is for free in ASMs, hence it should also be for free in lambda calculus. This leads us to add constants in lambda calculus to represent elements of the data structures and the diverse static operations. And also, of course, to add a new kind of reductions. For instance, with the $\mathbb{N}$ data structure, if constants a, b, c represent integers m, n, p and constant d represents a function $f : \mathbb{N}^2 \to \mathbb{N}$ such that $f(m, n) = p$ then there is a reduction $d(a, b) \rightsquigarrow c$.

Q: Adding constants in lambda calculus is a much studied topic.

A: Yes, but what is done in [20] is very peculiar. There is no variable involved in the new reductions, only ground terms with no lambda.

Q: Do you count these new reductions in your simulation?

A: As you wish. If you want to count them, this is no problem. There is a fixed ℓ (depending on the simulated ASM) such that one step of the ASM run corresponds to exactly k beta reductions plus ℓ "new" reductions.

Q: So lambda calculus is operationally equivalent to ASMs.

A: No. In lambda calculus, a beta reduction is not always a small step action since the number of redexes in a lambda term can be arbitrarily large! Beta reduction is a parallel action which goes beyond small step algorithms. Thus, the operational strength of lambda calculus lies somewhere between that of small step ASMs and that of parallel ASMs.

Q: When you simulate an ASM step by a fixed amount of successive beta reductions and "new" reductions, do these beta reductions have small step action or do they reduce arbitrarily large number of redexes simultaneously?

A: They reduce a bounded number of redexes. So they do have small step action. No problem, the simulation is fair.

4.3 The imperative language Loop$^\omega$

A: Non-Turing complete programming languages can also be compared relative to their operational strength. The first such result [14] is relative to the Loop language of Ritchie and Meyer [29] and to T_0, the level 0 of system T (which is equivalent to classical primitive recursion, extensionally as well as intentionally [10]). Loop and T_0 have the same operational strength. One step in any one of these languages can be simulated by one step in the other language.

Q: How do you prove that?

A: Well, this result has been extended in [15] to the whole system T and to an extension Loop$^\omega$ of the programming language Loop with first-class procedures (true closures) and mutable procedural variables (aka function pointers). We shall rather present this extended result.

Q: So you consider an imperative language Loop$^\omega$. That is exciting. Is it a usual one?

A: Loop$^\omega$ can easily be written using C# syntax (where anonymous first-class procedures are called delegates [ISO, 2003]) and then compiled with a C# compiler.

In Loop$^\omega$, instructions are as follows: block of commands, bounded loop, assignment, incrementation, decrementation and procedure call. In Backus-Naur notation, this can be written

$$\begin{array}{rrcl} (\textit{command}) & c & ::= & \{s\} \\ & & | & \texttt{for}\ y\ :=\ 1\ \texttt{to}\ e\ \{s\} \\ & & | & y := e \mid \texttt{inc}(y) \mid \texttt{dec}(y) \\ & & | & p(\vec{e};\vec{y}) \end{array}$$

In a sequence, one can declare mutable variables and constant variables:

$$\begin{array}{rrcl} (\textit{sequence}) & s & ::= & \epsilon \\ & & | & c;\ s \\ & & | & \texttt{cst}\ y\ =\ e;\ s \\ & & | & \texttt{var}\ y\ :=\ e;\ s \end{array}$$

In the declaration of a procedure, arguments are of two kinds : readable only (`in`) and writable only (`out`). This distinction is similar to ADA syntax and semantics:

$$(\textit{anonymous procedure}) \quad a \quad ::= \quad \texttt{proc}(\texttt{in}\ \vec{y};\texttt{out}\ \vec{z})\ \{s\}$$

Q: Looks like usual imperative programming languages. Do you have some examples of programs?

A: Sure. Here is an a Loop$^\omega$ program that computes the Ackermann function.

```
proc (in m: int, n: int; out r: int) {                          1
    proc next(in y: int, n: int; out p: int) {                  2
        p:=y;                                                   3
        inc(p);                                                 4
    }p                                                          5
    var g: proc (in int; out int);                              6
    g:=next;                                                    7
    for i:= 1 to m {                                            8
        cst h = g;                                              9
        proc aux(in y: int; out p: int) {                       10
            h(1,p);                                             11
            for j:= 1 to y {                                    12
                h(p; p);                                        13
            }p;                                                 14
        }p;                                                     15
        g:=aux;                                                 16
    }p;                                                         17
    g(n; r);                                                    18
}r;                                                             19
```

Q: A few comments would be welcome...

A: Recall Ackermann's equations:

$$\begin{cases} Ack(0,n) &= n+1 \\ Ack(m+1,0) &= Ack(m,1) \\ Ack(m+1,n+1) &= Ack(m,Ack(m+1,n)) \end{cases} .$$

Introduce $A(x) = \lambda y.Ack(x,y)$ and observe that $A(m)$ is defined by induction on m in such a way that the induction step defines $A(m+1) = \lambda n.Ack(m+1,n)$ by induction on n using function $A(m)$.

All procedures in the program are functional so that we shall write $g(x) = y$ instead of $g(x,y)$ and the same with h and *aux*. Procedure *next* in lines 2-5 is the successor function. In case $m = 0$, the loop in lines 8-17 vanishes. Hence the output is $g(n;r)$ (line 18), i.e. the successor function (line 7). Thus, we get the basic case of the induction giving $A(0)$. For $m > 0$, we argue by induction on m and assume that the loop in lines 8-17 makes g be the function $A(m)$ after m iterations of the loop. Then an $m+1$-th iteration defines a function *aux* which takes value $A(m)(1) = Ack(m,1)$ on 0 (line 11). Which is the base case of the induction on n to get $Ack(m+1,n)$. For $n > 0$, we argue by induction and assume that the loop in lines 12-14 makes *aux* be the function $\lambda n.Ack(m+1,n)$ after n iterations of the loop. So that the value of p is $aux(n) = Ack(m+1,n)$. Then an $n+1$-th iteration makes $aux(n+1)$ equal to $A(m)(Ack(m+1,n)) = Ack(m,Ack(m+1,n)) = Ack(m+1,n+1)$ (cf. line 13).

Q: Of course, you can similarly program the best time algorithm for the *min* function in Loop$^\omega$. Just follow Colson's definition of *min* at level 1 of system T.

A: No, no. In T, you get the best algorithm for *min* with the call-by-name strategy. And Loop$^\omega$ only simulates the call-by-value strategy.

Let us mention that, as far as we know, Loop$^\omega$ is the first total imperative language in which the Ackermann function can be programmed.

Back to the definition of Loop$^\omega$. Of course, some constraints are imposed by the type system [15] and allow us to give a simple operational semantics in terms of transition systems which we prove to be equivalent to the natural semantics. Also, every variable has a default value which depends only on its type.

Another operational semantics is given through the translation of a program of Loop$^\omega$ into a term of system T. This translation preserves the operational semantics. This is how we obtain a step-by-step simulation.

4.4 Translating Loop$^\omega$ programs into system T terms

First, let us write $\texttt{let}\ (x_1,\ldots,x_n) = u\ \texttt{in}\ t$ as an abbreviation for the redex $\lambda(x_1,\ldots,x_n).t\ \ u$. The intuition behind the translation is as follows: if $\vec{x}$ denotes

the variables of the environment then the block $\{c_1; \ldots; c_n\}$ is translated into

$$\texttt{let } \vec{x} = c_1^* \texttt{ in} \ldots \texttt{ let } \vec{x} = c_n^* \texttt{ in } \vec{x}$$

where the * operation is defined below.

We may be more precise, cf. [15], and annotate blocks by the sets of variables that get modified. For instance, the block $\{\texttt{inc}(x); \texttt{inc}(x)\}$ is annotated $\{\texttt{inc}(x); \texttt{inc}(x)\}_x$.

The translation is defined as follows.

Definition 4.1. *The stars e^* and $\{s\}^*_{\vec{x}}$ of an expression e and a block $\{s\}_{\vec{x}}$ in a term of system T are defined by mutual induction:*

$\bar{n}^*$	*is*	$S^n(0)$
y^*	*is*	y
$\{\}^*_{\vec{x}}$	*is*	$\vec{x}$
$(\texttt{proc}(\texttt{in } \vec{y}; \texttt{out } \vec{z})\{s\}_{\vec{z}})^*$	*is*	$\lambda\vec{y}.\{s\}^*_{\vec{z}}[\vec{z}_0/\vec{z}]$
where $\vec{z}_0$ denotes the default value for each type of z variables		
$\{\texttt{var } y := e;\ s\}^*_{\vec{x}}$	*is*	$\{s\}^*_{\vec{x}}[e^*/y]$
$\{\texttt{cst } y = e;\ s\}^*_{\vec{x}}$	*is*	$\texttt{let } y = e^* \texttt{ in } \{s\}^*_{\vec{x}}$
$\{y := e;\ s\}^*_{\vec{x}}$	*is*	$\texttt{let } y = e^* \texttt{ in } \{s\}^*_{\vec{x}}$
$\{\texttt{inc}(y);\ s\}^*_{\vec{x}}$	*is*	$\texttt{let } y = \texttt{succ}(y) \texttt{ in } \{s\}^*_{\vec{x}}$
$\{\texttt{dec}(y);\ s\}^*_{\vec{x}}$	*is*	$\texttt{let } y = \texttt{pred}(y) \texttt{ in } \{s\}^*_{\vec{x}}$
$\{p(\vec{e}; \vec{z});\ s\}^*_{\vec{x}}$	*is*	$\texttt{let } \vec{z} = p^*\ \vec{e}^* \texttt{ in } \{s\}^*_{\vec{x}}$
$\{\{s_1\}_{\vec{z}};\ s_2\}^*_{\vec{x}}$	*is*	$\texttt{let } \vec{z} = \{s_1\}^*_{\vec{z}} \texttt{ in } \{s_2\}^*_{\vec{x}}$
$\{\texttt{for } y := 1 \texttt{ to } e\ \{s_1\}_{\vec{z}};\ s_2\}^*_{\vec{x}}$	*is*	$\texttt{let } \vec{z} = \texttt{rec}(e^*, \vec{z}, [\vec{z}, y]\{s_1\}^*_{\vec{z}})$ $\texttt{in } \{s_2\}^*_{\vec{x}}$

Observe that in the imperative program some instructions cost one computation step but other ones, namely, declaring a variable and managing a loop counter, do not cost any computation step. The instructions that do cost are mapped into `let` expressions.

The following theorem ensures that the translation gives a step by step simulation.

Theorem 4.2. *For any well-typed state (c, μ) (where c is a program and μ is the environment), if $\vec{x} = dom(\mu)$ then*

$$(c, \mu) \to (c', \mu') \quad \textit{implies} \quad c^*[\mu(\vec{x})^*] \rightsquigarrow c'^*[\mu'(\vec{x})^*]$$

where $\to$ denotes one step reduction of the Loop$^\omega$ *language and $\rightsquigarrow$ denotes one step reduction in the system T language.*

Q: Does your imperative language capture all functions of system T ?

A: Yes, this is the first result. Let us recall a major result concerning system T : functions over $\mathbb{N}$ that are definable in this system correspond exactly to functions that are provably total in first-order Peano arithmetic (see Schütte's book [37]). More precisely, consider the syntactic hierarchy of fragments T_n of system T associated to type order where $o(nat) = 0$ and $o(A \to B) = \max(o(B), 1 + o(A))$. Then the class of functions representable in T_n is identical to the class of functions provably recursive in the fragment of Peano arithmetic where induction is restricted to Σ_{n+1} sentences. In particular, T_0 corresponds to the class of primitive recursive functions. We define a similar hierarchy of fragments Loop_n of Loop^ω and we show that both translations relate programs of Loop_n and terms of T_n. The particular case $n = 0$ shows that functions representable in a language with higher-order procedures but without procedural variables (which is a sublanguage of Loop_0) are primitive recursive. This corollary generalizes a previous result presented in [14] where Meyer and Ritchie's Loop language was translated into T_0.

Now, the fundamental result of [15] is that all properties about complexity of programs in system T with call-by-value are retrieved in our Loop^ω language, so *they are algorithmically equivalent*.

Q: Of course, call-by-value is sine qua non.

A: Sure. Recall that the best algorithm for *min* can be obtained in system T with call-by-name.

Q: So, this also corroborates results about system T such as the fact that the best algorithm for *min* is not captured by Loop^ω (which was proved in [12] for system T with call-by-value).

Well, the striking point is that the two languages you describe, namely Loop^ω and system T with call-by-value not only compute the same class of functions but also the same class of algorithms.

5 What is a primitive recursive algorithm?

5.1 Primitive recursive running time

A: Let us consider the class of primitive recursive functions and the question "What is a primitive recursive algorithm?".

Q: There is an answer with Turbo ASMs [6].

A: The Turbo approach to define primitive recursive ASMs constrains the syntax of ASM programs to be a clone of primitive recursive definitions. And this

constraint has far reaching consequences. In fact, what Colson proves can be transferred to this approach. In particular, the best algorithm for the *min* function is missing. Now, *we want these missing algorithms to be considered as primitive recursive algorithms*.

The solution proposed in [39] is based on a classical theorem about primitive recursive functions:

Theorem 5.1. *A function f is primitive recursive if and only if there exists a Turing machine which computes f in time bounded by a primitive recursive function.*

Q: One of my teachers used to call it an analog of Lebesgue's dominated convergence theorem...

A: This theorem ensures the equivalence of two conditions. The first one is about the function itself whereas the second one is about how to compute that function hence about algorithms.

Q: Turing machines are not a very good model for algorithms. I guess your next step is to replace them by Abstract State Machines.

A: Yes, you guessed right! The above statement which is a theorem involving Turing machines will now become a definition involving ASMs.

Definition 5.2. *An algorithm is said to be primitive recursive if and only if it corresponds to an ASM which halts in primitive recursive time.*

It's very simple! Based on the identification of algorithms to ASMs. And it captures the best algorithm for the *min* function.

Q: I presume that the output of an ASM run is the value of some dynamic symbol α when the ASM halts. Let $t(input)$ be the running time. Do you require the ASM to halt exactly at time $t(input)$ or do you consider the value of α at time $t(input)$? In the first case, the ASM halts "naturally" whereas in the second case you just ignore subsequent states.

A: There is no difference: just add $t(input)$ as a static function! A simple counter will ensure halting in due time. This is easy to be done in an ASM program. Nevertheless, we shall prefer the approach which ignores subsequent states. The reason is that we do not want to spoil the clarity of ASM programs by the management of a counter to get halting at the proper time.

Q: I see a problem with Definition 5.2. Every function f admits a primitive recursive algorithm: just put f itself in the background of the ASM! Moreover, such an ASM computes any value of f in exactly one step.

A: Right. There are trivial algorithms which carry no operational contents. This is the price to pay to the oracular nature of algorithms: oracles are what is in the

background. You just considered the trivial case where the oracle is the function itself.

Now, there is more to answer your observation. Taking the background into consideration, we can define a notion of primitive recursive algorithm relative to a fixed background.

Q: Well, well. So, the most natural way of defining primitive recursive functions is inadequate to capture a reasonable notion of primitive recursive algorithm. You base the notion of primitive recursive algorithm on primitive recursive bounds on the length of runs of ASMs.

A: Runs and length of runs are at the root of the formalization given by ASMs: the first ASM postulate is that algorithms are discrete time transition systems!

Q: Which makes your approach an intrinsic one.

5.2 Basic arithmetical primitive recursive algorithms

Q: So, you have defined from scratch a class of algorithms to compute primitive recursive functions (cf. Definition 5.2). This class is large enough to contain all known natural algorithms to get these functions using a reasonably simple background. You consider this class as the natural candidate for the desired notion of primitive recursive algorithm. Now, the definition makes no reference to any programming language. An interesting question then arises: find a programming language that implements this class of algorithms.

A: In this approach to primitive recursive algorithms, we are led to consider a computation of an algorithm as a triple $(\mathcal{A}, I, c)$ where $\mathcal{A}$ is an abstract state machine, I is an initial state and c is a function that represents the length of the part of the ASM run that we consider (ignoring all subsequent states).

Q: Oh! I heard about something like that thirty years ago, a program is equivalent to the pair of a sequential function and a computation strategy for it. As I remember this was called *sequential algorithms on concrete data structures* (cf. [3]). I think it's not sufficient to consider similar pairs. You have to require that static functions and initial interpretations of dynamic symbols be primitive recursive. Else, you would capture non-primitive recursive functions! Wouldn't you?

A: Sure. Now, there is another point to consider. To focus on programming languages, we have to compare a theoretical set of algorithms with the set of algorithms of a programming language. So, in fact, we have to look at the expressive power (in term of algorithms) of control structures in programming languages.

Q: On that matter, no doubt that ASMs have the smallest set of control structures: the conditional and nothing else.

A: Though there is no loop instruction in ASMs, it is no problem to simulate the successive steps of a loop by the successive steps of the ASM run. In fact, the ASM run is a big loop which somehow externalizes all loop instructions. Indeed, this externalization is one of the main ideas of ASMs.

Q: Simultaneous updates are allowed with ASMs, a rather uncommon feature in imperative programming languages.

A: There are two ways to cope with simultaneous updates. First, by simply allowing them in the programming language. But, as you said, this would badly depart from “real" programming languages. So we exclude simultaneous instructions (or evaluations) in our programming language and keep, as is usual, completely sequentialized computations. Now, the number of simultaneous updates is bounded. Though we cannot simulate step-by-step, we can simulate one ASM step by a fixed number of computation steps of the program. Again, we appeal to a flexible time unit.

Q: I see still another problem related to the ASM background. To balance the poorness of the control structure, ASMs allow powerful data structures: first order logical data structures. And this is not allowed in most programming languages. Of course, programming languages can emulate such data structures but there is a price for the emulation!

A: You are right. In order to stay close to real programming languages, we restrict the family of data structures of ASMs to the ones usually allowed in programming languages. This leads to the following definitions.

Definition 5.3. *An ASM is* $\mathbb{N}$*-typed if its multisort domain is reduced to* $\{0, 1\}$ *and* $\mathbb{N}$*, that is Booleans and integers.*

Now, the (static) background of an $\mathbb{N}$-typed ASM consists of some Boolean and integer elements and some functions with Boolean and integer inputs and outputs. And its dynamic vocabulary contains constant (i.e. arity zero) and function symbols to represent such elements and functions.

This is still too much to match any existing programming language. First, as mentioned in §5.1, we shall fix a reasonably simple background on data structures $\{0, 1\}$ and $\mathbb{N}$. We shall also introduce another constraint on the dynamic vocabulary: no function symbol of arity ≥ 1.

Definition 5.4. *An* $\mathbb{N}$*-typed ASM A is called* basic arithmetical *if*

i. its static framework is reduced to constants, Boolean operations on $\{0, 1\}$ *and the predecessor and successor functions on* $\mathbb{N}$*,*
ii. and its dynamic vocabulary contains only nullary symbols.

Q: This is a very restricted family of ASMs.

A: Since static functions in an ASM are evaluated for free, as concerns resource complexity, the restriction to basic arithmetical ASMs may be seen as a reasonable choice. But, first, let us give a basic arithmetical ASM program for the best algorithm to compute $min(m, n)$:

$$\left|\begin{array}{lll} \texttt{if} \quad x = 0 & \texttt{then} & res := n \\ \texttt{if} \quad x \neq 0 \wedge y = 0 & \texttt{then} & res := m \\ \texttt{if} \quad x \neq 0 \wedge y \neq 0 & \texttt{then} & \left|\begin{array}{l} x := x - 1 \\ y := y - 1 \end{array}\right. \end{array}\right.$$

Q: So there are four dynamic constants: m, n contain the inputs and are not modified, x, y take values m, n in the initial state and are the core of the computation.

A: Yes.

Q: Basic arithmetical ASMs seem to be very close to the Loop language.

A: It is true that a single computation step of basic arithmetical ASMs is a very rudimentary action. Now, the run of the ASM is an external loop which is an unbounded loop, not a bounded one like those in the Loop language. This makes basic arithmetical ASMs far more powerful than Loop. In fact, extensionally, basic arithmetical ASMs are Turing complete!

Q: How do you see that?

A: Simulate any two counter machine C. Consider static constants of type $\mathbb{N}$ to encode the finitely many states of C. Use three dynamic constants of type $\mathbb{N}$ to encode the state and the contents of the counters.

Q: It seems that basic arithmetical ASMs are exactly multicounter machines.

A: No. Counter machines are not able to compare the contents of two counters in a single computation step. Such a comparison requires a computation involving parallel decrementations very similar to the best computation of the *min* function. On the opposite, an equality test between two dynamic constants is an elementary instruction for basic arithmetical ASMs. So, basic arithmetical ASMs are operationally more powerful than multicounter machines.

Q: So you are going to define a class of primitive recursive algorithms using basic arithmetical ASMs.

A: Yes, we consider a variant of Definition 5.2.

Definition 5.5. *Basic arithmetical primitive recursive algorithms (in short* Apra*) are the algorithms associated to basic arithmetical ASMs which halt in time bounded by a primitive recursive function.*

Thus, a basic arithmetical primitive recursive algorithm can be viewed as a pair (A, f) where A is a basic arithmetical ASM and f is a primitive recursive function which bounds the lengths of the runs of A.

Q: Of course, due to Theorem 5.1, halting in primitive recursive time is equivalent to halting in time bounded by a primitive recursive function.

A: Sure. However, you may ignore the exact halting time and merely know a primitive recursive bound.

5.3 APRA and the imperative programming language LOOP$_{\texttt{halt}}$

Q: Why is APRA a reasonable class?

A: We hope to convince you that this class is the good one. Let us show you some programming language that satisfactorily implements APRA. Recall that we really care about programming languages.

We start with the the sub-language LOOP of LOOP$^{\omega}$ with no high-order procedural variables. It is similar to the Meyer-Ritchie *Loop* language [29] which extensionally expresses only primitive recursive functions. Add to this language LOOP a command to force the program to halt:

$$c \ ::= \ \ldots \mid \texttt{halt;}$$

This gives an imperative language LOOP$_{\texttt{halt}}$.

Q: Is `halt` really an earth-shaking command?

A: Recall that the best time algorithm for *min* cannot be programmed in LOOP (cf. [14] for a direct proof). But we have seen an $\mathbb{N}$-typed ASM program for that *min* algorithm. Now, here is a LOOP$_{\texttt{halt}}$ program for it.

```
if m=0 then min := 0 and halt;                    1
if n=0 then min := 0 and halt;                    2
x := m;                                           3
y := n;                                           4
for i := 1 to m do                                5
        x := x - 1;                               6
        y := y - 1;                               7
        if y=0 then min := n and halt;            8
end for;                                          9
min := m;                                        10
```

Q: Nice and so simple. So `halt` adds new algorithms. Now, how do LOOP$_{\texttt{halt}}$ and APRA operationally compare?

A: The easy direction is to associate a basic arithmetical ASM to a $\text{Loop}_{\texttt{halt}}$ program. Attach dynamic constants of type $\mathbb{N}$ to the variables in the program, including the counter variables of loop instructions and the counter for the line currently executed. For instance, the above $\text{Loop}_{\texttt{halt}}$ for the *min* function becomes the following ASM program (where || separates the instructions in a block):

`if`	$c = 1 \wedge m = 0$	`then`	$(min := 0 \parallel c := 11)$	`else` $c := 2$	1
`if`	$c = 2 \wedge n = 0$	`then`	$(min := 0 \parallel c := 11)$	`else` $c := 3$	2
`if`	$c = 3$	`then`	$(x := m \parallel c := 4)$		3
`if`	$c = 4$	`then`	$(y := n \parallel c := 5)$		4
`if`	$c = 5 \wedge i = m$	`then`	$c := 10$	`else` $c := 6$	5
`if`	$c = 6$	`then`	$(x := x - 1 \parallel c := 7)$		6
`if`	$c = 7$	`then`	$(y := y - 1 \parallel c := 8)$		7
`if`	$c = 8 \wedge y = 0$	`then`	$(min := n \parallel c := 11)$	`else` $c := 9$	8
`if`	$c = 9$	`then`	$(i := i + 1 \parallel c := 5)$		9
`if`	$c = 10$	`then`	$(min := m \parallel c := 11)$		10

Observe that when $c = 11$ the ASM program does nothing. Thus, the ASM assignment $c := 11$ corresponds to the `halt` command.

Q: Ok, it is an easy exercise to devise an ASM program such that the obtained basic arithmetical ASM simulates step by step the execution of any $\text{Loop}_{\texttt{halt}}$ program. So every algorithm associated to some $\text{Loop}_{\texttt{halt}}$ program is in Apra. What about the converse inclusion?

A: Let A be a basic arithmetical ASM with ASM program π_A and running time bounded by a primitive recursive function f. To get a $\text{Loop}_{\texttt{halt}}$ program which captures the operational contents of A, we proceed in three main steps: 1) define a shell program, 2) get a core program, 3) “inject" the shell program into the core program.

To get the shell program, we first translate the ASM program π_A into a $\text{Loop}_{\texttt{halt}}$ program. Then we duplicate the variables associated to the ASM dynamic constants. These new variables will represent the state before an update. Thus, comparing old and new values, we can detect a fixed point of the ASM run.

Q: But there is nothing but conditionals and updates in π_A. So this is quite a degenerate Loop program!

A: Yes, this is a sequence of conditionals and updates that mimic the π_A program. This is, in fact, the best simulation we can manage since we simulate simultaneous updates by sequences. Thus, we loosen the step by step simulation: for some fixed k, one ASM step is simulated by $\leq k$ steps of the Loop program. We can also get exactly k steps if we want.

Q: So far, there is still no loop. And what you get is not a very attractive program.

A: We agree, this is not the usual way to construct programs (for the time being!). But we just want to show that this programming language is expressive enough as concerns primitive recursive algorithms.

Now, we introduce the core program.

Recall that the running time of the ASM run is bounded by some primitive recursive function f. The core program is any Loop program with running time greater than or equal to f. There are such core programs: since f is primitive recursive there exists a Loop program that computes f; complete this program with a loop bounded by the value of f just computed.

Q: The running time of the core program can be far much greater than f since f may not have good implementations in Loop.

A: This is where the magic command `halt` comes in!

We "inject" the shell program into the core program. This means that at each step of the core program we also execute the shell program to mimic one step of the run of the given basic arithmetical ASM. Using the duplicated variables, we know when the ASM run halts. Besides the shell program we also inject in the core program some conditional instruction

$$\texttt{if } C \texttt{ then halt}$$

where condition C uses the variables associated to the ASM (plus their duplicates) to check whether the simulated ASM run halts or not.

Q: Summing up, the core program has a running time long enough so that the shell program can be iteratively executed a number of times at least equal to the ASM running time. And the command `halt` allows to stop the execution of the core program at the right time. Now, how do you define this "injection" of the shell program into the core program?

A: Let P be a Loop program and Q be a $\text{Loop}_{\texttt{halt}}$ program. We define the insertion $P[Q]$ of Q in P by induction on the length of P:

P	$\equiv$	$x := e;\ com_s$
$P[Q]$	$\equiv$	$Q\ x := e;\ com_s[Q]$
P	$\equiv$	$\texttt{if } e \texttt{ then } com_1 \texttt{ else } com_2 \texttt{ endif};\ com_s$
$P[Q]$	$\equiv$	$Q\ \texttt{if } e \texttt{ then } com_1[Q] \texttt{ else } com_2[Q] \texttt{ endif};\ com_s[Q]$
P	$\equiv$	$\texttt{loop } expr_{int} \texttt{ do } com \texttt{ endloop};\ com_s$
$P[Q]$	$\equiv$	$Q\ \texttt{loop } expr_{int} \texttt{ do } com[Q]\ Q \texttt{ endloop};\ com_s[Q]$

Q: Semicolons are missing after Q in your table.

A: No. In fact, Q represents a piece of code ending with a semicolon!

Q: Ok. You do not obtain exact operational simulation. The environment of the $\text{Loop}_{\texttt{halt}}$ program is richer than that of the simulated basic arithmetical ASM.

Q: Right. We obtain the following result.

Theorem 5.6. *Let g be a function from $\mathbb{N}^k$ to $\mathbb{N}$ and A be an* Apra *algorithm which computes g in time bounded by a primitive recursive function f. Then there is a* $\text{Loop}_{\texttt{halt}}$ *program that simulates A and runs in time $O(f)$.*

This leads to consider escape commands as good control structure for programming languages (see [35] for a pedagogical discussion on that subject). Besides `halt`, one can consider kind of “escape from a loop" or “escape from a k-nested loop" or introduce more sophisticated control structures such that exception (see [1] for instance).

5.4 Functional implementation of Apra

Q: Well, what you showed me makes good use of the fact that the Loop language is close to ASMs with the update (or assignment) notion. But what about a functional implementation of Apra?

A: Following [14] and [15], we extract from Gödel system T the core of the functional language that simulates the class Apra. The fragment of system T which captures Apra is exactly the language induced by the definition of *primitive recursion with variable parameters* (see [33]) with a mixed call-by strategy (by value on β–reduction and by-name for `rec` and `if` reductions).

Q: Hum... A functional language with two kinds of reductions. Do you know any language which truly shares such a feature?

A: In fact, in each usual functional language designers have added a way to use the opposite reduction strategy. For instance, Haskell for call-by-need and Ocaml for call-by-value for instance. This is no coincidence.

Q: So, you start from a Loop_{halt} program and you translate it to a system T term?

A: No. Roughly speaking, a halting command does not go well with functional programming nor with a mathematical model. The simulation relies on another control structure related to the bounded loop itself that can be called *conditional bounded loop* and denoted by LoopC.

Q: This kind of control structure has been used in PL/1.

A: This control structure can also be viewed as a conditional `exit` from a loop which can appear anywhere among the instructions inside the loop.

For convenience, we shall consider *down-to* bounded loops: loops which go downto 1.

$$\textit{command} \quad ::= \quad \ldots \mid \texttt{for}\ x_i := t\ \texttt{downto}\ 1\ \texttt{onlyIf}\ b\ \{s\}$$

The informal operational semantics is “loop until 1 is reached or b is false".

Before entering the translation, let us mention the definition of expressions in LoopC :

$$\begin{array}{lcl} e & ::= & e_{int} \mid e_{bool} \\ e_{int} & ::= & \bar{n} \mid x \mid x-1 \mid x+1 \\ e_{bool} & ::= & \texttt{true} \mid \texttt{false} \mid x_i = x_j \mid \neg b \mid b_1 \wedge b_2 \end{array}$$

The desired translation is as follows.

Definition 5.7. *The translation (denoted by $^+$) of a* LoopC *program with variables $\vec{x} = (x_1, \ldots, x_n)$ into a term in system T is defined by induction on expressions and commands as follows:*

$$\begin{array}{lll} \bar{n}^* & is & S^n(0) \\ \overline{\texttt{true}}^+ & is & \texttt{true} \\ \overline{\texttt{false}}^+ & is & \texttt{false} \\ x_i^+ & is & x_i \\ (x_i+1)^+ & is & \texttt{succ}(x_i) \\ (x_i-1)^+ & is & \texttt{pred}(x_i) \\ \{\}^+ & is & \vec{x} \\ \{x_i = e;\ s\}^+ & is & \texttt{let}\ x_i\ =\ e^+\ \texttt{in}\ \{s\}^+ \\ \{c;\ s\}^+ & is & \texttt{let}\ \vec{x}\ =\ c^+\ \texttt{in}\ \{s\}^+ \\ & & \textit{if c is not an assignment} \\ (\texttt{if}\ b\ \{s_1\}\ \texttt{else}\ \{s_2\})^+ & is & \texttt{if}(b^+, \{s_1\}^+, \{s_2\}^+) \\ (\texttt{for}\ x_i\ \texttt{in}\ e\ \texttt{downto}\ 1\ \texttt{onlyIf}\ b\ \ \{s\})^+ & is & \\ \multicolumn{3}{l}{\qquad \texttt{rec}_{N,N}(e^+, \lambda\vec{x}.\vec{x},\ [y, x_i]\lambda\vec{x}.\texttt{if}(b^+, \texttt{let}\ \vec{x} = \{s\}^+\ \texttt{in}\ y\ \vec{x},\ \vec{x}))\ \ \vec{x}} \end{array}$$

Q: I don't understand, your translation of the conditional bounded loop gives a term in the system T at level 1 (due to $\texttt{rec}_{N,N}$). Is the target language beyond primitive recursion?

A: A priori, you're right! But if you analyze the term you will see that it doesn't use the full power of level one of system T. In fact, the translated term encodes a primitive recursive schema called primitive recursion with variable parameters which has the form

$$\begin{array}{lcl} f(0, \vec{y}) & = & g(\vec{y}) \\ f(x+1, \vec{y}) & = & h(x, f(x, j(x, \vec{y})), \vec{y}) \end{array}$$

where g, h and j are also defined by primitive recursion with variable parameters.

Q: This looks much like the second-order primitive recursion to get the best algorithm for *min*, cf. §3.3.

A: This recursion schema offers a new control structure for the functional language (of primitive recursion). Though it is not expressible as such in system

T, using level 1 of system T, we can mimic it. Now, observe that we do not use level 1 for its extensional power but we use it for its operational power which goes beyond that of level 0 even for primitive recursive functions.

Q: Ok, but in system T with call-by-value strategy, you mentioned earlier that there is no way to obtain the good algorithm for the *min* function!

A: Yes, with such kind of language call-by-value is an handicap. Other strategies may enlarge the algorithmic expressive power. The good strategy is the one that reduces `rec` by-name and `let` by-value.

Q: So, your result is similar to that of Theorem 4.2 relative to the translation of Loop$^\omega$ into system T. One reduction step in LoopC is simulated by one reduction step in T. Is that correct?

A: In fact, the simulation is only lockstep: one step in LoopC is translated by one or two steps in the `rec` term.

Theorem 5.8. *If* $\langle c, (\vec{x}, \vec{n})\rangle \to \langle c_1, (\vec{x}, \vec{n_1})\rangle$ *then* $c^+[\vec{n}^+/\vec{x}] \rightsquigarrow^{\leq 2} c_1^+[\vec{n_1}^+/\vec{x}]$ *where* $\to$ *denotes one reduction step of the* LoopC *language and* $\rightsquigarrow^{\leq i}$ *denotes a sequence of at most i reduction steps in the system T language.*

Acknowledgments.
Many thanks to Yuri Gurevich for numerous illuminating discussions. Many thanks too to Charles R. Wallace and Benjamin Rossman for suggestions and corrections to the first draft of this paper.

References

[1] *Consolidated ADA Reference Manual: Language and Standard Libraries*, Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[2] Ph. Andary, B. Patrou, and P. Valarcher. A representation theorem for primitive recursive algorithms, *Fundamenta Informaticae*, 107(4): 313-330 (2011).

[3] Gérard Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20(3):265–321, 1982.

[4] Therese C. Biedl, Jonathan F. Buss, Erik D. Demaine, Martin L. Demaine, Mohammad Taghi Hajiaghayi and Tomás Vinar. Palindrome recognition using a multidimensional tape. *Theoretical Computer Science*, 302(1-3):475–480, 2003.

[5] Lenore Blum, Mike Shub and Steve Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin American Mathematical Society*, 21(1):1–46, 1989.

[6] E. Börger and R. Stärk, *Abstract state machines: a method for high-level system design and analysis*, Springer-Verlag, 2003.

[7] Olivier Bournez, Manuel L. Campagnolo, Daniel S. Graça and Emmanuel Hainry. Polynomial differential equations compute all real computable functions on computable compact intervals. *Journal of Complexity*, 23(3):157–166, 2007.

[8] Stephen Brookes and Denis Dancanet. Sequential algorithms, deterministic parallelism, and intensional expressiveness. *22nd Symposium on Principles of Programming Languages (POPL'95)*, 13–24, 1995.

[9] Loïc Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83(1):57–69, 1991. Preliminary version in Proceedings ICALP 1989, *Lecture Notes in Computer Science*, 372:194–206, 1989.

[10] Loïc Colson. Représentation intensionnelle d'algorithmes dans les systèmes fonctionnels. *Thèse de doctorat*, Université Paris 7, 1991.

[11] Loïc Colson. A unary representation result for system T. *Annals of Mathematics and Artificial Intelligence*, 16:385–403, 1996.

[12] Loïc Colson and Daniel Fredholm. System T, call-by-value and the minimum problem. *Theoretical Computer Science*, 206:301–315, 1998.

[13] Thierry Coquand. Une preuve directe du thérorème d'ultime obstination. *Comptes Rendus de l'Académie des Sciences*, Série I, 314:389–392, 1992.

[14] Tristan Crolard, Samuel Lacas, and Pierre Valarcher. On the expressive power of the Loop language. *Nordic Journal of Computing*, 13(1-2):46–57, 2006.

[15] Tristan Crolard, Emmanuel Polonowski and Pierre Valarcher. Extending the loop language with higher-order procedural variables. *ACM Transactions on Computational Logic* 10(4):1–37 (2009)

[16] René David. Un algorithme primitif récursif pour la fonction Inf. *Comptes Rendus de l'Académie des Sciences*, Série I, 317:899–902, 1993.

[17] René David. The Inf function in the system F. *Theoretical Computer Science*, 135:423-431, 1994.

[18] René David. On the asymptotic behaviour of primitive recursive algorithms. *Theoretical Computer Science*, 266(1-2):159–193, 2001.

[19] René David. Decidability results for primitive recursive algorithms. *Theoretical Computer Science*, 300(1-3):477–504, 2003.

[20] Marie Ferbus and Serge Grigorieff. ASMs and Operational Algorithmic Completeness of Lambda Calculus. *Fields of Logic and Computation*, Nachum Dershowitz, Wolfgang Reisig editors. Lecture Notes in Computer Science 6300:301–327, Springer, 2010.

[21] Serge Grigorieff and Pierre Valarcher. Evolving multialgebras unify all usual sequential computation models. *STACS 2010*, 301–327, Jean-Yves Marion, Thomas Schwentick, editors, 2010.

[22] Serge Grigorieff and Pierre Valarcher. Functionals using Bounded Information and the Dynamics of Algorithms. *LICS 2012*.

[23] Serge Grigorieff and Pierre Valarcher. Computation models and ASM frameworks. *In preparation*.

[24] Yuri Gurevich. Evolving algebras: an attempt to discover semantics. *Bulletin of EATCS*, 43:264–284, June 1991.

[25] Yuri Gurevich. The Sequential ASM Thesis. *Bulletin of EATCS*, 43:264–284, February 1999.

[26] Yuri Gurevich. What Is an Algorithm? *SOFSEM*, Lecture Notes in Computer Science, 7147:31–42, 2012.

[27] F. C. Hennie. One-Tape, Off-Line Turing Machine Computations. *Information and Control*, 8(6):553–578, 1965.

[28] Kurt Schütte. *Fundamental algorithms*. The Art of Computer Programming, volume 1. Addison-Wesley, 1968.

[29] Albert R. Meyer and Dennis M. Ritchie. The complexity of loop programs *Proc. 22nd National ACM Conference*, 465–470, 1976.

[30] David Michel and Pierre Valarcher. A total functional programming language that computes APRA. *Studies in Weak Arithmetic, CSLI Lecture Notes*, 196:1–19, Stanford, 2009.

[31] Yiannis Moschovakis. On primitive recursive algorithms and the greatest common divisor function. *Theoretical Computer Science*, 301 (1-3):1–30, 2003 .

[32] Wolfgang J. Paul. Kolmogorov Complexity and Lower Bounds. *Proc. Fundamentals of Computation Theory (FCT)*, Budach, L., editor, pages 325–334, Berlin/Wendisch-Rietz Akademie-Verlag, 1979.

[33] Rózsa Péter. *Recursive Functions*. Academic Press, 1967.

[34] H. G. Rice. Recursive real numbers. *Proceedings American Mathematical Society*, 5:784–791, 1954.

[35] Eric S. Roberts. Loop exits and structured programming: reopening the debate. *SIGCSE'95: Proceedings of the twenty-sixth SIGCSE technical symposium on Computer Science Education*, 268–272, 1995.

[36] Hartley Rogers Jr. *Theory of recursive functions and effective computability*. McGraw-Hill, 1967.

[37] Kurt Schütte. *Proof theory*. Springer, 1977.

[38] Alan Mathison Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proceedings London Math. Soc., series 2*, 42:230–265, 1936. Correction ibid. 43, pp 544-546 (1937).

[39] Pierre Valarcher. A complete characterization of primitive recursive intensional behaviours. *Theoretical Informatics and Applications*, 42(1):62–82, 2008.

Shuffling and Unshuffling

Dane Henshall
School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1
Canada
dslhensh@uwaterloo.ca

Narad Rampersad
Department of Math/Stats
University of Winnipeg
515 Portage Avenue
Winnipeg, MB, R3B 2E9
Canada
narad.rampersad@gmail.com

Jeffrey Shallit
School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1
Canada
shallit@cs.uwaterloo.ca

Abstract

We consider various shuffling and unshuffling operations on languages and words, and examine their closure properties. Although the main goal is to provide some good and novel exercises and examples for undergraduate formal language theory classes, we also provide some new results and mention some open problems.

1 Introduction

Two kinds of shuffles are commonly studied: perfect shuffle and ordinary shuffle.

Bulletin of the EATCS no 107, pp. 131–142, June 2012

For two words $x = a_1a_2\cdots a_n$, $y = b_1b_2\cdots b_n$ of the same length, we define their *perfect shuffle* $x \amalg y = a_1b_1a_2b_2\cdots a_nb_n$. For example, `term` $\amalg$ `hoes` = `theorems`. Note that $x \amalg y$ need not equal $y \amalg x$. This definition is extended to languages as follows:

$$L_1 \amalg L_2 = \bigcup_{\substack{x\in L_1,\ y\in L_2\\ |x|=|y|}} \{x \amalg y\}.$$

If x^R denotes the reverse of x, then note that $(x \amalg y)^R = y^R \amalg x^R$.

It is sometimes useful to allow $|y| = |x| + 1$, where $x = a_1\cdots a_n$, $y = b_1\cdots b_{n+1}$, in which case we define $x \amalg y = a_1b_1\cdots a_nb_nb_{n+1}$.

The *ordinary shuffle* $x \,\text{III}\, y$ of two words is a finite *set*, the set of words obtainable from merging the words x and y from left to right, but choosing the next symbol arbitrarily from x or y. More formally,

$$x \,\text{III}\, y = \{z \ :\ z = x_1y_1x_2y_2\cdots x_ny_n \text{ for some } n \geq 1 \text{ and words } x_1,\ldots,x_n,y_1,\ldots,y_n \text{ such that } x = x_1\cdots x_n \text{ and } y = y_1\cdots y_n\}.$$

This definition is symmetric, and $x \,\text{III}\, y = y \,\text{III}\, x$. The definition is extended to languages as follows:

$$L_1 \,\text{III}\, L_2 = \bigcup_{x\in L_1,\ y\in L_2} (x \,\text{III}\, y).$$

Shuffle is associative; we have

$$(L_1 \,\text{III}\, L_2) \,\text{III}\, L_3 = L_1 \,\text{III}\, (L_2 \,\text{III}\, L_3)$$

for all languages L_1, L_2, L_3.

(As a mnemonic, the symbol III is larger than $\amalg$ in size, and similarly III generally produces a set larger in cardinality than $\amalg$.)

As is well-known, the shuffle (resp., perfect shuffle) of two regular languages is regular, and the shuffle (resp., perfect shuffle) of a context-free language with a regular language is context-free. Perhaps the easiest way to see all these results is by using morphisms and inverse morphisms, and relying on the known closure properties of these transformations, as follows:

If $L_1, L_2 \subseteq \Sigma^*$, create a new alphabet Σ' by putting primes on all the letters of Σ. Define $h_1(a) = h_2(a') = a$ and $h_1(a') = h_2(a) = \epsilon$ for $a \in \Sigma$. Define $h(a) = h(a') = a$ for $a \in \Sigma$. Then

$$L_1 \,\text{III}\, L_2 = h(h_1^{-1}(L_1) \cap h_2^{-1}(L_2)).$$

In a similar way,

$$L_1 \amalg L_2 = h(h_1^{-1}(L_1) \cap h_2^{-1}(L_2) \cap (\Sigma\Sigma')^*).$$

However, the shuffle (resp., perfect shuffle) of two context-free languages need not be context-free. For example, if $L_1 = \{a^m b^m \ : \ m \geq 1\}$ and $L_2 = \{c^n d^n \ : \ n \geq 1\}$, then $L := L_1 \amalg L_2$ is not a CFL. If it were, then $L \cap a^+c^+b^+d^+ = \{a^m c^n b^m d^n \ : \ m, n \geq 1\}$ would be a CFL, which it isn't (via the pumping lemma).

Similarly, if $L_3 = \{a^m b^{2m} \ : \ m \geq 1\}$ and $L_4 = \{a^{2n} b^n \ : \ n \geq 1\}$, then $L_3 \shuffle L_4 = \{a^{2n}(ba)^n b^{2n} \ : \ n \geq 1\}$, which is clearly not a CFL.

For these, and other facts, see [1].

2 Self-shuffles

Instead of shuffling languages together, we can take a language and shuffle (resp., perfect shuffle) each word with itself. Another variation is to shuffle each word with its reverse. This gives four different transformations on languages, which we call self-shuffles:

$$\begin{aligned} \mathrm{ss}(L) &= \bigcup_{x \in L} \{x \amalg x\} \\ \mathrm{pss}(L) &= \bigcup_{x \in L} x \shuffle x \\ \mathrm{ssr}(L) &= \bigcup_{x \in L} \{x \amalg x^R\} \\ \mathrm{pssr}(L) &= \bigcup_{x \in L} x \shuffle x^R. \end{aligned}$$

We would like to understand how these transformations affect regular and context-free languages. We obtain some results, but other questions are still open.

Theorem 1. *If L is regular, then* $\mathrm{ss}(L)$ *need not be context-free.*

Proof. We show that $\mathrm{ss}(\{0,1\}^*)$ is not a CFL. Suppose it is, and consider $L' = \mathrm{ss}(\{0,1\}^*) \cap R$, where $R = \{01^a0^{b+1}1^{c+1}0^d1 \ : \ a, b, c, d \geq 1\}$. Since R is regular, it suffices to show that L' is not context-free.

Now consider an arbitrary word $w \in L'$. Then $w = 01^a0^{b+1}1^{c+1}0^d1$ for some $a, b, c, d \geq 1$, and there exists a $y \in \{0,1\}^*$ such that $w \in y \amalg y$. The structure of w allows us to determine y. Let y_1 and y_2 be copies of y such that $w \in y_1 \amalg y_2$, and the first letter of w is taken from y_1.

The first symbol of y is evidently 0. It follows that the prefix 01^a of w is taken entirely from y_1, since the 0 is taken from y_1 by definition and the first symbol of y_2 is 0. Therefore 01^a is a prefix of y_1.

It follows that y_2 also contains 01^a as a prefix, and since $a \geq 1$ this is only possible if the first 0 of y_2 is located in the 0^{b+1} block of w. Otherwise, y_2 would

be a subsequence of 0^d1 and y_1 would have $01^a0^{b+1}1^{c+1}$ as a prefix (implying that $y_1 \neq y_2$). Furthermore, the second symbol of y_2 being 1 implies that exactly one of the 0's in the 0^{b+1} block is from y_2. Thus the rest are from y_1 and 01^a0^b is a prefix of y_1.

Note that y_1 and y_2 both end in 1, and w ends in 0^d1. By the same logic as before, we can conclude that 0^d1 is a suffix of exactly one of them, and that the other ends in the 1^{c+1} block. Thus y_2 contains 0^d1 as a suffix and y_1 ends in the 1^{c+1} block (otherwise, $y_1 \neq y_2$).

Finally, since the second last symbol of y_1 is 0 and y_1 ends in the 1^{c+1} block, we can conclude that y_1 contains exactly one 1 from the 1^{c+1} block and that $y_1 = 01^a0^b1$. Unshuffling y_1 from w yields $y_2 = 01^c0^d1$.

Recall that $y_1 = y_2$. So,

$$y_1 = 01^a0^b1 = 01^c0^d1 = y_2$$

and since $a, b, c, d \geq 1$ we know that

$$a = c \qquad \text{and} \qquad b = d.$$

If $w \in L'$ then

$$\begin{aligned} w &= 01^a0^{b+1}1^{c+1}0^d1 \\ &= 01^a0^{d+1}1^{a+1}0^d1 \\ &= 01^a0^d(01)1^a0^d1. \end{aligned}$$

Since w was arbitrary, we have

$$\begin{aligned} L' &= \{01^a0^{b+1}1^{c+1}0^d1 : a = c, b = d, \text{ and } a, d \geq 1\} \\ &= \{01^n0^m(01)1^n0^m1 : m, n \geq 1\}, \end{aligned}$$

which is clearly not a CFL, using the pumping lemma. □

Remark 2. In a previous version of this paper, proving that ss($\{0, 1\}^*$) is not context-free was listed as an open problem. After this was solved by D. Henshall, a solution was given by Georg Zetzsche independently.

Similarly, we can show

Theorem 3. $L = \bigcup_{w\in\{0,1\}^*}(w \amalg w \amalg w)$ *is not context-free.*

Proof. We use Ogden's lemma. Consider

$$L = \{w \amalg w \amalg w \ : \ w \in \{0, 1\}^*\} \cap 0^*10^*10^*1.$$

Pick $s = 0^n10^n10^n1$ in L to pump. Write $s = uvxyz$ and mark the middle block of 0's. If v begins in the middle block of 0's, then pump up to obtain $s' = 0^n10^j10^k1$, where $n < j$ and $n \leq k$. We can't have $s' \in w \amalg w \amalg w$ because the first w (the one ending at the first 1) is too short. If v begins in the first block of 0's, then y occurs in the middle block, so now pump down to obtain $s' = 0^i10^j10^n1$, where $i \leq n$ and $j < n$. Again, we can't have $s' \in w \amalg w \amalg w$, because the third w (the one ending at the third 1) must contain all of the 0's immediately preceding the final 1, and hence is too long. □

Clearly ss($\{0,1\}^*$) is in NP, since given a word w we can guess x, guess the order in which x is shuffled with itself, and hence test if $w \in x \amalg x$. However, we do not know whether we can solve membership for ss($\{0,1\}^*$) in polynomial time. This question is apparently originally due to Jeff Erickson [2], and we learned about it from Erik Demaine.

Open Problem 4. Is ss($\{0,1\}^*$) in P?

We mention a few related problems. Using dynamic programming, Mansfield [4] showed that given words w, x, y, one can decide in polynomial time if $w \in x \amalg y$: for each i and j, determine if $w[1..i+j]$ is in the shuffle of $x[1..i]$ with $y[1..j]$. Later, the same author [5] and, independently, Warmuth and Haussler [6] showed that, given n and words $w, x_1, x_2, \ldots, x_n$, deciding if $w \in x_1 \amalg x_2 \amalg \cdots \amalg x_n$ is NP-complete. However, the decision problem implied by Open Problem 4 asks something different: given w, does there exist x such that $w \in x \amalg x$?

Open Problem 5. Determine a simple closed form for

$$a_k(n) := \left| \bigcup_{x \in \{0,1,\ldots,k-1\}^n} (x \amalg x) \right|.$$

The first few terms are given as follows:

n	0	1	2	3	4	5	6	7	8	9
$a_2(n)$	1	2	6	22	82	320	1268	5102	20632	83972
$a_3(n)$	1	3	15	93	621	4425	32703	248901		
$a_4(n)$	1	4	28	244	2332	23848	254416			
$a_5(n)$	1	5	45	505	6265	83225				
$a_6(n)$	1	6	66	906	13806	225336				

Clearly $a_i(0) = 1$, $a_i(1) = i$, and $a_i(2) = 2i^2 - i$. Empirically we have $a_i(3) = 5i^3 - 5i^2 + i$, $a_i(4) = 14i^4 - 21i^3 + 5i^2 + 3i$, and $a_i(5) = 42i^5 - 84i^4 + 32i^3 + 21i^2 - 10i$. This suggests that $a_i(n) = \frac{\binom{2n}{n}}{n+1} i^n - \binom{2n-1}{n+1} i^{n-1} + O(i^{n-2})$, but we do not have a proof.

3 Perfect self-shuffle

We can consider the same question for perfect shuffle. We define

$$\mathrm{pss}(L) = \bigcup_{x \in L} \{x \amalg x\}.$$

Theorem 6. *Both the class of regular languages and the class of context-free languages are closed under* pss.

Proof. Use the fact that $\mathrm{pss}(L) = h(L)$, where h is the morphism mapping $a \to aa$ for each letter a. □

4 Self-shuffle with reverse

We now characterize those words y that can be written as a shuffle of a word with its reverse; that is, as a member of the set $x \amalg x^R$.

An *abelian square* is a word of the form xx' where x' is a permutation of x.

Theorem 7. *(a) If there exists x such that $y \in x \amalg x^R$, then y is an abelian square.*
(b) If y is a binary abelian square, then there exists x such that $y \in x \amalg x^R$.

We introduce the following notation: if $w = a_1 a_2 \cdots a_n$, then by $w[i..j]$ we mean the factor $a_i a_{i+1} \cdots a_j$.

Proof. (a) If y is the shuffle of x with its reverse, then the first half of y must contain some prefix of x, say $x[1..k]$. Then the second half of y must contain the remaining suffix of x, say $x[k+1..n]$. Then the second half of y must contain, in the remaining positions, some prefix of x, reversed. But by counting we see that this prefix must be $x[1..k]$. So the first half of y must contain the remaining symbols of x, reversed. This shows that the first half of y is just $x[1..k]$ shuffled with $x[k+1..n]^R$, and the second half of y is just $x[k+1..n]$ shuffled with $x[1..k]^R$.

So the second half of y is a permutation of the first half of y.

(b) It remains to see that every binary abelian square can be obtained in this way. To see this, note that if x contains j 0's and $n-j$ 1's, then we can get y by shuffling $0^j 1^{n-j}$ with its reverse. We get the 0's in x by choosing them from $0^j 1^{n-j}$, and we get the 1's in x by choosing them from $(0^j 1^{n-j})^R$. □

Remark 8. The word 012012 is an example of a ternary abelian square that cannot be written as an element of $w \amalg w^R$ for any word w.

Remark 9. The preceding proof gives another proof of the classic identity

$$\binom{2n}{n} = \binom{n}{0}^2 + \cdots + \binom{n}{n}^2.$$

To see this, we use the following bijections: the binary words of length $2n$ having exactly n 0's (and hence n 1's) are in one-one correspondence with the abelian squares of length $2n$, as follows: take such a word and complement the last n bits. This transformation is clearly invertible. Thus there are $\binom{2n}{n}$ binary abelian squares of length $2n$.

On the other hand, there are $\binom{n}{i}^2$ words that are abelian squares and have a first and last half, each with i 0's. Summing this from $i = 0$ to n gives the result.

Corollary 10. *The language*

$$\mathrm{ssr}(\{0,1\}^*) = \bigcup_{x \in \{0,1\}^*} (x \,\text{Ш}\, x^R)$$

is not a CFL, but is in P.

Proof. From above, intersecting $\mathrm{ssr}(\{0,1\}^*)$ with $0^+1^+0^+1^+$ gives

$$\{0^m 1^n 0^{m+2k} 1^n \ : \ m, n \geq 1 \text{ and } k \geq 0\} \ \cup \ \{0^m 1^{n+2k} 0^m 1^n \ : \ m, n \geq 1 \text{ and } k \geq 0\}.$$

Now the pumping lemma applied to $z = 0^n 1^n 0^n 1^n$ shows this is not a CFL.

Since we can easily test if a string is an abelian square by counting the number of 0's in the first half, and comparing it to the number of 0's in the second half, it follows that $\mathrm{ssr}(\{0,1\}^*)$ is in P. □

As before, we can define

$$b_k(n) := \left| \bigcup_{x \in \{0,1,\ldots,k-1\}^n} (x \,\text{Ш}\, x^R) \right|.$$

For $k = 2$, our results above explain $b_k(n)$, but we do not know a closed form for larger k.

The first few terms are given as follows:

n	0	1	2	3	4	5	6	7	8	9
$b_2(n)$	1	2	6	20	70	252	924	3432	12870	48620
$b_3(n)$	1	3	15	87	549	3657	25317	180459		
$b_4(n)$	1	4	28	232	2116	20560	208912			
$b_5(n)$	1	5	45	485	5785	73785				
$b_6(n)$	1	6	66	876	12906	203676				

Clearly $b_i(0) = 1$, $b_i(1) = i$, and $b_i(2) = 2i^2 - i$. Empirically, we have $b_i(3) = 5i^3 - 6i^2 + 2i$, $b_i(4) = 14i^4 - 27i^3 + 17i^2 - 3i$, and $b_i(5) = 42i^5 - 110i^4 + 94i^3 - 17i^2 - 8i$. This suggests that $b_i(n) = \frac{\binom{2n}{n}}{n+1} i^n - \left(\binom{2n-1}{n-1} - 2^{n-1}\right) i^{n-1} + O(i^{n-2})$, but we do not have a proof.

5 Perfect self-shuffle with reverse

We now consider the operation $w \to w \sqcup\!\sqcup w^R$ applied to languages. Recall that $\mathrm{pssr}(L) = \bigcup_{x \in L} \{x \sqcup\!\sqcup x^R\}$.

Theorem 11. *If L is regular then* pssr(L) *is not necessarily regular.*

Proof. Let $L = 0^+10^+$. Then $\mathrm{pssr}(L) \cap 0^+110^+ = \{0^n110^n \ : \ n \geq 2\}$, which is clearly not regular. □

Theorem 12. *If L is context-free then* pssr(L) *is not necessarily context-free.*

Proof. Let $L = \{0^m1^m2^n3^n \ : \ m, n \geq 1\}$. Then $\mathrm{pssr}(L) \cap (03)^+(12)^+(21)^+(30)^+ = \{(03)^n(12)^n(21)^n(30)^n \ : \ n \geq 1\}$, and this language is easily seen to be non-context-free. □

Theorem 13. *If L is regular then* pssr(L) *is necessarily context-free.*

We defer the proof of Theorem 13 until Section 6.4 below.

6 Unshuffling

Given a finite word $w = a_1a_2 \cdots a_n$ we can decimate it into its odd- and even-indexed parts, as follows:

$$\begin{aligned} \mathrm{odd}(w) &= a_1a_3 \cdots a_{n-((n+1) \bmod 2)} \\ \mathrm{even}(w) &= a_2a_4 \cdots a_{n-(n \bmod 2)} \end{aligned}$$

Similarly, given $w = a_1a_2 \cdots a_n$ we can extract its first and last halves, as follows:

$$\begin{aligned} \mathrm{fh}(w) &= a_1a_2 \cdots a_{\lfloor n/2 \rfloor} \\ \mathrm{lh}(w) &= a_{\lfloor n/2 \rfloor + 1} \cdots a_n \end{aligned}$$

We now turn our attention to four "unshuffling" operations:

$$\begin{aligned} \mathrm{bd}(w) &= \mathrm{odd}(w)\mathrm{even}(w) \\ \mathrm{bdr}(w) &= \mathrm{odd}(w)\mathrm{even}(w)^R \\ \mathrm{bdi}(w) &= \mathrm{fh}(w) \sqcup\!\sqcup \mathrm{lh}(w) \\ \mathrm{bdir}(w) &= \mathrm{fh}(w) \sqcup\!\sqcup \mathrm{lh}(w)^R \end{aligned}$$

6.1 Binary decimation

We first consider a kind of binary decimation, which forms a sort of inverse to perfect shuffle.

Given a word $w = a_1 a_2 \cdots a_{2n}$ of even length, note that

$$\mathrm{bd}(w) = a_1 a_3 \cdots a_{2n-1} a_2 a_4 \cdots a_{2n}$$

is formed by "unshuffling" the word into its odd- and even-indexed letters. For example, the French word `maigre` becomes the word `mirage` under this operation.

Theorem 14. *Neither the class of regular languages nor the class of context-free languages is closed under* bd.

Proof. Consider the regular (and context-free) language $L = (00 + 11)^+$. Then $\mathrm{bd}(L) = \{ww \ : \ w \in \{0,1\}^+\}$, which is well-known to be non-context-free. □

6.2 Binary decimation with reverse

We now consider the operation bdr, which is a kind of binary decimation with reverse. Note that

$$\mathrm{bdr}(a_1 a_2 \cdots a_{2n}) = a_1 a_3 \cdots a_{2n-1} a_{2n} \cdots a_4 a_2.$$

For example, bdr(`friend`) = `finder` and bdr(`perverse`) = `preserve`.

Theorem 15. *The class of regular languages is not closed under* bdr.

Proof. Let $L = (00)^+11$. Then $\mathrm{bdr}(L) = \{0^n 110^n \ : \ n \geq 1\}$, which is not regular. □

Theorem 16. *The class of context-free languages is not closed under* bdr.

Proof. Consider $L = \{(03)^n(12)^n \ : \ n \geq 1\}$. Then $\mathrm{bdr}(L) = \{0^n 1^n 2^n 3^n \ : \ n \geq 1\}$, which is not context-free. □

Theorem 17. *If L is regular, then* bdr(L) *is context-free.*

Proof. We show how to accept words of bdr(L) of even length; words of odd length can be treated similarly.

On input $w = b_1 b_2 \cdots b_{2n}$, a PDA can guess $x = a_1 a_2 \cdots a_{2n}$ in parallel with the elements of the input. At each stage the PDA compares a_i to $b_{(i+1)/2}$ if i is odd; and otherwise it pushes a_i onto the stack (if i is even). At some point the PDA nondeterministically guesses that it has seen a_{2n} and pushed it on the stack; it now pops the stack (which is holding $a_{2n} \cdots a_4 a_2$) and compares the stack contents to the rest of the input w.

The PDA accepts if $x \in L$ and the symbols matched as described. □

6.3 Inverse decimation

We now consider a kind of inverse decimation, which shuffles the first and last halves of a word.

Note that if $w = a_1 \cdots a_{2n}$ is of even length, then

$$\mathrm{bdi}(w) = a_1 a_{n+1} a_2 a_{n+2} \cdots a_n a_{2n}.$$

Further, bdi(bd(w)) = bd(bdi(w)) for w of even length.

Theorem 18. *If L is regular then so is* bdi(L).

Proof. On input x we simulate the DFA for L on the odd-indexed letters of x, starting from q_0, and we simulate a second copy of the DFA for L on the even-indexed letters, starting at some guessed state q. Finally, we check to see that our guess of q was correct. □

Theorem 19. *The class of context-free languages is not closed under* bdi.

Proof. Let $L = \{0^m 1^m 2^{2n} 3^{4n} \ : \ m, n \geq 1\}$. It is easy to see that

$$\mathrm{bdi}(L) = \begin{cases} (01)^{m-3n}(02)^{2n}(03)^n(13)^{3n}, & \text{if } m \geq 3n; \\ (02)^{m-n}(03)^n(13)^m(23)^{3n-m}, & \text{if } n \leq m \leq 3n; \\ (03)^m(13)^m(23)^{2n}(33)^{n-m}, & \text{if } m \leq n. \end{cases}$$

Consider $L' := \mathrm{bdi}(L) \cap (03)^+(13)^+(23)^+$. From the above we have $L' = \{(03)^n(13)^n(23)^{2n} \ : \ n \geq 1\}$, which is evidently not context-free. □

6.4 Inverse decimation with reverse

Note that if $w = a_1 \cdots a_{2n}$ is of even length, then $\mathrm{bdir}(w) = a_1 a_{2n} a_2 a_{2n-1} \cdots a_n a_{n+1}$. If $w = a_1 \cdots a_{2n+1}$ is of odd length, we define

$$\mathrm{bdir}(w) = a_1 a_{2n+1} a_2 a_{2n} \cdots a_n a_{n+2} a_{n+1}.$$

Theorem 20. *If L is regular then so is* bdir(L).

Proof. On input x we simulate the DFA M for L on the odd-indexed letters of x, starting from q_0. We also create an NFA M' accepting L^R in the usual manner, by reversing the transitions of M, and making the start state the set of final states of M, and we simulate M' on the even-indexed letters of x. Finally, we check to see that we meet in the middle. □

Theorem 21. *The class of context-free languages is not closed under* bdir.

Proof. Consider $L = \{0^{2m}1^{4m}2^n3^n \ : \ m, n \geq 1\}$. Then L is a CFL, and it is easy to verify that

$$\mathrm{bdir}(0^{2m}1^{4m}2^n3^n) = \begin{cases} (03)^n(02)^n(01)^{2m-2n}(11)^{m+n}, \text{if } m \geq n; \\ (03)^n(02)^{2m-n}(12)^{2n-2m}(11)^{3m-n}, \text{if } m \leq n \leq 2m; \\ (03)^{2m}(13)^{n-2m}(12)^n(11)^{3m-n}, \text{if } 2m \leq n \leq 3m; \\ (03)^{2m}(13)^{n-2m}(12)^{6m-n}(22)^{n-3m}, \text{if } 3m \leq n \leq 6m; \\ (03)^{2m}(13)^{4m}(23)^{n-6m}(22)^{3m}, \text{if } n \geq 6m. \end{cases}$$

Assume $\mathrm{bdir}(L)$ is a CFL. Then $L' := \mathrm{bdir}(L) \cap (03)^+(13)^+(22)^+$ is a CFL, and from above we have $L' = \{(03)^{2m}(13)^{4m}(22)^{3m} \ : \ m \geq 1\}$, which is not a CFL. □

As Georg Zetzsche has kindly pointed out to us, the operation bdir was studied previously by Jantzen and Petersen [3]; they called it "twist". They proved our Theorems 20 and 21.

We now return to the proof of Theorem 13, which was postponed until now. We need two lemmas:

Lemma 22. *Suppose L is a regular language. Then $L' = \{ww^R \ : \ w \in L\}$ is a CFL.*

Proof. On input x, a PDA can guess w and verify it is in L, while pushing it on the stack. Nondeterministically it then guesses it is at the end of w and pops the stack, comparing to the input. □

Lemma 23. *For all words w we have $w \sqcup\!\sqcup w^R = \mathrm{bdir}(w)\,\mathrm{bdir}(w)^R$.*

Proof. If w is of even length then

$$\begin{aligned} w \sqcup\!\sqcup w^R &= (\mathrm{fh}(w)\mathrm{lh}(w)) \sqcup\!\sqcup (\mathrm{fh}(w)\mathrm{lh}(w))^R \\ &= (\mathrm{fh}(w)\mathrm{lh}(w)) \sqcup\!\sqcup (\mathrm{lh}(w)^R\mathrm{fh}(w)^R) \\ &= (\mathrm{fh}(w) \sqcup\!\sqcup \mathrm{lh}(w)^R)(\mathrm{lh}(w) \sqcup\!\sqcup \mathrm{fh}(w)^R) \\ &= \mathrm{bdir}(w)\mathrm{bdir}(w)^R. \end{aligned}$$

A similar proof works for w of odd length. □

We can now prove Theorem 13.

Proof. From Lemma 23 we have

$$\mathrm{pssr}(L) = \bigcup_{x \in L} x \sqcup\!\sqcup x^R = \bigcup_{x \in L} \mathrm{bdir}(x)\,\mathrm{bdir}(x)^R = \bigcup_{x \in \mathrm{bdir}(L)} xx^R.$$

If L is regular, then $\mathrm{bdir}(L)$ is regular, by Theorem 20. Then, from Lemma 22, it follows that $\mathrm{pssr}(L)$ is a CFL. □

7 Acknowledgment

We are grateful to Georg Zetzsche for his remarks.

References

[1] J. Berstel. *Transductions and Context-Free Languages*. Teubner, 1979.

[2] J. Erickson. How hard is unshuffling a string? `http://cstheory.stackexchange.com/questions/34/how-hard-is-unshuffling-a-string`, August 16 2010.

[3] M. Jantzen and H. Petersen. Cancellation in context-free languages: enrichment by reduction. *Theoret. Comput. Sci.* **127** (1994), 149–170.

[4] A. Mansfield. An algorithm for a merge recognition problem. *Disc. Appl. Math.* **4** (1982), 193–197.

[5] A. Mansfield. On the computational complexity of a merge recognition problem. *Disc. Appl. Math.* **5** (1983), 119–122.

[6] M. K. Warmuth and D. Haussler. On the complexity of iterated shuffle. *J. Comput. Sys. Sci.* **28** (1984), 345–358.

Report on BCTCS 2012

The 28th British Colloquium for Theoretical Computer Science

2-5 April 2012, University of Manchester

Ian Pratt-Hartmann

The British Colloquium for Theoretical Computer Science (BCTCS) is an annual forum in which researchers in Theoretical Computer Science can meet, present research findings, and discuss developments in the field. It also provides an environment for PhD students to gain experience in presenting their work in a wider context, and to benefit from contact with established researchers.

BCTCS 2012 was hosted by the University of Manchester, and held from 2nd to 5th April, 2012. The event attracted over 50 participants, and featured an interesting and wide-ranging programme of six invited talks (two from the same speaker) and 33 contributed talks, covering virtually all areas of the subject. This year, BCTCS was collocated with the 19th Workshop for Automated Reasoning (ARW), which attracted over 30 participants; plenary sessions were shared between the two events. Abstracts for all of the talks are provided below.

The conference began with an invited talk by Mike Edmunds, of Cardiff University, entitled "The Antikythera Mechanism and the early history of mechanical computing." Other invited talks were given by Reiner Hähnle, of the Technische Universität, Darmstadt, ("Formal verification of software product families"), Nicole Schweikardt of the Goethe-Universität, Frankfurt am Main ("On the expressive power of logics with invariant uses of arithmetic predicates") and Daniel Kroening, of Oxford University ("SAT over an Abstract Domain"). As in previous years, the London Mathematical Society sponsored a keynote talk in Discrete Mathematics: for this, Rod Downey, of the Victoria University of Wellington, gave two lectures on "Fundamentals of Parametrized Complexity." The financial support of the London Mathematical Society (LMS) is gratefully acknowledged.

BCTCS 2013 will be hosted by the University of Bath from 25th to 28th March, 2013. Researchers and PhD students wishing to contribute talks concerning any aspect of Theoretical Computer Science are cordially invited to do so. Further details are available from the BCTCS website at `www.bctcs.ac.uk`.

Bulletin of the EATCS no 107, pp. 145–165, June 2012

Invited Talks at BCTCS 2012

Mike Edmunds, University of Cardiff
The Antikythera mechanism and the early history of mechanical computing

Perhaps the most extraordinary surviving relic from the ancient Greek world is a device containing over thirty gear wheels dating from the late 2nd century B.C., and now known as the Antikythera Mechanism. This device is an order of magnitude more complicated than any surviving mechanism from the following millennium, and there is no known precursor. It is clear from its structure and inscriptions that its purpose was astronomical, including eclipse prediction. In this illustrated talk, I will outline the results—including an assessment of the accuracy of the device—from our international research team, which has been using the most modern imaging methods to probe the device and its inscriptions. Our results show the extraordinary sophistication of the Mechanism's design. There are fundamental implications for the development of Greek astronomy, philosophy and technology. The subsequent history of mechanical computation will be briefly sketched, emphasising both triumphs and lost opportunities.

Reiner Hähnle, Technische Universität Darmstadt
Formal verification of software product families

Formal verification techniques for software product families not only analyse individual programs, but act on the artifacts and components which are reused to obtain multiple software products. As the number of products is exponential in the number of artifacts, it is essential to perform verification in a modular fashion instead of verifying each product separately: the goal is to reuse not merely software artifacts, but also their verification proofs. In our setting, we realize code reuse by delta-oriented programming, an approach where a core program is gradually transformed by code "deltas" each of which corresponds to a product feature. The delta-oriented paradigm is then extended to contract-based formal specifications and to verification proofs. As a next step towards modular verification we transpose Liskov's behavioural subtyping principle to the delta world. Finally, based on the resulting theory, we perform a syntactic analysis of contract deltas that permits us to automatically factor out those parts of a verification proof that stays valid after applying a code delta.

Nicole Schweikardt, Goethe-Universität
On the expressive power of logics with invariant uses of arithmetic predicates

In this talk I consider first-order formulas (FO, for short) where, apart from the symbols in the given vocabulary, also predicates for linear order and arithmetic may be used. For example, order-invariant formulas are formulas for which the

following is true: if a structure satisfies the formula with one particular linear order of the structure's universe, then it satisfies the formula with any linear order of the structure's universe. Arithmetic-invariant formulas are defined analogously, where, apart from the linear order, other arithmetic predicates may be used in an invariant way. When restricting attention to finite structures, it is known that order-invariant FO is strictly more expressive than plain FO, and arithmetic-invariant FO can express exactly the properties that belong to the circuit complexity class AC^0. On the other hand, by Trakthenbrot's theorem we know that order-invariance (on the class of finite structures) is undecidable. In this talk I want to give an overview of the state-of-the art concerning the expressive power of order-invariant FO and arithmetic-invariant FO.

Daniel Kroening, Oxford University
SAT over an abstract domain

We present a generalisation of the DPLL(T) framework to abstract domains. As an instance, we present a sound and complete analysis for determining the range of floating-point variables in embedded control software. Existing approaches to bounds analysis either use convex abstract domains and are efficient but imprecise, or use floating-point decision procedures, and are precise but do not scale. We present a new analysis that elevates the architecture of a modern SAT solver to operate over floating-point intervals. In experiments, our analyser is consistently more precise than a state-of-the-art static analyser and significantly outperforms floating-point decision procedures.

Rod Downey, Victoria University of Wellington
Fundamentals of parametrized complexity

Parameterized complexity is a multivariant view of complexity seeking to utilize the order present in natural problems to establish practical tractability, or to provide tools that such a methodology won't work. Since the original work in the early 1990's, there has been a clearly defined set of techniques tunes to this idea.

In this pair of tutorial lectures I will discuss the basic methods used for the construction of parameterized algorithms, and some of the methods for showing parameterized intractability and optimality of the computational classes.

This method is something that a modern person doing algorithms should know.

The first lecture will be about the positive techniques, and the second on limitations. The material will be accessible to a beginning graduate student.

Contributed Talks at BCTCS 2012

Martin Adamčík, University of Manchester

Collective reasoning under uncertainty and inconsistency

In practice probabilistic evidence is incomplete and often contradictory. To build an artificial expert system such as one recognizing diseases from list of symptoms, one classical approach is to define an inference process which picks the "most rational" probabilistic belief function which an agent should have, based solely on the given evidence. For a single incomplete but consistent probabilistic knowledge base satisfying certain reasonable topological criteria, the Maximum Entropy (ME) inference process championed by Jaynes and others on the basis of combinatorial arguments, has several different justifications, and was uniquely characterized by an elegant list of axioms developed by Paris and Vencovská. ME enables a single rational agent to choose an optimal probabilistic belief function on the basis of his incomplete but consistent evidence. If however probabilistic evidence is derived from more than one agent, where the evidence from each individual agent is consistent, but the evidence from all agents together is inconsistent, then the question as to how to merge the evidence in such a manner as to be able to choose a single "most rational" probabilistic belief function on the basis of the merged evidence from all agents, has been much less studied from a general theoretical viewpoint.

In this talk we briefly describe a "social" inference process extending ME to the multi-agent context, called the Social Entropy Process (SEP), based on Kullback-Leibler information distance, and first formulated by Wilmers. SEP turns out to be a generalisation of the well-known logarithmic pooling operator for pooling the known probabilistic belief functions of several agents. We show that SEP satisfies a natural variant of the important principle of Irrelevant Information which is known to be satisfied by ME. We also indicate how the merging process described by SEP satisfies a suitable interpretation of the set of merging axioms for knowledge bases formulated by Konieczny and Pino Pérez in.

Chris Banks, University of Edinburgh

Towards a logic of biochemical processes

The Continuous Pi-calculus, cπ, is a continuous time and continuous space process calculus. The prime motivation cπ is for the modelling of the evolution of biochemical processes where the state of process is the real concentration of its constituent species and these concentrations are evolving continuously. Our aim is to provide a logic suitable for expressing properties of such processes, an algorithm for model checking, and tools to support the analysis of these processes.

Our proposed logic is based on Linear Temporal Logic with real constraints.

The deterministic nature of cπ processes means that a linear time logic is sufficient for expressing their temporal properties. However, in the context of biochemical processes, it is desirable to allow the expression of contextual properties. One might like to express how the system changes in different contexts, for example, with the introduction of new species into the system. The proposed logic contains an operator similar to the guarantee from spatial logic which allows the expression of such properties.

We also aim to provide a model checking algorithm to verify assertions in the logic. The problem is how to model check over a continuous state space. One approach to the model checking problem is to take the numerical solutions of ODEs which describe the process; this gives a discrete, deterministic approximation of the process dynamics within a finite time interval. Model checking can then be done using a relatively simple algorithm. Using this technique, software tools for analysis of cπ processes are being developed as part of the project.

Richard Barraclough

A unifying theory of control dependence and its application to arbitrary program structures

There are several similar definitions of control dependence in the literature. These are given in terms of control flow graphs which have had extra restrictions imposed (for example, end-reachability). We define two new generalisations of nontermination insensitive and nontermination sensitive control dependence called *weak* and *strong control-closure*. These are defined for all finite directed graphs, not just control flow graphs, and are hence allow control dependence to be applied to a wider class of program structures than before.

We define an underlying semantics for control dependence by defining two relations between graphs: weak and strong projections. We prove that the graph induced by a set of vertices is a weak/strong projection of the original if and only if the set is weakly/strongly control-closed. Thus, all previous forms of control dependence also satisfy our semantics. Weak and strong projections, thus, precisely capture the essence of control dependence both in our generalisations and all the previous – more restricted – forms. More fundamentally, these semantics can be thought of as correctness criteria for future definitions of control dependence.

Brandon Bennett, University of Leeds

An 'almost analytic' sequent calculus for first-order S5 with constant domains

We present a cut-free sequent calculus for the first-order modal logic S5 with constant domains. The system has the advantage of simplicity, in that all the rules are straightforward and intuitive. The rule set is analytic apart from one rule for eliminating a $\Box$ operator in the succedent of a sequent. Although this rule is not strictly analytic, it does not introduce new non-logical symbols, and

hence provides for an 'almost analytic' proof mechanism. The system is close in form to Gentzen's original sequent calculus for first-order logic. It does not employ any generalisation or augmentation of the basic form of a sequent and only involves very simple side conditions restricting the applicability of two of the rules. Moreover these conditions can be checked locally by looking only at the syntactic form of the immediate conclusion of the rule application.

Adequacy of the system is demonstrated by an inductive cut-elimination proof, which shows equivalence to a well-established Hilbert system formulation of first-order S5. The current proof has the shortcoming that it requires that the antecedents and succedents of a sequent be multi-sets rather than ordinary sets, which seems to be an unnecessary complication. Further work is ongoing to determine whether the possibility of having duplicated formulae in sequents is essential to the completeness of the system or whether such formulae are redundant.

Mihai Burcea, University of Liverpool
Online multi-dimensional dynamic bin packing of unit fraction and power fraction items

We study 2D and 3D dynamic bin packing, in which items arrive and depart at arbitrary times. The 1D problem was first studied by Coffman, Garey, and Johnson motivated by the dynamic storage problem. Bar-Noy et al. have studied packing of unit fraction items (i.e., items with lengths $1/w$ for some integer $w \geq 1$), motivated by the window scheduling problem.

We extend the study of 2D and 3D dynamic bin packing to unit fraction and power fraction items (i.e., items with lengths $1/2^k$ for some integer $k \geq 0$). The objective is to pack the items into unit-sized bins such that the maximum number of bins ever used over all time is minimized. We give a scheme that divides the items into classes and show that applying the first-fit algorithm to each class is 6.7850- and 21.6108-competitive for 2D and 3D, respectively, for unit fraction items. Similarly, we provide a scheme dividing power fraction items into classes for which the first-fit algorithm is 6.2455- and 20.0783-competitive for 2D and 3D, respectively. These are in contrast to the 7.788 and 22.788 competitive ratios for 2D and 3D general sized items.

This is joint work with Prudence W.H. Wong and Fencol C.C. Yung.

Evelyn-Denham Coates, Logic Code Generator Ltd, London, UK
Optimum sort algorithms with o(N) moves

We show implementation of an unstable algorithm that uses $o(N)$ additional memory to do no more than $N\lceil\log_2(N)\rceil - \lceil\log_2(N)\rceil$ comparisons, and no more than $3N$ data moves to sort an array of N values. We show modification to the algorithm so that it does unstable in-place sort with about the same number of comparisons and $O(N)$ data moves. We use our main algorithm to implement a stable merge-sort

with $o(N)$ pointers and $N + C - S$ data moves, where S = number of single cycle permutations in the set to be sorted and C = number of permutation cycles. The operation and operational complexity is as with natural merge-sort except for the additional amount of memory. We show an implementation of the main algorithm that uses $\log_2(N)$ parallel steps with $\frac{1}{2}N(N+1)$ interconnected processors to sort N input values. We present tabulated data from experimental test results on our main algorithm.

The algorithm does no more than $O(N(\log_2(N))^3)$ bit level operations. Without mathematical fanfare or much theoretical exposition, we contend that our algorithm demonstrate an instance solution to the comparing sort problem where a decoupling of comparisons from data moves improves the operational complexity on the number of comparisons and the number of data moves. We pose a theoretical challenge for additional research and development with this approach. We consider our result to be the best so far and possible the final solution to the sort/merge problem from what we have seen in the literature.

Laurence Day, University of Nottingham
The Silence of the Lambdas

At last year's BCTCS, I presented the preliminary results of implementing a modular compiler for a language supporting arithmetic and exceptions which has been constructed as the least fixpoint of functors, where functions over said language are defined as catamorphisms. In this talk, I will recap the necessary ideas before going on to discuss the implementation of the de-Bruijn indexed lambda calculus in this system and the need to switch from catamorphisms to explicit recursion when dealing with term substitution. I will conclude by discussing the potential impact that such a shift may have on notions such as modular proofs.

Michael Gabbay, King's College London
A very simple, explicit construction of some models of beta-equality and beta-eta-equality

We discuss a (relatively) new method of providing models of λ-reduction by which we are able to interpret λ-terms compositionally on "possible world" structures with a ternary accessibility relation. The simplicity of the structures is striking, moreover, they provide us with a surprising richness of interpretations of function abstraction and application.

We show how the models can differentiate between 'extensional' λ-reduction, which supports β-contraction and η-expansion, and 'intensional' reduction which supports only β-contraction. We state semantic characterisation (i.e. completeness) theorems for both. We then show how to extend the method to provide a sound and complete class of models for reduction relations that additionally support β-expansion and η-contraction (i.e. β-equality and η-equality). In this respect

the models we present differ from the familiar models of the λ-calculus as they can distinguish, semantically, between intensional and extensional λ-equality.

For the main result of the paper, we outline an explicit construction of a model of untyped λ-calculus. Again, the simplicity of the construction is striking. Furthermore the construction is sufficiently general that it can be modified to construct models either of $\beta\eta$-equality or simply β-equality.

Finally, we draw some speculative connections between the models constructed and neural nets, abstractly construed. This opens up the possibility that we can view a neural network as computing a λ-term in some sense.

Murdoch Gabbay, Heriot-Watt University
Game semantics using nominal techniques

Game semantics gives denotation to logic and computation using as metaphor a dialogue between *Proponent* and *Opponent*. This can be modelled as a labelled acyclic graph called a *pointer sequence*: nodes are labelled with Proponent / Opponent moves; edges represent the move's *justification*. We propose a model of pointer sequences based on nominal sets, using *atoms* to model edges. Atoms are just a countably infinite set of distinct symbols $a, b, c, \ldots$. Questions and answers are q and a. Pointers are rendered as a pair of atoms. The tip of an arrow is represented as *coabstraction* $[a]$ or $[b]$. Coabstractions bind 'into the future', and are a new idea to nominal techniques. The tail of an arrow is an atom occurence like a or b. *Dangling pointers* are just *free names* (in the sequence above c is free).

Nominal sequences have the following good properties: (1) Closure under subsequences. A subgraph of a pointer sequence is not a pointer sequence, because it might have 'dangling pointers'. (2) Closure under concatenation. Names link up and there are no reindexing isomorphisms. It is less obvious how pointer sequences concatenate. (3) Nominal sequences are an inductive datatype and can be manipulated with standard tools. There is also a specific *nominal* advantage: it enables efficient management of renaming pointers. This is why we use *names* and not e.g. numbers, which are permutatively asymmetric. Taking names and permutations as *primitive* gives good meta-theoretic properties since 'obvious' symmetry properties up to 'reindexing' become obvious. This style of name management and has proven effective in other applications. We shall see that it is also effective here, and we speculate that mechanisation of game semantics using our nominal model will be significantly easier than with pointer sequences.

This is joint work with Dan Ghica

Thomas Gorry, University of Liverpool
Communication-less agent location discovery

We study a randomised distributed communication-less coordination mechanism for uniform anonymous agents located on a circle. The agents perform their ac-

tions in synchronised rounds. At the beginning of each round an agent chooses the direction of its movement from *clockwise* and *anticlockwise*, as well as its speed $0 \leq v \leq 1$ during this round. We assume that the agents are not allowed to overpass, i.e., when an agent collides with another it instantly starts moving with the same speed in the opposite direction. The agents cannot leave marks on the ring, they have zero vision and they cannot exchange messages. However, on the conclusion of each round each agent has access to a detailed trajectory of its movement during this round. This information can be processed and stored by the agent for further analysis.

We assume that n mobile agents are initially located on a circle with circumference one at arbitrary but distinct positions unknown to other agents. The main *location discovery task* to be performed by each agent is to determine the initial position of every other agent and eventually to stop at its initial position, or proceed to another task, in a fully synchronised manner. Our main result is a fully distributed randomised (Las Vegas type) algorithm, solving the *location discovery problem w.h.p* in $O(n \log^2 n)$ rounds. We also show how this mechanism can be adopted to distribute the agents evenly, at equidistant positions, and how to coordinate their joint effort in patrolling the circle. Note that our result also holds if initially the agents do not know the value of n and they have no coherent sense of direction.

Tom Grant, University of Leicester
Maximising lifetime for fault-tolerant target coverage in sensor networks

We present the problem of maximising the lifetime of a sensor network for fault-tolerant target coverage in a setting with composite events. Here, a composite event is the simultaneous occurence of a combination of atomic events, such as the detection of smoke and high temperature. We are given sensor nodes that have an initial battery level and can monitor certain event types, and a set of points at which composite events need to be detected. The point and sensor nodes are located in the Euclidean plane, and all nodes have uniform sensing radius. The goal is to compute a longest activity schedule with the property that at any point in time, each event point is monitored by at least two active sensor nodes.

We present a $(6 + \varepsilon)$-approximation algorithm by devising an approximation algorithm with the same ratio for the dual problem of minimising the weight of a fault-tolerant sensor cover. This generalises previous approximation algorithms for geometric set cover with weighted unit disks and is obtained by enumerating properties of the optimal solution that guide a dynamic programming approach.

Paolo Guagliardo, Free University of Bozen-Bolzano
On the relationship between view updates and logical definability

Given a set of views defined over a database, the *view update problem* consists

in finding suitable ways of propagating an update performed on the views to the underlying database in a consistent and unique way. In this talk, we highlight the strong connection between the view update problem and the notion of definability in logic, and we revisit the abstract functional framework by Bancilhon and Spyratos in a setting where views and updates are exactly given by functions that are expressible in first-order logic. We give a characterisation of views and their inverses based on the notion of definability, and we introduce a general method for checking whether a view update can be uniquely translated as an update of the underlying database.

Christopher Hampson, King's College London
Modal Products with the difference operator

The modal logic **Diff** of the difference operator is known to be Kripke complete with respect to the class of symmetric, pseudo-transitive frames. These frames closely resemble **S5**-relations (i.e. equivalence relations) and it is little surprise that the validity problems for **Diff** and **S5** have the same co-NP complexity, and both logics enjoy the finite model property.

Here we turn our attention to two-dimensional product logics $L_1 \times L_2$, by which we mean the multimodal logic of all product frames where the first component is a frame for L_1 and the second a frame for L_2. It is well-known that product logics of the form $L \times \mathbf{S5}$ are usually decidable whenever L is a decidable (multi)modal logic. We even have that $\mathbf{S5} \times \mathbf{S5}$ enjoys the exponential finite model property. However, it is little understood how product logics of the form $L \times \mathbf{Diff}$ behave.

Here we present some cases where the transition from $L \times \mathbf{S5}$ to $L \times \mathbf{Diff}$ not only increases the complexity of the validity problem, but in fact introduces undecidability. The logics we consider are (i) $\mathbf{K}_u \times \mathbf{Diff}$, which is shown to be undecidable in contrast to $\mathbf{K}_u \times \mathbf{S5}$, which lies in co-N2ExpTime, (ii) $\mathbf{PTL}_{\bigcirc\Box} \times \mathbf{Diff}$, which is shown to be non-r.e. in contrast to the ExpSpace-completeness of $\mathbf{PTL}_{\bigcirc\Box} \times \mathbf{S5}$, and (iii) $\mathbf{Diff} \times \mathbf{Diff}$, which is shown to lack the finite model property, in contrast to $\mathbf{S5} \times \mathbf{S5}$ as mentioned above. Our undecidability and non-r.e.ness results are obtained by reductions of halting-type problems for Minsky machines on two registers, which are known to be Turing-complete.

This is joint work with Agi Kurucz.

Tie Hou, Swansea University
Modeling a language of realizers using domain-theoretic semantics

How to synthesize efficient programs from proofs obeying their formal specifications has been a long sought after goal. One method of program extraction is to employ a realizability interpretation. Kleene first introduced the concept of realizability with the idea of defining a relation between natural numbers and logical sentences. Later many other notions on realizability were introduced, e.g.

the "modified realizability" of Kreisel and the "function realizability" of Kleene-Vesley. The possibility of effectively obtaining a program and its verification proof is based on a sound realizability interpretation.

We study the domain-theoretic semantics of a Church-style typed λ-calculus with constructors, pattern matching and recursion, and show that it is closely related to the semantics of its untyped counterpart. The motivation for this study comes from program extraction from proofs via realizability where one has the choice of extracting typed or untyped terms from proofs. Our result shows that if the extracted type is regular, the choice does not matter.

The proof uses hybrid logical relations. Logical relations have been used successfully to prove properties of typed systems. Famous examples are the strong normalization proofs by Tait and Girard using logical relations called computability predicates or reducibility candidates. The crucial feature of a logical relation is that it is a family of relations indexed by types and defined by induction on types such that all type constructors are interpreted by their logical interpretations.

The reason for studying this domain-theoretic semantics is that it allows for very simple and elegant proofs of computational adequacy, and hence for the correctness of program extraction. Since domain theory combines the computational features of functions with the mathematical definition of function as a mapping from one domain to another, from the perspective point of view, a functional language is basically a shorthand notation for domain-theoretic concepts.

Phillip James, Swansea University
Domain-specific languages and automatic verification

In this talk, we explore the support of automatic verification via careful design of a domain specific language (DSL). For verification, such a specialized language has two effects: (i) Only specific proof goals can be expressed in the language. (ii) The language semantics includes axioms expressing domain knowledge. We illustrate these ideas within the Railway Domain. The semantics of our DSL is a specification in the algebraic specification language CASL. To provide proof support, we use various automated theorem provers which are accessible via the Heterogeneous tool-set (Hets). Finally, we provide concrete verification results illustrating that careful design of a DSL and systematic use of domain knowledge is useful for supporting automatic verification. The result is a step towards a platform for creating domain specific languages with effective automatic verification tools for domain engineers.

Sam Jones, University of Leicester
Groups, formal language theory and decidability

One natural way to describe a group G is by means of a “presentation" consisting of a set X of generators for G and a set R of relations between words over X. If X

is finite then G is finitely generated, and if R is finite then G is finitely presented.

The word problem for a finitely generated group G is an algorithmic question which asks: given two words α and β over some (finite) generating set for G are the elements of G represented by the words α and β the same? An equivalent formulation of this question is: given two words α and β over some generating set for G is the element of G represented by $\alpha\beta^{-1}$ the identity element of G? In this way we can think of the word problem for G as the problem of determining membership of the set of all words which represent the identity element of G.

In this talk I give a brief overview of some of the interactions between group theory and formal language theory, in particular, I will focus on the word problem for groups and the study of the word problem as a formal language. I will explain how groups can be classified in terms of the type of automata which accept their word problem. I will then talk about some decidability questions and results in formal language theory on which I have been working which were motivated by the study of the word problem for groups as a formal language.

Stanislaw Kikot, Birkbeck College, London

The length of query rewriting for OWL 2 QL

Let Σ be a signature consisting of a finite number of constants, unary and binary predicates (A_i and R_i, respectively) and equality. We consider a class of first-order *theories* $\mathcal{T}$ with formulas of the form

- $\forall x(C_1(x) \to C_2(x))$, where $C_1(x)$ and $C_2(x)$ are $A_i(x)$ or $\exists y R_i(x, y)$,
- $\forall x \forall y(R_i(x, y) \to R_j(x, y))$ and $\forall x \forall y(R_i(x, y) \to R_j(y, x))$,

and *conjunctive queries* $\mathbf{q}(\vec{x}) = \exists \vec{y}\, \varphi(x, y)$, where $\varphi(x, y)$ is a conjunction of atoms $A_i(t_1)$ and $R_i(t_1, t_2)$ and t_1 and t_2 are either constants or variables from $\vec{x}, \vec{y}$. The *query rewriting problem* is, given a theory $\mathcal{T}$, a query $\mathbf{q}(\vec{x})$ and a first-order language $\mathcal{L}$, construct a first-order formula $\mathbf{q}^{\mathcal{T}}(\vec{x})$ in the language $\mathcal{L}$ (we call it "the $\mathcal{L}$-rewriting of a query $\mathbf{q}(\vec{x})$ with respect to a theory $\mathcal{T}$") such that for all sets $\mathcal{A}$ of *data* (ground atoms of the form $A(a), R(a, b)$), we have $\mathcal{A} \cup \mathcal{T} \models \mathbf{q}(\vec{a})$ if and only if $\mathcal{A} \models \mathbf{q}^{\mathcal{T}}(\vec{a})$. For example, the rewriting of the query $\mathbf{q}(x) = \exists y R(x, y)$ with respect to the theory $\{\forall x(A(x) \to \exists y R(x, y))\}$ is $\mathbf{q}^{\mathcal{T}}(x) = A(x) \vee \exists y R(x, y)$.

We prove that for "natural" languages $\mathcal{L}$ the minimal size of $\mathbf{q}^{\mathcal{T}}(\vec{x})$ may be exponential in the combined size of $\mathbf{q}(\vec{x})$ and $\mathcal{T}$. On the other hand, we supply an algorithm which gives short rewritings for all reasonable practical cases.

We believe that our research is relevant to the next generation search engine design, where individuals range, say, over web pages, people and products, and examples of unary predicates are "contains the word *dextrose*" and "contains some information relevant to biology".

Andrew Lawrence, Swansea University
Extracting a DPLL Algorithm

In order for verification tools to be used in an industrial context they have to be trusted to a high degree and in some cases need to be certified. We have come up with a new application of program extraction to develop correct certifiable decision procedures. SAT-solvers are one such decision procedure which are common in verification tools. The majority of SAT-solvers used in an industrial context are based on the DPLL proof system. We have performed a correctness proof of the DPLL proof system in the Minlog theorem prover. Using the program extraction facilities of Minlog we have been able to obtain a formally verified SAT-solving algorithm. When run on a CNF formula this algorithm produces a model satisfying the formula or a DPLL derivation showing its unsatisfiability. Computational redundancy was then removed from the algorithm by labelling certain universal quantifiers in the proof as non-computational. The performance of the resulting program was tested with a number of pigeonhole formulae.

David Love, Sheffield Hallam University
Why Don't Cantor's Sets Compute?

In this talk we explore the rejection of the *structural equivalence* used in Hilbet's formal theory (and by extension those of computation). For Hilbert's formal theories, we assume that the structure *of* the theory is congruent with the structures described *by* the theory. Such a congruence is necessary for the self-referential properties of Hilbert's formal theories (which are in turn responsible for the power and scope of those theories).

Even rejecting the assumption of structural equivalence we show that we can build coherent mathematical theories: even if these are not actually formal theories. In this talk we do so using the example of two 'halting like' machines, creating two non-formal analogues of the more well known formal halting machines. By doing so we show that we can explore the space of mathematical theories beyond Hilbert formal theories – without rejecting the vast body of work that has been undertaken on those theories since Hilbert's pronouncement of 1904.

Yavor Nenov, University of Oxford
Computability of topological logics over Euclidean spaces

In the last decades, formalisms for representing and reasoning with spatial knowledge have been of a significant interest to the AI community. Such formalisms are usually referred to as *spatial logics* and consist of a logical language whose variables are interpreted as subsets of a topological space, called *regions*, and whose non-logical symbols have a fixed geometric interpretation. Spatial logics whose non-logical symbols represent topological relations and operations are

called *topological logics*.

We consider quantifier-free languages that feature symbols for Boolean operations and relations (e.g. *union*, *complementation*, etc.) and a predicate symbol for one of two notions of *connectedness* – the property of being topologically connected or the property of having a connected interior. We take the variables of each language to range over different collections of regions in a Euclidean space of dimension higher than one ($\mathbb{R}^n$, $n \geq 2$). We investigate the computability and the computational complexity of the resulting topological logics and show that, despite being based on a very simple logical syntax, they generally exhibit a very high computational complexity, and with few exceptions are all undecidable. The considered logics stand in stark contrast to other studied quantifier-free topological logics, which are all decidable and of relatively low computation complexity.

Jude-Thaddeus Ojiaku, University of Liverpool

Online makespan scheduling of linear deteriorating jobs on parallel machines

Traditional scheduling assumes that the processing time of a job is fixed. Yet there are numerous situations that the processing time increases (deteriorates) as the start time increases. Examples include scheduling cleaning or maintenance, fire fighting, steel production and financial management. Scheduling of deteriorating jobs was first introduced on a single machine by Browne and Yechiali, and Gupta and Gupta independently. In particular, lots of work has been devoted to jobs with linear deterioration. The processing time p_j of job J_j is a linear function of its start time s_j, precisely, $p_j = a_j + b_j s_j$, where a_j is the normal or basic processing time and b_j is the deteriorating rate. The objective is to minimize the makespan of the schedule.

The problem has been mainly studied in the context of offline setting, with optimal offline solutions on single machine and FPTAS on parallel machines. We first consider simple linear deterioration, i.e., $p_j = b_j s_j$. It has been shown that on m parallel machines, when jobs are given one by one and the algorithm has to schedule a job before knowing the next one (online-list model), LS (List Scheduling) is $(1 + b_{\max})^{1-\frac{1}{m}}$-competitive. We extend the study to the online-time model where each job is also associated with a release time. We show that for two machines, no deterministic online algorithm is better than $(1 + b_{\max})$-competitive, implying that the problem is more difficult in the online-time model than in the online-list model. We also show that LS is $(1 + b_{\max})^{2(1-\frac{1}{m})}$-competitive, meaning that it is optimal when $m = 2$. We further consider the case when one of the two machines is unavailable for a fixed time period. For the online-list model, we give an optimal semi-online-list algorithm when $b_{\max}$ is known in advance.

We also study another linear deterioration function, namely, $p_j = a_j + b\, s_j$. In the online-list model, on m machines, we show that RR (Round Robin) is α-competitive and LS is α-competitive in a special case, where α is the ratio of

maximum and minimum normal processing times.

Arnoud Pastink, University of Liverpool
Approximate Nash Equilibria in an uncoupled setup with limited communication

Since it was shown that finding a Nash equilibrium is PPAD-complete, attention has been given to other equilibrium concepts that give approximately a Nash equilibrium in polynomial time. The two most common concepts are (additive) ϵ-approximate Nash equilibria and well-supported Nash equilibria (ϵ-SuppNE); the first requires a strategy of every player such that deviating can give a gain of at most ϵ, and the latter requires that every pure strategy that is played with positive probability is an ϵ-approximate best-response.

Most algorithms for computing ϵ-approximate Nash equilibria assume that all payoffs are known by everybody. There are very few results about approximations in an uncoupled setup where players only know their own payoff matrix. In this uncoupled setup we allow the players to communicate a limited amount of communication. The best ϵ-approximate Nash equilibrium procedure with limited communication in an uncoupled setup at the moment is a simple algortihm which achieves a 0.5–approximate Nash equilibrium by looking at strategies with a support size of 2 and a communication complexity of $O(\log n)$. For ϵ-SuppNE, no non-trivial approximations are known with limited communication.

I will present algorithms that achieve a 0.438–approximate Nash equilibrium and a 0.732-SuppNE, both with a polylogarithmic communication complexity. These results are achieved by cleverly using the properties of the zero-sum game of the player's own payoff matrix.

Robert Piro, University of Oxford
Model-theoretic characterisation of TBoxes and the TBox rewritability Problem

With Knowledge Representation in AI, knowledge is represented in logic, thus giving the representation a clearly defined semantics and making it machine processible. Description Logics (DL), which are essentially decidable fragments of First Order Logics, have been introduced to facilitate this. The decidability allows for automated reasoning and the developing of tool support. The properties and statements a specific DL allows to express must be carefully chosen, as the reasoning complexity rises with the expressiveness of a logic. To cater for the different needs and requirements, a whole zoo of DLs have been introduced and classified w.r.t. their complexity.

An interesting problem therefore is to determine their expressivity. Traditionally, the expressivity of a logic is determined by a characterisation theorem, in which model theoretic properties are determined, such that every first order for-

mula with these properties is expressible as formula of the logic in hand and vice versa. The famous theorem of van Benthem, which characterises the DL $\mathcal{ALC}$ on concept level, is of this kind as well as the work of Kurtonina and de Rijke, who gave characterisations for a whole zoo of description logics.

TBoxes however, which contain sentences describing concept hierarchies and play an important role in ontologies, have not been investigated. Thus, the talk will concentrate on characterisation theorems of TBoxes of the $\mathcal{ALC}$-family as well as $\mathcal{EL}$-TBoxes. Additionally we shall present the rewritability problem for TBoxes, which asks whether a TBox of a certain DL is expressible as TBox of another DL with lower complexity.

Giles Reger, University of Manchester
Quantified event automata: towards expressive and efficient runtime monitors

Runtime verification techniques have recently focused on parametric specifications where events take data values as parameters. These techniques exist on a spectrum inhabited by both efficient and expressive techniques. These characteristics are usually shown to be conflicting - in state-of-the-art solutions, efficiency is obtained at the cost of loss of expressiveness and vice-versa. To seek a solution to this conflict we explore a new point on the spectrum by defining an alternative runtime verification approach. We introduce a new formalism for concisely capturing expressive specifications with parameters. Our technique is more expressive than the currently most efficient techniques while at the same time allowing for optimizations.

In this talk we present event automata and show how they can be extended with quantification to achieve an expressive formalism for monitoring the behaviour of programs at runtime. Using a range of examples, we will demonstrate how these quantified event automata can be interpreted and used within the context of runtime monitoring.

Yanti Rusmawati, University of Manchester
Dynamic networks as concurrent systems and supervised evolution

Highly dynamic and complex computing systems often need to adapt to changing external and internal environments. One approach to this is to build in evolvability as a feature of such systems. Dynamic networks provide examples of such systems, in which a collection of nodes are linked through edges with the number of nodes and edges varying over time. Applications include internet, mobile networks, and unreliable networks. The dynamic behaviours impact on message-passing mechanisms and computations attempting to reach ‘consensus’ or compute global properties. In order to undertake formal reasoning about such systems, abstract models are essential.

We consider abstract descriptions of dynamic networks by developing a formal

framework. We view dynamic networks as concurrent systems, in which there are at least two kinds of processes: a disrupter (which disrupts the connectivity of dynamic networks) and an organizer (which attempts to run the normal execution). Various notions of fairness enable us to reason about message-passing and routing. We also consider how dynamic networks may be modelled via supervised evolution, where components consist of computational systems which are 'monitored' by a supervisory system which may evolve the computation if necessary.

Hugh Steele, University of Manchester
Double glueing and MLL full completeness

Linear Logic (LL) is a deductive system that has garnered considerable attention over the past two decades. Its correspondence to a polymorphic lambda calculus has made it a topic particularly of interest to theoretical computer scientists. The logic's original description takes the form of a sequent calculus, making it ungainly at times; but there has been success expressing the proof theory of LL and of its smaller logical fragments in other ways. Derivations can be described graphically (with propositional atoms and connectives being represented by vertices) or as arrows in an appropriate category.

Naturally it is important for a category describing any logic to be as accurate a model as possible in order for it to be considered useful. The formalisation of this concept is known as 'full completeness': a categorical model of a logic is fully complete if all of its arrows correspond directly to a derivation. In this talk we demonstrate how it is possible to create fully complete models of MLL, the strongly normalising multiplicative fragment of LL, from certain degenerative models. The approach taken makes use of a 'double glueing' construction placed on top of tensor-generated compact closed categories with biproducts; and the new arguments which we employ to show the resulting categories have the properties desired are based around considering the combinatorics behind this construction using standard linear algebra.

Alistair Stewart, University of Edinburgh
Polynomial time algorithms for multi-type branching processes and stochastic context-free grammars

We show that one can approximate the least fixed point solution for a multivariate system of monotone probabilistic polynomial equations in time polynomial in both the encoding size of the system of equations and in $\log(1/\epsilon)$, where $\epsilon > 0$ is the desired additive error bound of the solution.

We use this result to resolve several open problems regarding the computational complexity of computing key quantities associated with some classic and heavily studied stochastic processes, including multi-type branching processes and stochastic context-free grammars.

Martin Sticht, University of Bamberg

A Game-Theoretic Decision Procedure for the constructive Description Logic cALC

In the last years, several languages of Description Logic have been introduced to model knowledge and perform inference on it. There have been several propositions for different application scenarios. The constructive Description Logic *cALC* deals with uncertain or dynamic knowledge.

We make use of a game-theoretic dialogue-based proof technique that has its roots in philosophy and introduce rules so that we can perform reasoning in *cALC* and the modal-logical counterpart CK. The game-theoretic presentation can be considered as an alternative technique to tableau-based proofs, emphasising interaction semantics. As we will see, showing validity is more complex but in return we have a philosophical approach that might make it possible to find out more about related constructive theories and that provides a rich playground of possibilities to extend or alter the underlying semantics.

Dirk Sudholt, University of Sheffield

The analysis of evolutionary algorithms: why evolution is faster with crossover

Evolutionary algorithms use search operators like mutation, crossover and selection to 'evolve' good solutions for optimisation problems. In the past decades there has been a long and controversial debate about when and why the crossover operator is useful. The 'building-block hypothesis' assumes that crossover is particularly helpful if it can recombine good 'building blocks', i. e. short parts of the genome that lead to high fitness. However, attempts at proving this rigorously have been inconclusive; there have been no rigorous and intuitive explanation for the usefulness of crossover. In this talk we provide such an explanation. For functions where 'building blocks' need to be assembled, we prove that a simple evolutionary algorithm with crossover is twice as fast as the fastest evolutionary algorithm using only mutation. The reason is that crossover effectively turns fitness-neutral mutations into improvements by combining the right building blocks at a later stage. This leads to surprising conclusions about the optimal mutation rate.

Christopher D. Thompson-Walsh, University of Cambridge

Extending a Rule-Based Biological Modelling Language Semantics with Containment

Rule-based modelling of biochemical systems has emerged as an important approach in the development of computational techniques for the analysis of these systems. Many biological systems of chemical reactions exhibit a combinatorial explosion in the number of possible species and reactions. Rule-based techniques

rely on using rules which specify *patterns*, rather than explicit species, to succinctly describe these reactions.

One such rule-based modelling language is Kappa, a calculus which defines how a graph, representing a system of linked agents, can be modified by rules that specify which changes may occur at places that match specific local patterns. It has a clean graph-rewriting based semantics; and though the calculus has a wide degree of applicability, it has emerged as a natural description of well-mixed, protein-protein interaction systems and pathways in molecular biology. However, tdoes not presently model the partition of space by membranes, resulting in multiple well-mixed and interacting compartments.

In this talk, we describe work expanding on this graph-based semantics to add containment structure. This containment structure allows us to begin to model the various ways in which biological mixtures are partitioned and enclosed by membranes, which have important effects in real biological systems.

This is joint work with Jonathan Hayman and Glynn Winskel.

Patrick Totzke, University of Edinburgh
Weak bisimulation approximants for BPP processes

In automated verification we want to algorithmically check if a system satisfies a given property. The two main approaches are model checking and equivalence checking. In model checking, the properties are given as formulae of a temporal logic and one checks if a system satisfies these formulae. In equivalence checking, one checks if two given systems are in some semantical sense equal and thereby verify if an implementation is equivalent to a specification that encodes the desired properties. The question arises for which kinds of systems and equivalences this is decidable and if so, what are the complexity bounds.

We consider *weak bisimulation*, an equivalence that has a very intuitive characterization in terms of two-person games and look at systems called *Basic Parallel Processes*, which were introduced as derivations of commutative context-free grammars and are equi-expressible with communication-free Petri nets. The decidability of checking *weak bisimilarity* for *BPP* is a long standing open problem.

In this talk we explore the well known approach of weak bisimulation *approximants* and see how far this takes us for different notions of approximation. We successfully apply the approximation method to restricted classes of BPP processes and establish a few rather surprising lower bounds for the convergence levels of approximants considered before. Lastly, we define new subclass which demands a lot of additional structure. Surprisingly, all "hard" systems that we use to show lower bounds are contained in this very restricted subclass.

Chiara Del Vescovo, University of Manchester

The modular structure of an ontology: atomic decomposition

Ontologies are special logical theories: they are finite sets of axioms in a language that belongs to the family of Description Logics, which are decidable fragments of First Order Logic. They aim to describe knowledge about a domain of interest; and in general they are complex systems, unstructured and large. Decomposing ontologies into *modules* is widely accepted as a fruitful mechanism to ease processing, modifying, analyzing, and reusing parts of an ontology. However, modularisation is a difficult task to achieve for ontologies, because we want to preserve logical properties.

There exist several notions of logically coherent modules for logical theories. Each kind of module determines in the ontology a different *modular structure*, i.e. a set of logically coherent chunks of the ontology and interesting relations between these chunks. However, these suffer from being based on a loose conceptualization of logical connection, and in some notable examples the ontology cannot be decomposed into smaller bits, even if it seems to be well structured.

An important family of modules of ontologies is based on the notion of deductive Conservative Extensions (d-CEs): such modules encapsulate all the ontology's knowledge about a set of terms Σ, called *signature*. We focus on *locality-based modules*, that are computable in polynomial time, whose the most important notions are $\bot$, $\top$, and $\top\bot^*$. Locality-based modules are currently used in many scenarios, e.g. the reuse of part of an ontology.

In our talk, we are going to present Atomic Decomposition (AD), which is the modular structure induced by locality-based modules. We show how ADs are efficiently computed, and describe some of their properties. Finally, we discuss a consequence of ADs on building a model of an ontology.

Domagoj Vrgoc, University of Edinburgh

Regular expressions for data words

In data words, each position carries not only a letter form a finite alphabet, but also a data value coming from an infinite domain. There has been a renewed interest in these due to applications in querying and reasoning about data models with complex structural properties, notably XML, and more recently, graph databases. Logical formalisms designed for querying such data often require concise and easily understandable presentations of regular languages over data words.

Our goal, therefore, is to define and study regular expressions for data words. As the automaton model, we take register automata, which are a natural analog of NFAs for data words. We first equip standard regular expressions with limited memory, and show that they capture the class of data words defined by register automata. The complexity of the main decision problems for these expressions

(nonemptiness, membership) also turns out to be the same as for register automata. We then look at a subclass of these regular expressions that can define many properties of interest in applications of data words, and show that the main decision problems can be solved efficiently for it.

Abstract of PhD Thesis

Author: Alina García-Chacón

Title: The Complexity of Angel-Daemons and Game Isomorphism

Language: English

Supervisor: Joaquim Gabarro

Institute: Universitat Politècnica de Catalunya, UPC (Barcelona Tech)
Departament de Llenguatges i Sistemes Informàtics, LSI

Date: May 7, 2012

Abstract

The analysis of the computational aspects of strategic situations is a basic field in Computer Sciences. Two main topics related to strategic games have been developed. First, introduction and analysis of a class of games (so called angel/daemon games) designed to asses web applications, have been considered. Second, the problem of isomorphism between strategic games has been analysed. Both parts have been separately considered.

Part I: Angel-Daemon Games. A service is a computational method that is made available for general use through a wide area network. The performance of web-services may fluctuate; at times of stress the performance of some services may be degraded (in extreme cases, to the point of failure). In this thesis *uncertainty profiles* and *Angel-Daemon* games are used to analyse service-based behaviours in situations where probabilistic reasoning may not be appropriate.

In such a game, an *angel* player acts on a bounded number of "angelic" services in a beneficial way while a *daemon* player acts on a bounded number of "daemonic" services in a negative way. Examples are used to illustrate how game theory can be used to analyse service-based scenarios in a realistic way that lies between over-optimism and over-pessimism. The resilience of an orchestration to service failure has been analysed - here angels and daemons are used to model services which can fail when placed under stress. The Nash equilibria of a corresponding *Angel-Daemon* game may be used to assign a "robustness" value to an orchestration.

Finally, the complexity of equilibria problems for *Angel-Daemon* games has been analysed. It turns out that *Angel-Daemon* games are, at the best of our knowledge, the first natural example of zero-sum succinct games. Deciding the existence of a pure Nash equilibrium or a dominant strategy for a given player is Σ_2^p-complete. Furthermore, computing the value of an *Angel-Daemon* game

Bulletin of the EATCS no 107, pp. 169–172, June 2012

is EXP-complete. Thus, matching the already known complexity results of the corresponding problems for the generic families of succinctly represented games with exponential number of actions.

Part II: Game Isomorphism. The question of whether two multi-player strategic games are equivalent and the computational complexity of deciding such a property has been addressed. Three notions of isomorphisms, *strong*, *weak* and *local* have been considered. Each one of these isomorphisms preserves a different structure of the game. *Strong* isomorphism is defined to preserve the utility functions and Nash equilibria. *Weak* isomorphism preserves only the player preference relations and thus pure Nash equilibria. *Local* isomorphism preserves preferences defined only on "close" neighbourhood of strategy profiles.

The problem of the computational complexity of game isomorphism, which depends on the level of succinctness of the description of the input games but it is independent of the isomorphism to consider, has been shown. Utilities in games can be given succinctly by Turing machines, boolean circuits or boolean formulas, or explicitly by tables. Actions can be given also explicitly or succinctly. When the games are given in *general form*, an explicit description of actions and a succinct description of utilities have been assumed. It is has been established that the game isomorphism problem for general form games is equivalent to the circuit isomorphism when utilities are described by Turing Machines; and to the boolean formula isomorphism problem when utilities are described by formulas. When the game is given in explicit form, it is has been proven that the game isomorphism problem is equivalent to the graph isomorphism problem.

Finally, an equivalence classes of *small* games and their graphical representation have been also examined.

Table of Contents

Author's address Alina García-Chacón
Universitat Autònoma de Barcelona (UAB)
Institut de Biotecnología i de Biomedicina (IBB)
Grup d'Aplicacions Biomèdiques
de la Ressonància Magnètica Nuclear (GABRMN)
Unitat de Bioinformàtica, Despacho Bioinformàtica 4
CP: 08193 Bellaterra (Cerdanyola del Vallès), Spain
Tel.: 93 581 2807 Fax: +34 93 581 1264
E-mail: agarcia@gabrmn.uab.es
Web: http://gabrmn.uab.es/agarcia

PhD download http://gabrmn.uab.es/agarcia