35th Nordic Workshop on Programming Theory (NWPT)

(Extended abstracts)

Michael Kirkedal Thomsen, Maja Hanne Kirkeby, and Fritz Henglein (eds.)

Copenhagen, November 6 - 8 2024

Preface

The 35th Nordic Workshop on Programming Theory (NWPT) is an annual regional-scope workshops on programming theory, targeted especially at younger researchers. It is idealized to present recent results and/or work-in-progress, and to meet colleagues from the Nordic and Baltic countries.

It has a long tradition and this year will mark it's 35th edition. The workshop attracts PhD students and researchers from the Nordic region, and in recent years, it has expanded to include the Baltic countries as well. It serves as a valuable platform for young PhD students to present their research ideas in a friendly environment and receive input for improvement from field experts. Attendees have the opportunity to establish new contacts within the Nordic region and enhance their communication skills for presenting their research.

The NWPT consists of 7 technical sessions. The sessions welcome 21 regular presentations presenting recent results and work-in-progress in areas programming analysis, verification, language development, semantics of programming languages, programming language design and programming methodology, formal specification of programs, program verification, tools for program verification and construction, program transformation and refinement, real-time, hybrid/cyber-physical systems modeling and verification, models of concurrency and distributed computing, model checking, model-based testing.

The program also features three invited keynote talks by João Saraiva, University of Minho and HASLab / INESC TEC, Portugal titled *Energy Efficiency in Programming Languages*, Sam Staton, University of Oxford, UK titled *Programming theory meets statistical modelling*, and Martin Berger, University of Sussex & Montanarius Ltd, UK titled *Towards GPU-accelerated automated reasoning*.

Program Committee

We also thanks the work of the programming committee, consisting of: Alceste Scalas, Antonis Achilleos, Antti Valmari, Chad Nester, Danny Bøgsted Poulsen, Dylan McDermott, Fabrizio Montesi, Håkon Robbestad Gylterud, Jaakko Järvi, Johannes Borgström, Keijo Heljanko, Magnus Madsen, Magnus Myreen, Marina Waldén, Marjan Sirjani, Martin Elsman, Martin Steffen, Mikhail Barash, Morten Rhiger, Niccolò Veltri, Patrick Bahr, Philipp Ruemmer, Roberto Guanciale, Sandro Stucki, Thomas T. Hildebrandt, Violet Ka I Pun, Volker Stolz, Wojciech Mostowski.

Organisation

We would like to thank the organisers of the workshop:

- Michael Kirkedal Thomsen, University of Copenhagen and University of Oslo
- Maja Hanne Kirkeby, Roskilde University
- Jens Classen, Roskilde University
- Joachim Tilsted Kristensen, University of Oslo
- Matilde Mouritsen Broløs, University of Copenhagen

Also a special thanks to Mikkel Gorm Kæregård Jørgensen and Björg Birkholm Magnúsdóttir from University of Copenhagen for invaluable support in the planning and execution of the workshop.

Support

The organisation behind NWPT thanks the following for support of NWPT:

CARLSBERG FOUNDATION

Carlsberg Foundation, grant CF24-0923, to help cover catering during the workshop and expected for the invited speakers.

PROSA

PROSA, Danish union for IT-professionals, for support of publicity and the closing event.

UNIVERSITY OF COPENHAGEN

University of Copenhagen supports NWPT'24 with location and organisation.

RUC

Roskilde University supports NWPT'24 with organisation.

Contents

Invited Keynotes

João Saraiva; Energy Efficiency in Programming Languages

Affiliation

University of Minho and HASLab / INESC TEC, Portugal

Abstract

In this talk I will compare a large set of programming languages regarding their energy efficiency. We have taken 19 solutions to well defined programming problems, expressed in (up to) 27 programming languages, from well know repositories such as the Computer Language Benchmark Game and Rosetta Code.

In our research, my group built a framework to automatically, and systematically, run, measure and compare the efficiency of such solutions. Ultimately, it is based on such comparison that we propose a serious of efficiency rankings, based on multiple criteria.

Our results show interesting findings, such as, slower/faster languages consuming less/more energy, and how memory usage influences energy consumption. We also show how to use our results to provide software engineers support to decide which language to use when energy efficiency is a concern.

In this talk I will also present our most recent results on leveraging power caps to save energy consumption across programming languages.

Biography

João Saraiva is an Associate Professor at the Department of de Informatics, University of Minho, Braga, Portugal, and a researcher member of HASLab/INESC TEC. He obtained a MSc degree from University do Minho in 1993 and a Ph.D. degree in Computer Science from Utrecht University in 1999. His main research contributions have been in the field of programming language design and implementation, program analysis and transformation, functional programming, and green software. He has experience in participating and coordinating research projects in his research areas, both at national level with projects funded by FCT (projects: PURe, IVY, AMADEUS, CROSS, SSaaPP, AutoSeer, FATBIT, and GreenSwLab) and at international level with projects funded by EPSRC (UK), FLAD/NSF (USA) and by the European Union.

João Saraiva is one of the founders of the successful series of summer schools on Generative and Transformational Techniques in Software Engineering (GTTSE), which he co-organized in 2005, 2007, 2009, 2011, and 2015 (volumes 4143, 5235, 6491, 7680 and 10223 of LNCS - Tutorial by Springer-Verlag) in Braga. He was the organizing chair of ETAPS'07, The European Joint Conferences on Theory and Practice of Software, organized in Braga in 2007, and the worksho co-chair of ICSE'24 held in Lisbon.

Sam Staton; Programming theory meets statistical modelling

Affiliation

University of Oxford, UK

Abstract

I will discuss the idea that concepts from programming theory have a role to play in statistical modelling. Indeed they are already playing this role to some extent, but in different guises. These concepts include abstract types and lazy data structures, as well as more theoretical ideas such as effect gradings, monoidal indeterminates and sheaf categories. So I will present some opportunities for using programming theory to inform and formalize the abstract structure of statistics and probability, including some recent and ongoing results from myself and collaborators. I won't assume much familiarity.

Biography

Sam is a professor of Computer Science in Oxford. He has previously worked in Nijmegen, Paris and Cambridge. The talk will be based on work funded by the ERC grant "BLAST: Better Languages for Statistics" and the ARIA Project "Employing categorical probability towards safe AI".

Martin Berger; Towards GPU-accelerated automated reasoning

Affiliation

University of Sussex & Montanarius Ltd, UK

Abstract

Graphics Processing Units (GPUs) are the work-horses of high-performance computing. The acceleration they provide to applications compatible with their programming paradigm can surpass CPU performance by several orders of magnitude, as notably evidenced by the advancements in deep learning. A significant spectrum of applications, especially within automated reasoning—like SAT/SMT solvers—has yet to reap the benefits of GPU acceleration. In this talk we discuss recent work that successfully implemented program synthesis on GPUs and used it to accelerate learning of logical specifications from examples. We conclude by mapping out a research programme to move more formal verification workloads to GPUs.

Biography

Martin Berger did his PhD in formal models for distributed systems at Imperial College. He's currently an associate professor in the Department of Informatics at the University of Sussex. He's also working as a verification consultant for the microprocessor industry, and is one of the maintainers of the official RISC-V instruction set architecture (`https://github.com/riscv/sail-riscv`). His research interests include: logic and verification, typing systems, process calculus, meta-programming, JIT compiler.

Extended Abstracts

Verification of Data-flow Networks Using the KeY Theorem Prover

Duc Anh Nguyen[1], Philipp Rümmer[1,2], and Wang Yi[1]

[1] Uppsala University, Uppsala, Sweden
[2] University of Regensburg, Germany

Abstract

We present a contract-based method to verify the functional correctness of data-flow networks modeled in MIMOS, a toolchain currently developed in Uppsala. We specify the functional correctness of a network using local contracts annotated on the network components, and global contracts on the inputs and outputs of the network. By utilizing the functional determinism of the model, we can construct automatically a sequential program that is functionally equivalent to the original network. The KeY theorem prover then takes the sequential program with the contracts and checks whether the network conforms to the provided contracts.

1 Introduction

Model-based design paradigms (SDF [6], Lustre [3], Simulink [2]) have gained popularity, especially in embedded and safety-critical systems because of their ability to guarantee functionality, timing correctness, and the absence of deadlock. However, as software is getting more complex, and the industry is moving toward heterogeneous systems, so is the need for a more expressive model that also considers concurrency and composability while remaining functional and timing deterministic.

Our recent work on MIMOS [9] attempts to provide a new software design paradigm based on the Kahn Process Network (KPN) [4], a well-known model of computation for functionally deterministic networks. This work-in-progress paper considers the problem of verifying the functional correctness of the process networks modeled under the MIMOS framework. We opt for a contract-based approach, in which a contract is given to each node, and a global contract is given for the whole network. We show that the network can be automatically transformed into a functionally equivalent sequential program; therefore, contract-based tools and techniques for sequential programs can be used. In our work, each node is implemented in Java; hence, the KeY theorem prover [1] is used to prove the contract.

2 Sequential transformation

In MIMOS, each node is associated with a step function mapping a fixed number of elements from input streams to a fixed number of elements in output streams. In this paper, we only consider acyclic, multiple input, and 1-output networks. Figure 1 shows an example network, figure 2 shows its corresponding stream functions and figure 3 shows stream functions defined recursively using step functions. We use upper case (F1) to denote stream functions and lower case (f1) to denote step functions of each node.

Since data flowing through the network is transformed by step functions along the path, step functions can be composed, resulting in one step function for the whole network, as shown in the last equation in figure 3. This function shows how the network transforms inputs into

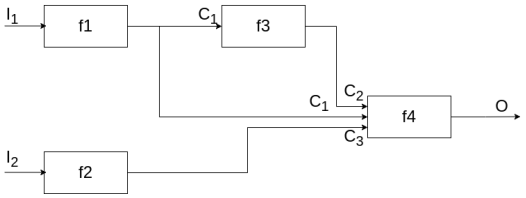


Figure 1: An example KPN network

$$
\begin{aligned}
O &= F4(C_1,\, C_2,\, C_3)\\
C_2 &= F3(C_1)\\
C_1 &= F1(I_1)\\
C_3 &= F2(I_2)
\end{aligned}
$$

Figure 2: System of stream equation of the network in figure 1

$$
\begin{aligned}
C_1 &= F1(i_1 \bullet I_1) = f1(i_1) \bullet F1(I_1)\\
C_3 &= F2(i_2 \bullet I_2) = f2(i_2) \bullet F2(I_2)\\
C_2 &= F3(c_1 \bullet C_1) = f3(c_1) \bullet F3(C_1) = f3(f1(i_1)) \bullet F3(C_1)\\
O &= F4(c_1 \bullet C_1,\, c_2 \bullet C_2,\, c_3 \bullet C_3) = f4(c_1,\, c_2,\, c_3) \bullet F4(C_1,\, C_2,\, C_3)\\
&= f4(f1(i_1),\, f3(f1(i_1)),\, f2(i_2)) \bullet F4(C_1,\, C_2,\, C_3)
\end{aligned}
$$

Figure 3: Recursive definition of stream functions using step functions

outputs. It can be seen from the function definition that if a sequential program invokes the nodes' step functions in the composition order, it will obtain the same output as the network. The composition order can be achieved by sorting the node topologically from inputs to outputs.

Each step function is associated with a contract consisting of pre- and post-conditions. If input values satisfy the pre-conditions, output values have to satisfy the post-conditions. The system-level step function, composed of nodes' step functions, is associated with a global contract, also in the form of pre- and post-conditions. The pre-conditions express assumptions about the values of the network inputs, and the post-conditions specify the expected values of the network outputs.

3 Implementation detail

In the MIMOS tool, each node's step function is given as a Java method. Therefore, we decided to use the Java Modeling Language (JML) [5] to specify contracts for each node. A top-level Java method is automatically generated, which invokes all step functions in the correct order. This method represents one iteration of the whole network. The global contract of the network is specified as a JML contract for this top-level method. All contracts are provided manually by the user. Finally, we feed all methods and specifications to the KeY tool, which then attempts to verify all contracts automatically. In our experiments with smaller case studies in MIMOS, verification was usually completed within a couple of seconds.

4 Conclusion and Future Works

In this paper, we applied a contract-based approach to verify the functional contracts of MIMOS networks. The contracts comprise pre- and post-conditions and are verified automatically using

the KeY theorem prover.

In related work, verification of another extension of KPNs, Dataflow Process Networks (DPNs), has been studied in [8], in which a contract-based approach is presented. Networks, together with contracts and scheduling information, are encoded in Boogie language and verified using the SPIN model checker. In [7], Lin et al. verifies networks of reactors by converting the network, implemented in a subset of C, to an SMT model and then use Z3 SMT solver to solve it.

As an avenue of future work, we will consider networks with multiple outputs and networks with loops. Since MIMOS models are inherently asynchronous, contracts over multiple output streams need to utilize timing information to check which outputs' values are observed at the same time. To verify networks with loops, loop invariants are needed to help prove the entire network.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[2] Simulink Documentation. Simulation and model-based design, 2020.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[4] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, 1974.

[5] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[6] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[7] Shaokai Lin, Yatin A. Manerkar, Marten Lohstroh, Elizabeth Polgreen, Sheng-Jung Yu, Chadlia Jerad, Edward A. Lee, and Sanjit A. Seshia. Towards building verifiable cps using lingua franca. *ACM Trans. Embed. Comput. Syst.*, 22(5s), September 2023.

[8] Jonatan Wiik, Johan Ersfolk, and Marina Waldén. A contract-based approach to scheduling and verification of dynamic dataflow networks. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–10, 2018.

[9] Wang Yi, Morteza Mohaqeqi, and Susanne Graf. Mimos: A deterministic model for the design and update of real-time systems. In Maurice H. ter Beek and Marjan Sirjani, editors, *Coordination Models and Languages*, pages 17–34, Cham, 2022. Springer Nature Switzerland.

Finding Inductive Invariants Fastly - A Support Technique for Deductive Software Verification

Thomas Baar[1] and Volker Stolz[2]

[1] Hochschule für Technik und Wirtschaft Berlin, Germany
thomas.baar@htw-berlin.de
[2] Western Norway University of Applied Sciences
volker.stolz@hvl.no

1 Motivation

We consider the case of formally verifying the implementation of a function being correct with respect to an annotated contract consisting of pre-/post-condition. When using tools like KeY [1] or Verifast [4], the most challenging task for the user is to provide the right code annotations allowing the verification tool to verify the correctness of the function's implementation automatically. Probably, the most widely known annotation to be provided by the user are loop invariants, which can be challenging to find [2].

We have recently experienced this pain when implementing the Hotel-Room-Locking protocol (a case study for Alloy [3]) in C and verifying this C-code using Verifast. The protocol prescribes actions such as *checkin*, *checkout*, *enterRoom*, etc., and how they change the current system state (e.g., validity of keys, occupation status of room). The correctness property to be shown is that whenever a guest is able to enter a room with his key, then this room must be occupied by the very same guest. Unfortunately, the correctness property cannot be proven straightforwardly, since it is not an inductive property.

2 Approach

In order to illustrate the burden of the verification engineer to provide the right annotations, we define a very simple but nethertheless representable example system: Our system consists of two integer variables `x` and `y` and two actions `changeX()` and `changeY()`. Initially, both variables are set to `0`. The action `changeX()` is implemented as follows:

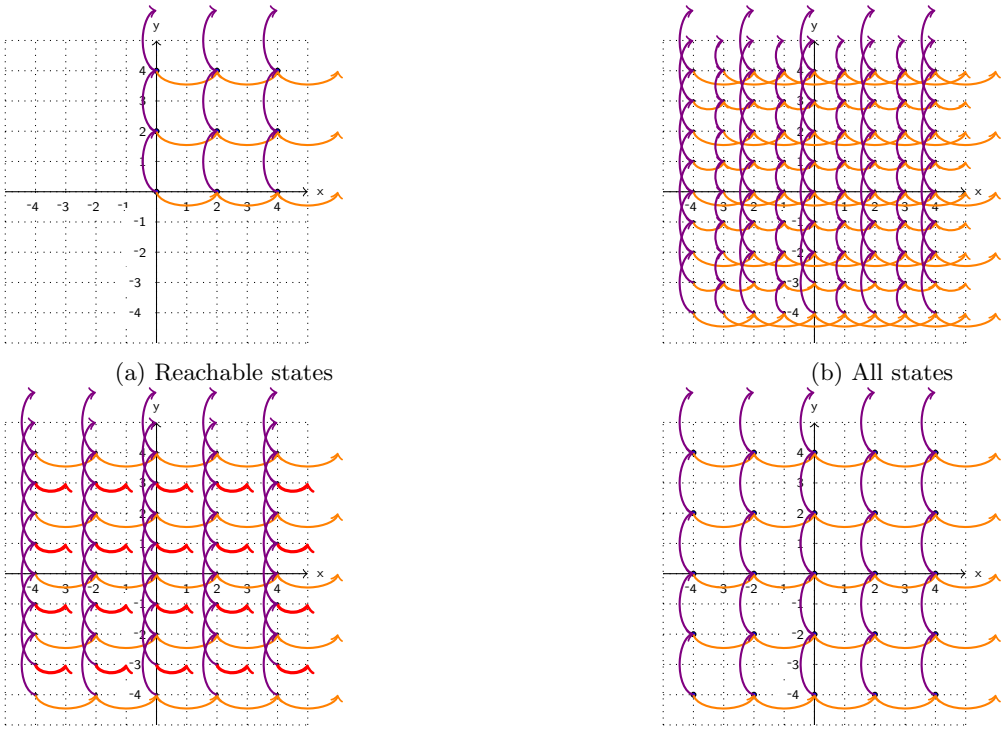
```
changeX() { if (y % 2 == 0){x = x + 2;} else { x = x + 1; } }
```

Whenever `changeX()` is invoked, the value of variable `x` is increased. The difference of the increase has value `2` when the value of (the other) variable `y` is even, and value `1` otherwise.

The other action `changeY()` is implemented fully analogously: the value of `y` is increased by `1` or `2`, depending on whether `x` is even or not.

Note that variable `x`/`y` changes its value only in `changeX()`/`changeY()`, respectively. We further observe that the value of `x` remains even, as long a value of `y` is even. If `changeX()` is invoked in a situation in which `y` is odd, then the value of `x` changes from even to odd and vice versa.

Fig. 1a shows the system behaviour and marks the reachable states when the system starts in (0,0). The figure illustrates that each action when invoked in a reachable state increases the variable `x`/`y` always by `2`. In contrast, Fig. 1b shows the system behaviour for all possible states, not just the reachable. Here, also smaller steps are possible, e.g. when `changeX()` is started in state (2,1).



(a) Reachable states

(b) All states

(c) Property

(d) Invariant

Figure 1: Analysis of the system

It is easy to see that all reachable states satisfy the property P: `x % 2 == 0` (`x` is even). However, this property P alone is not provable, because P is not an *inductive* property and not obeyed by implementation `changeX()`. In Fig. 1c we see for example that `changeX()` started in state (2,3) results in state (3,3) which does not satisfy P.

To overcome this problem, an invariant INV, which is stronger as the original property P, is verified. In our case, the invariant is defined as INV: `(x % 2 == 0 && y % 2 == 0)`. As Fig. 1d suggests, this INV is inductive indeed, so that INV can easily be proven by Verifast.

To sum up: Each system has a set of reachable states S_{reach} and this set is by definition inductive: When an action is invoked in any state from this set, it results in a state from this set too (otherwise, this post state would not be reachable).

Moreover, we have a property P to be verified and we assume that this property holds in each reachable state. Thus, the set of states satisfying P is a superset of reachable states ($S_{reach} \subseteq S_P$) but this set might not be inductive: There might be a state $s \in S_P$, so that an action started in s would yield to a state outside of S_P. To address this issue, the verification engineer has to propose an invariant INV, which is

- *correct* ($S_{reach} \subseteq S_{INV}$), and
- *stronger* as P ($S_{INV} \subseteq S_P$), and
- *inductive* (S_{INV} cannot be left by any action)

3 User Support to Evaluate Invariant Candidates

The verification engineer could be effectively supported in providing the right invariant when a tool with the following features would be available:

- generate a subset of all possible system states (no matter whether reachable or not) of managable size called U
- mark within U the subset of reachable states
- for a given property P
  - mark all states within U satisfying P
  - analyze whether the set for P is inductive or not

Such a tool could give early feedback, whether an invariant proposed by the user is indeed correct and inductive for the (subset of all possible) states in U. The tool might also assist the user with a visualization similar to the diagrams we provided above for the tiny system `changeX()`/`changeY()`. However, visualization is usually not that straightforward as in this example with a two-dimensional state space of integers, which can be nicely drawn is a two-dimensional grid. However, even if the state space has a much higher dimension, visualization might still be helpful and manageable when the user considers a projection to the low-dimensional state space.

References

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors. *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*, volume 12345 of *Lecture Notes in Computer Science*. Springer, 2020.

[2] Dirk Beyer and Martin Spiessl. LIV: loop-invariant validation using straight-line programs. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 2074–2077. IEEE, 2023.

[3] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. Publisher, 2012. MIT Press.

[4] Bart Jacobs, Jan Smans, and Frank Piessens. *The VeriFast Program Verifier: A Tutorial*. Available from https://github.com/verifast/tutorial.

Compositional Design and Verification of Swarm Protocols

Florian Furbach[1], Alceste Scalas[1], Roland Kuhn[2], Emilio Tuosto[3], and Hernán Melgratti[4]

[1] Technical University of Denmark, Denmark
[2] Actyx AG, Germany
[3] Gran Sasso Science Institute, Italy
[4] University of Buenos Aires, Argentina

Abstract

Swarm protocols are a recently introduced formalism for specifying and verifying the intended behaviour of a distributed ensemble of agents, used e.g. for factory automation. Unfortunately, existing design and verification techniques for swarm protocols are not compositional. We explain the problem and present our ongoing work that addresses it.

1 Background: Swarm Protocols

Swarm protocols [7] formalise the intended behaviour of distributed interacting agents, referred to as *machines*. A swarm operates under a *local-first paradigm* [6, 5], enabling each machine to make progress independently without requiring up-to-date global information or an active connection. This decentralized design enhances the availability and efficiency of swarm systems. Swarm protocols are implemented in the Open Source Actyx toolkit [1] for factory automation.

Machines in a swarm communicate by emitting *events* which propagate asynchronously throughout the swarm. Each machine M *subscribes* only to certain types of events and ignores others. During execution, M maintains a local log l_M of received events, updating its state when new events alter the log. These events have a globally agreed total order [2], often defined by timestamps. Each machine in a swarm plays a specific *role*. Akin to *multiparty session types* [3, 4], the global behaviour of a swarm is specified as a swarm protocol G, which describes the interactions between different roles.

Consider e.g. the *Warehouse* protocol in Figure 1. The transition "**request**@**T**⟨partID⟩" means: a machine playing the transport role (**T**) can perform a **request** operation, emitting an event of type partID; then, a forklift (**FL**) **get**s the part at a certain position, and a **T**ransport **deliver**s the part. In between requests, a door (**D**) may **close** emitting the closing time.

Each machine implements a role-specific *projection* of G that subscribes to some events. Figure 2 shows a projection of the *Warehouse* protocol on role **D**: here, **D** subscribes to the

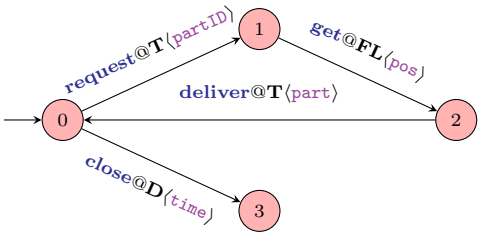


Figure 1: The swarm protocol *Warehouse*.

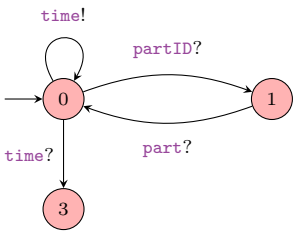


Figure 2: Projection of *Warehouse* on role **D**.

events of type `partID` and `part` (emitted by **T**), but does not subscribe to `pos` (emitted by **FL**). The projection may also emit the `time` event (denoted "`time`!") and read it back from the log (`time`?). A possible run of a swarm implementing the *Warehouse* protocol is:

1. A machine with the **T**ransport role performs a **request** and emits the event *PartID1*.
2. This event propagates to the local log of the **F**ork**L**ift machines.
3. Thus, a **F**ork**L**ift machine transitions to state 1, where it **get**s the part and emits *PosX*.
4. Concurrently, a second forklift also **get**s the part and emits its position *PosY*.
5. The **T**ransport machine receives the event *PosX* and **deliver**s *Part1*.
6. A **D**oor receives these events in its log, $l_D = PartID1.PosX.PosY.Part1$, and then **close**s, emitting the event *8PM* (of type `time`).
7. A **T**ransport, unaware of the event *8PM*, **request**s another part by emitting *PartID2*.

Eventually, all events propagate throughout the swarm, and all machines reach the same log, e.g., $l = PartID1.PosX.PosY.Part1.8PM.PartID2$. Observe that this example log l shows a case where events *PosY* and *PartID2* are emitted too late and thus invalidated: According to the *Warehouse* protocol in Figure 1, *PosY* conflicts with the event *PosX* that comes earlier in the log. Likewise, *PartID2* is invalid due to the earlier *8PM*. It's important to note that this swarm model assumes actions are reversible. However, in practical applications, certain operations may be irreversible, which the implementation must account for.

2 Compositional Design of Swarm Protocols

The current theory [7] and implementation [1] of swarm protocols only supports monolithic swarm design: i.e., it does not support developing a large and complex factory automation protocol by combining modular, reusable swarm components that are developed and verified independently. To address this issue, we introduce a theory of swarm composition, based on *interfacing roles*: i.e., given two swarm protocols G_1 and G_2 with suitable interfacing roles, we define the composition operation $G_1 \,||\, G_2$; also, given two swarms (i.e., ensembles of machines) S_1 and S_2 implemented by projecting G_1 and G_2, we define the swarm composition $S_1 \,||\, S_2$.

E.g., consider Figure 3: it shows a *Factory* swarm protocol where a **T**ransport **request**s and **deliver**s a part, which is then utilized by a **R**obot to **build** a car (emitting an event of type `car`). The **T**ransport role can act as an interface with the *Warehouse* protocol in Figure 1, allowing a synchronising composed behaviour: intuitively, the composed swarm protocol *Warehouse* || *Factory* can either advance within one sub-protocol, or execute a **T**ransport interface operation that spans both protocols. Figure 4 shows the swarm protocol arising from such a composition. Note that the operations **build** and **close** are concurrent.

3 Compositional Verification of Swarm Correctness

A key correctness property for a swarm of machines S projected from a swarm protocol G is *eventual fidelity*: once every event has been propagated to every machine in S, all machines must reach a consensus on which events are valid. This means they must conform to the

Figure 3: A *Factory* protocol that can interface with the *Warehouse* using the **T**ransport role.

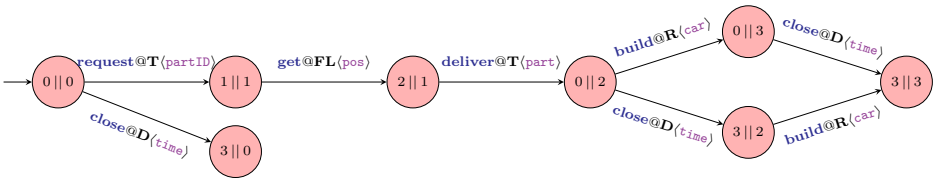


Figure 4: The swarm protocol arising from the composition $\mathit{Warehouse} \,||\, \mathit{Factory}$.

same execution of the swarm protocol G. Assuming every role except **D** subscribes to every type, the example is eventually faithful: All machines conform to an execution with the events $\mathit{PartID1.PosX.Part1.8PM}$, even though **D** does not subscribe to PosX. Existing work [7] ensures eventual fidelity by verifying a restrictive syntactic property of G called *well-formedness*. Unfortunately, their notion of well-formedness is not compositional: i.e., given two well-formed swarm protocols G_1 and G_2, their composition $G_1 \,||\, G_2$ may not be well-formed and must be verified "from scratch." Therefore, we do not use it in our approach. Moreover, the machines that implement G_1 may need to be manually adjusted by subscribing to many events from G_2 (and *vice versa*) to behave correctly in the composed swarm. These limitations make verification inefficient, and hamper the reusability of machine implementations. To solve these issues, we adopt a two-pronged approach, by introducing:

1. A new, *compositional* well-formedness property for a swarm protocol G. Technically, the new property is defined w.r.t. a set of subscriptions, and identifies a (typically small) set of *branching event types* that a role has to subscribe to when projected from G. An event type is branching, if there is a choice between it and other outgoing events. In Figure 4, `partID` and `time` are branching. Type `car` is not branching since there is no choice between `car` and `time` in state $0 \,||\, 2$, only their order changes.

2. A new *branch tracking* functionality in the semantics of each machine. The functionality works as follows: every emitted event contains a pointer to the last branching event that preceded it; a machine only accepts an event if it points to the expected branching event.

These two techniques require machines to only subscribe to a few event types, thus keeping the machines simple and the swarm efficient. Together, the compositional well-formedness and branch tracking guarantee that a swarm behaves correctly, meaning all machines eventually agree on which events are valid. This is outlined in Theorem 1 below.

Theorem 1 (Outline)**.** *Let G be a compositional well-formed swarm protocol. Let S be a swarm of branch-tracking machines that implements G. Then, S enjoys eventual fidelity to G.*

Moreover, our novel notions of swarm composition, compositional well-formedness, and branch tracking enable compositional verification. We only have to verify that the initial swarm protocols are well-formed. Any compositions are guaranteed to remain well-formed. By verifying that the initial swarm protocols are well-formed, we ensure the eventual fidelity of all swarm compositions, as outlined in Theorem 2 below.

Theorem 2 (Outline)**.** *Let G_1 and G_2 be compositional well-formed swarm protocols. Let S_1 and S_2 be swarms of branch-tracking machines that implement G_1 and G_2, respectively. Then:*

1. *The swarm protocol composition $G_1 \,||\, G_2$ is compositional well-formed, and*
2. *The swarm composition $\mathcal{A}(S_1, G_2) \,||\, \mathcal{A}(S_2, G_1)$ enjoys eventual fidelity to $G_1 \,||\, G_2$.*

In the outline of Theorem 2 above, $\mathcal{A}(S_1, G_2)$ is an *adaptation* that subscribes some machines in S_1 to some branching events (identified by our compositional well-formedness property) in G_2. This is necessary because the machines in S_1 may need to receive such events to avoid desynchronising from S_2. A symmetric adaptation is applied to S_2 using G_1. For instance, a **R**obot needs to subscribe to `time` to determine whether the doors **close**d before a **request**, which would invalidate it. This automatic adaptation allows for reusing machine implementations, and minimises communication across composed swarms.

4 Ongoing and Future Work

We are currently finalising the proofs of Theorem 1 and Theorem 2 outlined above. Moreover, we are implementing our new definitions of swarm protocol well-formedness and machine adaptation by extending the Actyx toolkit [1]. We plan to apply our compositional swarm design and verification results, and their implementation, to industrial factory automation scenarios.

5 Acknowledgments

This research was partly supported by the Horizon Europe grant 101093006 (TaRDIS).

References

[1] Actyx AG. Actyx developer website. https://developer.actyx.com, 2024. [Online; accessed on 29 October 2024].

[2] Sebastian Burckhardt. Principles of eventual consistency. *Found. Trends Program. Lang.*, 1(1–2):1–150, oct 2014.

[3] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT Symposium on principles of programming languages*, pages 273–284, 2008.

[4] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM (JACM)*, 63(1):1–67, 2016.

[5] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: You own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, page 154–178, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Roland Kuhn. Local-first cooperation: Autonomy at the edge, secured by crypto, 100% available. https://www.infoq.com/articles/local-first-cooperation/, 2021. [Online; accessed on 29 October 2024].

[7] Roland Kuhn, Hernán C. Melgratti, and Emilio Tuosto. Behavioural types for local-first software. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States*, volume 263 of *LIPIcs*, pages 15:1–15:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

Modeling systems via register machines for the verification of weak memory models

Elli Anastasiadi[1,2] and Samuel Grahn[1]

[1] Department of Information Technology, Uppsala University, Sweden
[2] Department of Computer Science, Aalborg University, Denmark
elli.anastasiadi@it.uu.se, anastasiadi.elli0@gmail.com, samuel.grahn@it.uu.se

Abstract

Register machines can be verified against weak memory models that might be undecidable in the general case. Here we demonstrate how to model system features via register machines so that we enable their verification.

Hardware optimization has been an active area of research for many decades. However, there is an inherent tradeoff between performance and simplicity (or predictability) of a system. To optimize memory accesses, *weak memory models* have become the norm for multicore processors. In order to avoid the long wait for data from memory, each core has its own local cache. This cache is then, from a programmers' point of view, nondeterministically saved into memory. This means that when a thread reads from a memory location, it does not necessarily read the latest value written to it by another thread, as the effect of that cache may not yet have propagated. There are several versions of weak memory models that provide different guarantees to the programmer. One of the most fundamental computational problems for a given consistency model `CM` is *consistency checking*. Consistency checking comes in two flavors: *testing* and *verification* [1, 2, 6, 7]. In this abstract we focus on the verification problem, in which we are given an *implementation* and asked to check whether *all* executions of the implementation satisfy `CM`. In an implementation, the number of allowed runs is usually infinite, and the runs themselves can be infinite. Moreover, there is no priori bound as to which finite restriction of them one has to consider in order to guarantee that all runs are safe. This has led to a plethora of hardness and undecidability results [2, 3], which indicate that in order to tackle the verification problem one needs to restrict the expressiveness of the formalism describing an implementation.

In our case we have picked to work with the classical example of a *register machine* model to describe the underlying implementation. The model is an extended finite-state machine with a finite set of registers that store data values from an unbounded domain. The machine interacts with a finite set of external threads through write (where the register machine inputs a value to a register) operations and read (outputting a stored value) operations performed on a finite set of variables. Furthermore, the machine can perform internal transitions to transfer data between registers. The model is conceptually simple, providing a concise framework to state our complexity results. At the same time, it is sufficiently robust to model relevant features needed to model cache protocols or distributed systems, such as rendezvous communication, broadcasting fences, and store buffers [3, 4]. Moreover, recent work use automata-like formalisms for learning models of implementations and detecting bugs [5]. Such work enhance the relevance of register machines for verifying program behaviors. In our approach, a register machine is an intermediate representation of an actual hardware architecture or a protocol for handling memory access, which we can verify against several weak memory semantics. The register machine, therefore, captures precisely what kind of memory accesses are offered to a given program and what values the architecture would return for those accesses. In this abstract we focus on demonstrating how to capture common system and programming language characteristics in

the setting of register machines. Specifically, we demonstrate how we model buffers and fences. However, in a similar way we can capture vector clocks.

1 Register machines

Assume a set Θ of threads, a set $\mathcal{V}$ of variables, and a set `Regs` of registers. We assume that the variables and the registers range over a (potentially infinite) set $\mathcal{D}$ of data values with the particular value $0 \in \mathcal{D}$. A *register machine* (or simply a *machine*) $\mathcal{M}$ is a tuple $\langle Q, q_{\texttt{init}}, \Delta \rangle$ where Q is the finite set of states, $q_{\texttt{init}} \in Q$ is the initial state, and Δ is the finite set of transitions. A transition is a triple of the form $\langle q, \mathtt{o}, q' \rangle$ where $q, q' \in Q$ are states, and $\mathtt{o}$ is an operation. The operation $\mathtt{o}$ can be in one of the following three forms:

- $(\mathsf{W}, \theta, x, a)$ receives the value of the variable x from θ and writes (stores) the value in register a. The environment selects the written value (the program running on $\mathcal{M}$).
- $(\mathsf{R}, \theta, x, a)$ reads of value of the variable x from the register a and delivers the stored value to θ.
- $a := a'$ copies the value stored in the register a' to the register a.

2 Modeling buffers and fences

In modern computers, written data is not immediately visible to other threads, but rather are placed in cache. This cache is then eventually – nondeterministically from the view of the programmer – copied back into memory.

Buffers: A buffer consists of a sequence of messages. The contents of each message, as well as the number of buffers, depend on the particulars of the modeled system. For example, in some setups buffers store the messages that are pending to be read by a process, while in others the information concerns the writes that have taken place in some part of the architecture.

To demonstrate how to model such a buffer we focus on the second case, and specifically in the x86 architecture. There each thread (or process) has its own *store buffer*, whose messages are information about writes. Whenever a thread writes a value, a message is appended to the buffer containing the neccessary information about this write. Whenever a thread tries to read from memory, it first looks in its write buffer and returns the value of the latest write to that address. If no such write message exists in the buffer, it gets the value from memory. The effect of the first message in a buffer can be applied to memory nondeterministically.

One important detail about these buffers is that they are bounded in practice, e.g. the cache of a computer is finite. This means we can include information about buffer contents in each state, while still keeping a finite (albeit exponentially larger) state space. Once the contents of the buffer are part of the state, one can add the transitions that match the modeled system.

As a simple example, we will construct a system with two threads, two variables and two store buffers, each of size 2. For each variable x there is a register x_{mem} representing the value in memory. A state in this machine is a set $\{B^\theta, B^\phi\}$, consisting only of the buffers, each of which is a queue of variable names. Each buffer B^i has two registers B^i_1, B^i_2, one for each message slot (due to the size bound of 2). For each variable, each state has read transitions to itself that reads from the appropriate register. For instance, if the buffer B^θ contains a message x at index i, there is a self-loop transition with label $(\mathsf{R}, \theta, x, B^\theta_i)$. Similarly, if there is no such message, there is a transition $(\mathsf{R}, \theta, x, x_{mem})$. Assuming the store buffer of thread θ has

an empty message slot with index i, we allow θ to perform a write. This write is a transition $\{B^\theta, B^\phi\}$ $(\mathsf{W}, \theta, x, B_i^\theta)$ $\{B^{\theta\prime}, B^\phi\}$, where $B^{\theta\prime}$ is the result of appending x to B^θ. The final type of transition is the handling of a message. If there is a message x in the front of a buffer B^θ (i.e. at index 1), we add a sequence of transitions, starting from the initial state $\{B_\theta, B_\phi\}$, to an auxilliary state q, into a final state $\{B^{\theta\prime}, B^\phi\}$, where $B^{\theta\prime}$ is B^θ but with the first message removed and everything else shifted forwards one step. The first transition writes the data in the first write message to memory, through a copy $x_{mem} := B_1^\theta$. The next transition moves the contents of the second message to the registers of the first; through $B_1^\theta := B_2^\theta$. In the case of a longer buffer, there would be a longer chain of such shifts.

Fences: A fence is a construct that synchronizes threads by ensuring any message buffers are empty. This can be modeled from a state q, by only allowing message handling transitions from q, unless each buffer in q is empty. In such a case, we have a dummy transition $a := a$ into a new state q', with empty buffers, corresponding to having passed through the fence.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA):135:1–135:29, 2018.

[2] Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.

[3] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. On verifying causal consistency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 626–638. ACM, 2017.

[4] Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods Syst. Des.*, 23(3):257–301, 2003.

[5] Simon Dierl, Paul Fiterau-Brostean, Falk Howar, Bengt Jonsson, Konstantinos Sagonas, and Fredrik Tåquist. Scalable tree-based register automata learning. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 87–108, Cham, 2024. Springer Nature Switzerland.

[6] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997.

[7] Weiyu Luo and Brian Demsky. C11tester: a race detector for C/C++ atomics. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 630–646. ACM, 2021.

GPU Consistency Analysis with Dartagnan

Haining Tong[1] and Keijo Heljanko[1,2]

[1] University of Helsinki, Helsinki, Finland
haining.tong@helsinki.fi
keijo.heljanko@helsinki.fi
[2] Helsinki Institute for Information Technology, Helsinki, Finland.

1 Introduction

In recent years, significant efforts have been made to formalize GPU consistency models [3, 12, 11]. These models specify how concurrent programs behave on GPU hardware and thus help users write programs that are free of concurrency bugs. Some of the efforts have introduced prototype tools to assist researchers and developers in analyzing the correctness of GPU programs with respect to these models. However, these tools are typically tightly coupled with a specific model, making it difficult to compare consistency features between different models. Additionally, since these tools are typically built on top of the Alloy framework [10], they are limited to straight-line code in pseudo-assembly without control flow instructions and struggle with scalability, particularly for programs with more than a few instructions.

In this abstract, we primarily discuss our prior work on integrating support for the Nvidia Ptx [11, 12] and Khronos Vulkan [3] consistency models into Dartagnan[1], an open-source Bounded Model Checking (BMC) tool designed to check state reachability under specific memory models. The technical details of the approach will be published in an accepted manuscript to ASPLOS2024 [13], and here we mainly outline the overall approach taken in that paper. The process involves formalizing GPU consistency models in the domain-specific language `.cat` [4, 7, 5], extending it with GPU-specific features, enhancing our Dartagnan to support the GPU consistency models, and implementing new front-ends for both pseudo-assembly syntax and a subset of real Spir-V assembly [2]. Additionally, we present a bug discovered in the original Vulkan consistency model.

2 Consistency Models and Bounded Model Checking

Memory consistency models can be formalised in `.cat` language [4, 7, 5], the core part of which is laid out in Figure 1. A consistency model in `.cat` is defined by tagged memory events (s), relations over memory events (r), and axioms over the relations (axm). Base event tags (t) are derived from the type of program instructions e.g., writes $\mathbb{W}$, reads $\mathbb{R}$, fences $\mathbb{F}$, barriers $\mathbb{B}$, and initial writes $\mathbb{I}$. In addition to base events, new tags (t) can also be derived using union ($s \cup s$), intersection ($s \cap s$), and set difference ($s \setminus s$). Similarly, relations (r) consists of base relations(b) e.g., program order po, read-from rf, coherence co; and new relations derived from them using union ($r \cup r$), intersection ($r \cap r$), difference ($r \setminus r$), relation composition ($r; r$), and relation inverse (r^{-1}). Axioms (axm) define the validity of weak behaviors according to the memory consistency model.

A weak behavior of a concurrent program is defined by the values that threads write to and read from the shared memory. Formally, a behavior is a tuple $(\mathbb{X}, \mathsf{rf}, \mathsf{co})$ of executed events, the read-from relation, and the coherence order relation. A behavior is allowed by a

[1] https://github.com/hernanponcedeleon/Dat3M

$$
\begin{array}{rcl}
mm & ::= & def \mid axm \mid mm \wedge mm \\
axm & ::= & \textbf{acyclic}(r) \mid \textbf{irreflexive}(r) \mid \textbf{empty}(r) \\
def & ::= & \texttt{let}\ d := r \mid \texttt{let}\ d := s \\
r & ::= & b \mid d \mid r^{-1} \mid r;r \mid r \cap r \mid r \cup r \mid r \setminus r \\
b & ::= & \mathsf{po} \mid \mathsf{rf} \mid \mathsf{co} \mid \mathsf{loc} \mid [t] \mid t \times t \mid \ldots \\
s & ::= & t \mid d \mid s \cap s \mid s \cup s \mid s \setminus s \\
t & ::= & \mathbb{I} \mid \mathbb{W} \mid \mathbb{R} \mid \mathbb{F} \mid \mathbb{B} \mid \ldots
\end{array}
$$

Figure 1: The `.cat` language for consistency models [5].

consistency model if it satisfies all axioms of the model, otherwise it is forbidden. The set of allowed behaviors defines the semantics of the program with respect to the consistency model. DARTAGNAN encodes a program's semantics w.r.t. a memory model as a SAT modulo theories (SMT) formula, and then uses BMC to detect violations of safety, liveness, or data race freedom (DRF) [6, 8] as defined by the model.

3 GPU Consistency Analysis

The most naive form of encoding for BMC is using one Boolean variable for each combination of a relation and an event pair. If the variable is set, then the pair belongs to the relation. This form generates a large formula and verification does not scale well. For a more compact encoding, DARTAGNAN employs static analysis techniques to compute the lower and upper bounds of the relations [9]. In this section, we introduce the unique features of GPU consistency models, which differ from those of CPU models, and describe how we incorporate support for these features.

Memory Scopes. To achieve high parallelism, modern GPUs utilize numerous processing cores to execute a massive numbers of threads in parallel. However, the latency of global memory accesses for thread communication tends to be notably high. To address this challenge, scopes were introduced as a synchronization mechanisms providing ordering guarantees only among a specific subset of threads. This allows for optimization of synchronization between threads.

Memory Fences and Control Barriers. GPUs offer fine-grained synchronization mechanisms via memory fences and control barriers. GPU memory fences are similar to CPU fences and can be used to synchronize accesses to shared memory. Control barrier ensures that all threads within a specified scope reach the barrier before any of them can proceed further.

Address Spaces. Modern GPUs use specialized accelerators with dedicated caches to boost performance [1]. Compared to generic memory paths, such specialized caches often provide weaker coherence guarantees. Moreover, the same memory address can be accessed via the conventional memory path and via a specialized cache. PTX introduced the notion of proxies to handle specialized accelerators with non-coherent cache hierarchies [11]. VULKAN uses the concept of storage classes to represent memory accesses via different cache types.

Cache Control. Both the PTX and VULKAN models use release-acquire semantics to establish memory synchronization. However, they choose different strategies for cache control. In PTX, the transitive nature of causality makes operations preceding a release pattern from one thread visible to other threads. VULKAN introduces the notions of availability and visibility to explicitly define the involvement of weak memory operations into a synchronization.

All static relations are fully computed from the source code of the program. The precise

RELATION	PRECISE COMPUTATION	SMT ENCODING
sr	$\{(e_1,e_2) \in \mathbb{X} \times \mathbb{X} \mid thread(e1) \in scope(e2) \wedge thread(e2) \in scope(e1)$ $\wedge \neg mutExcl(e_1,e_2) \wedge visibleFrom(thread(e_1), thread(e_2), scope(e_1))\}$	$(\mathsf{exec}_{e_1} \wedge \mathsf{exec}_{e_2})$ $\Leftrightarrow \mathsf{r}(e_1,e_2)$
$\mathsf{r} \in \{\mathsf{scta}, \mathsf{ssg}, \mathsf{swg}, \mathsf{ssw}\}$	$\{(e_1,e_2) \in \mathbb{X} \times \mathbb{X} \mid \neg mutExcl(e_1,e_2) \wedge$ $visibleFrom(thread(e_1), thread(e_2), \mathsf{r})$	
syncbar	$\{(e_1,e_2) \in \mathbb{B} \times \mathbb{B} \mid id(e_1) = id(e_2) \wedge \neg mutExcl(e_1,e_2)\}$	

Table 1: Precise computation and SMT encoding for static base relations.

computation along with the SMT encodings of these static relations are shown in Table 1. The bounds for the dynamic relations are provided in Table 2, while their SMT encodings are detailed in Table 3. Complete models can be found from the DARTAGNAN repository.

RELATION	LOWER BOUND	UPPER BOUND
sync_barrier	$\{(e_1,e_2) \in \mathbb{B} \times \mathbb{B} \mid id(e_1) = id(e_2) \wedge \neg mutExcl(e_1,e_2)$ $\wedge sameCTA(thread(e_1), thread(e_2))\}$	$\{(e_1,e_2) \in \mathbb{B} \times \mathbb{B} \mid \neg mutExcl(e_1,e_2)$ $\wedge sameCTA(thread(e_1), thread(e_2))\}$
sync_fence	$\emptyset$	$\{(e_1,e_2) \in \mathbb{F} \times \mathbb{F} \mid memOrd(e_1) = SC$ $\wedge memOrd(e_2) = SC \wedge \neg mutExcl(e_1,e_2)\}$
vloc	$\{(e_1,e_2) \in \mathbb{M} \times \mathbb{M} \mid mustAlias(e_1,e_2) \wedge$ $sameVirtual(e_1,e_2) \wedge \neg mutExcl(e_1,e_2)\}$	$\{(e_1,e_2) \in \mathbb{M} \times \mathbb{M} \mid mayAlias(e_1,e_2) \wedge$ $sameVirtual(e_1,e_2) \wedge \neg mutExcl(e_1,e_2)\}$

Table 2: Lower and upper bounds for dynamic base relations.

$\bigwedge_{e_1,e_2 \in \mathbb{X}}$	$(\mathsf{exec}_{e_1} \wedge \mathsf{exec}_{e_2} \wedge id(e_1) = id(e_2)) \Leftrightarrow \mathsf{sync\_barrier}(e_1,e_2)$
$\bigwedge_{(e_1,e_2) \in [\mathrm{F}\ \&\ \mathrm{SC}];\mathsf{sr};[\mathrm{F}\ \&\ \mathrm{SC}]}$	$((\mathsf{exec}_{e_1} \wedge \mathsf{exec}_{e_2}) \Leftrightarrow (\mathsf{sync\_fence}(e_1,e_2) \vee \mathsf{sync\_fence}(e_2,e_1))$ $\wedge (\mathsf{sync\_fence}(e_1,e_2) \Rightarrow \mathsf{clk}_{e_1}^{\mathsf{sync\_fence}} < \mathsf{clk}_{e_2}^{\mathsf{sync\_fence}}))$
$\bigwedge_{e_1,e_2 \in \mathbb{X}}$	$\mathsf{vloc}(e_1,e_2) \Rightarrow (\mathsf{exec}_{e_1} \wedge \mathsf{exec}_{e_2} \wedge addr(e_1) = addr(e_2))$
$\bigwedge_{e_1,e_2 \in \mathbb{X}}$	$\mathsf{co}(e_1,e_2) \Rightarrow (\mathsf{exec}_{e_1} \wedge \mathsf{exec}_{e_2} \wedge addr(e_1) = addr(e_2) \wedge \mathsf{clk}_{e_1}^{\mathsf{co}} < \mathsf{clk}_{e_2}^{\mathsf{co}})$

Table 3: SMT encoding for dynamic base relations. The encoding of co only affects PTX.

During the validation of the translated models, we identified a bug in the original VULKAN consistency model. A simple litmus test illustrating this issue is shown in Fig. 2. The original model permits both RMW operations to read the same value, which clearly violates atomicity. This issue arises because the original definition of $\mathsf{fr} = (\mathsf{rf}^{-1}; \mathsf{asmo}) | (([\mathbb{I}]; \mathsf{rf}) \wedge -1; ((\mathsf{loc}; [\mathbb{W}]) \backslash \mathsf{id}))$ does not account for the ordering of non-atomic writes. We proposed updating fr to include locord. This fix[2] has been reported to the maintainers of the VULKAN model, and they have confirmed and accepted our proposed change to the VULKAN specification.

```
1  P0@sg 0, wg 0, qf 0          | P1@sg 1, wg 0, qf 0          ;
2  st.av.dv.sc0 x, 1            | cbar.acq_rel.dv.semsc0 0     ;
3  cbar.acq_rel.dv.semsc0 0     | rmw.atom.dv.sc0.add r0, x, 1 ;
4  rmw.atom.dv.sc0.add r0, x, 1 |                              ;
5 exists
6 (P0:r0 == 1 /\ P1:r0 == 1)
```

Figure 2: A program showing a bug in the VULKAN model.

[2] https://github.com/KhronosGroup/Vulkan-MemoryModel/issues/36

References

[1] Mixed-proxy extensions for the NVIDIA PTX memory consistency model. https://github.com/NVlabs/mixedproxy. Accessed: 09/26/2023.

[2] SPIR-V, Extended Instruction Set, and Extension Specifications. https://registry.khronos.org/SPIR-V/. Accessed: 06/12/2024.

[3] Vulkan-MemoryModel. https://github.com/KhronosGroup/Vulkan-MemoryModel. Accessed: 09/26/2023.

[4] Jade Alglave. A shared memory poetics. *These de doctorat, L'université Paris Denis Diderot*, 2010.

[5] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016.

[6] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[8] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: One tool for all models. In Francesco Ranzato, editor, *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, volume 10422 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017.

[9] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 355–365. Springer, 2019.

[10] Daniel Jackson. Alloy: A language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019.

[11] Daniel Lustig, Simon Cooksey, and Olivier Giroux. Mixed-proxy extensions for the NVIDIA PTX memory consistency model: Industrial product. In Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang, editors, *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, pages 1058–1070. ACM, 2022.

[12] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. A formal analysis of the NVIDIA PTX memory consistency model. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 257–270. ACM, 2019.

[13] Haining Tong, Natalia Gavrilenko, Hernán Ponce de León, and Keijo Heljanko. Towards unified analysis of GPU consistency. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*, 2024. Accepted manuscript, to be published.

Deadlock and Buffer Overflow Detection for Timed Kahn Process Networks

Behnam Khodabandeloo[1], Chengzi Huang[1], Morteza Mohaqeqi[1], Susanne Graf[2], and Wang Yi[1]

[1] Uppsala University, Uppsala, Sweden
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, France

Abstract

We propose modeling real-time systems as a network of real-time tasks connected by communication channels, which are either FIFO queues or registers. The model is a timed extension of Kahn Process Networks and has been shown to be both timing and functionally deterministic. It is designed for the efficient analysis of safety properties in real-time systems. In this work, we develop a method for efficiently checking the absence of buffer overflow and deadlock in such networks.

1 Introduction

Achieving determinism in concurrent systems is inherently difficult due to the non-deterministic nature of their software architectures. However, guaranteeing predictability is vital, especially in safety-critical systems, like autonomous vehicles, where unexpected behavior can lead to catastrophic outcomes [1]. As the complexity of modern systems continues to grow—driven by the integration of AI, specialized hardware, and increasingly sophisticated algorithms—the need for deterministic behavior becomes even more critical.

The MIMOS model [2] extends Kahn Process Networks by adding timing aspects to represent real-time systems as networks of tasks connected via communication channel, which are either FIFO queues or registers. This model ensures both timing and functional determinism. Each task periodically computes a typed function, processing a tuple of data segments as input and producing a tuple of data segments as output. If, at the start of a new period, there are enough data elements for a complete input, as specified by the type of the function, it will be computed, and the output will be delivered at the deadline. In this work, we develop a method to efficiently check for the absence of buffer overflow and deadlock in such networks.

A MIMOS model is represented as a directed graph $G = (\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ is the set of nodes (software components) and $\mathcal{L}$ is the set of links (channels for data exchange). Each node $N_i \in \mathcal{N}$ is defined by the tuple (I_i, O_i, P_i, D_i):

- I_i: A set of input ports, where each $in \in I_i$ is described as $\langle \text{Type}, K_{i,in} \rangle$, where:
  - *Type* can be one of the following:

    FIFO: Blocking, requires non-empty input for activation.

    REGISTER: Non-blocking, returns the latest available value, accommodating undersampling.

    UPTO: Non-blocking FIFO, returns available data or an empty value if none is present, handling oversampling.
  - $K_{i,in}$ represents the segment size associated with the input port.
- O_i: A set of output ports, producing $H_{i,out}$ data tokens on output port $out \in O_i$ per activation at the end of the period.

- P_i: Activation period; activation requires all FIFO input ports to have sufficient tokens for the node to execute.
- D_i: Relative deadline, by which execution must complete in each period.

A link $L \in \mathcal{L}$ is represented as $(\text{src}(L).out, \text{sink}(L).in)$, connects the output port *out* of node $\text{src}(L)$ to the input port *in* of node $\text{sink}(L)$, transferring data tokens during each activation. Note that if feedback links are inadequately initialized or buffers are insufficiently sized, the system may experience deadlock or overflow. This work only addresses deadlocks and buffer overflows for long running executions where overflow (deadlock) occurs independently of the chosen buffer size (number of initial buffer elements).

2 The proposed solution

The proposed approach involves two phases: In the first phase, we calculate the actual period of each node within the MIMOS network for long running executions. Using these results, we then examine all links within the network to identify any buffer-overflow issues that may be present. Additionally, we investigate the network to detect the presence of deadlocks.

Formulation: Let P_i^A and P_i^I represent the actual period and maximum input period of node N_i, respectively. Our solution employs a fixed-point iteration method to compute these values at each iteration t, denoted as $P_i^A(t)$ and $P_i^I(t)$. Using the actual periods at iteration t, $\{P_1^A(t), ..., P_{|\mathcal{N}|}^A(t)\}$, the maximum input period at iteration $t+1$ is defined by:

$$P_i^I(t+1) = \begin{cases} \max\limits_{\substack{\forall in \in I_i, \\ \exists L=(N_s.out, N_i.in)}} \left(\text{Type}(in) = \text{FIFO} \implies \left(P_s^A(t) \times \frac{K_{i,in}}{H_{s,out}}\right), 0\right) & \text{if } |I_N^i| \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The actual period at iteration $t+1$ is:

$$P_i^A(t+1) = \max(P_i^A(t), P_i^I(t+1))$$

Since the actual period is bounded by the nominal period, we initialize the actual period at $t = 0$ as:

$$P_i^A(0) = P_i$$

Algorithm: To determine the actual period of each node in a MIMOS network, Algorithm 1 initializes each node's period to its nominal value (line 1). To improve convergence, it accounts for data dependencies by ordering nodes such that each node's predecessors are computed first. Since cycles exist in the MIMOS network, conventional topological sorting cannot be applied directly. By removing feedback links denoted by $\mathcal{L}^{BE}$(back edges in DFS), topological sorting becomes possible (line 2). Taking into account all edges, the algorithm then computes in "topological" order, each node's actual period by a fixed-point iteration (lines 3–17), terminating when all periods are stable (lines 14-16). For deadlock-free networks, the algorithm is guaranteed to converge within the defined number of iterations (see proof, below), with non-convergence indicating a potential deadlock (line 18). Figure 1a shows a case where the algorithm converges, reaching a final result after three iterations. Figure 1b depicts a scenario where the algorithm fails to converge.

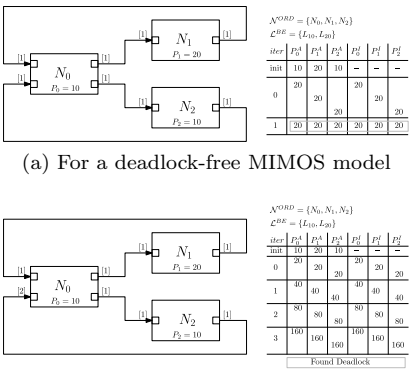
Buffer overflow: Algorithm 1 computes the actual period (P_i^A) for each node in the MIMOS network, which can be used to identify links at risk of buffer overflow. For a link $L = (N_j.out, N_i.in)$, the risk is evaluated using the inequality: $\frac{K_{i,in}}{P_j^A} \leq \frac{H_{j,out}}{P_i^A}$. By comparing

the data production of node N_j to the data consumption of node N_i, we can assess whether the link is operating at or near maximal capacity, and choose an appropriate stream input type to prevent buffer overflow errors from occurring.

Deadlock: The output of Algorithm 1 can help to detect deadlocks in a MIMOS system. Since each node's actual period increases monotonically, the algorithm should converge within $1 + |\mathcal{L}^{BE}|$ iterations in a deadlock-free network. If the algorithm does not stabilize after this point, a deadlock is present.

Algorithm 1 Actual Period Calculation

1: $\forall N_i \in \mathcal{N}, P_i^A \leftarrow P_i$
2: $\mathcal{N}^{ORD}, \mathcal{L}^{BE} \leftarrow getNodesPartialOrder(G(\mathcal{N}, \mathcal{L}))$
3: $iter \leftarrow 0$
4: **while** $iter \leq 1 + |\mathcal{L}^{BE}|$ **do**
5: $iter \leftarrow iter + 1$
6: **for each** $N_i \in \mathcal{N}^{ORD}$ **do**
7: **if** $|I_N^i| \neq 0$ **then**
8: $P_i^I \leftarrow$ compute using Equation 1
9: **else**
10: $P_i^I \leftarrow 0$
11: **end if**
12: $P_i^A \leftarrow \max(P_i^A, P_i^I)$
13: **end for**
14: **if** not found changing in P^A **then**
15: **return** true ▷ Not found deadlock
16: **end if**
17: **end while**
18: **return** false ▷ Found deadlock



(a) For a deadlock-free MIMOS model

(b) For a MIMOS model with deadlock

Figure 1: The proposed algorithm computations for actual period calculation

3 The correctness of the algorithm

Coverage: At each iteration of the algorithm, the actual period of a node can only increase (as indicated by line 12). If the network is deadlock-free, we can fix periods for each node satisfying the relationship imposed by the algorithm. The algorithm must converge in a finite number of steps.

Boundedness: If the given network has no feedback edges, the algorithm converges in the first iteration, as node periods are computed according to the dependency order. Now, consider the end of the t^{th} iteration and a node N_z, which is minimal in the dependency order, where $P_z^A(t-1) < P_z^A(t)$. This condition implies that $P_z^A(t) = P_z^I(t)$. Since the actual period of all nodes smaller in the dependency orders remains unchanged in this iteration (by choice of N_z), there must exist a feedback link (e.g., $L_{yz} = (N_y.out, N_z.in)$) to node N_z that causes this change. This implies that the actual data rate of feedback link L_{yz} becomes stable after iteration t. A similar claim can be made for another feedback link at iteration $(t-1)$. Generally, there is at least one feedback link whose actual data rate stabilizes in each iteration of the algorithm. Given that there are $|\mathcal{L}^{BE}|$ feedback links in total, the algorithm converges within at most $1 + |\mathcal{L}^{BE}|$ iterations.

References

[1] Marten Lohstroh, Soroush Bateni, Christian Menard, Alexander Schulz-Rosengarten, Jeronimo Castrillon, and Edward A. Lee. Deterministic coordination across multiple timelines. *ACM Trans. Embed. Comput. Syst.*, 23(5), aug 2024.

[2] Wang Yi, Morteza Mohaqeqi, and Susanne Graf. MIMOS: A deterministic model for the design and update of real-time systems. In *Coordination Models and Languages*, volume 13271 of *Lecture Notes in Computer Science*, pages 17–34. Springer, 2022.

On the Operational Semantics of the Free Cornering with Protocol Choice

Chad Nester[1] and Niels Voorneveld[2*]

[1] University of Tartu, Tartu, Estonia
nester@ut.ee
[2] Cybernetica AS, Tallinn, Estonia
niels.voorneveld@cyber.ee

This work concerns the dynamics (operational semantics) of the free cornering with protocol choice of a monoidal category. The free cornering construction provides a double categorical notion of program interaction, but presently exists only as a formal semantics. Our project is to work backwards from this formal semantics to obtain a simple interactive programming language. Our first step is to orient some of the equations of the free cornering with protocol choice to obtain a rewriting system, by which interactive programs may be evaluated. In this extended abstract we summarize our progress towards this goal.

We begin by recalling the free cornering with choice of a monoidal category[1], which is defined to be the single-object double category (see e.g., [1]) $\ulcorner\mathbb{A}\urcorner^{+\times}$ with horizontal edge monoid $\ulcorner\mathbb{A}\urcorner^{+\times}_H = (\mathbb{A}_0, \otimes, I)$ given by the the object monoid of $\mathbb{A}$, vertical edge monoid $\ulcorner\mathbb{A}\urcorner^{+\times}_V$ as in Figure 1, and cells as in Figure 2. Objects of $\mathbb{A}$ are interpreted as collections of resources, and morphisms $f : A \to B$ of $\mathbb{A}$ are interpreted as processes which consume the resources of A and produce the resources of B. Elements of $\ulcorner\mathbb{A}\urcorner^{+\times}_V$ are interpreted as *interaction protocols*. Every such protocol has a *left participant* and a *right participant*. To carry out the protocol A° the left participant sends the right participant an instance of A. Dually, to carry out $A^\bullet$ the right participant sends the left participant an instance of A. To carry out $U \otimes W$ the participants first carry out U and then carry out W, and I is carried out by doing nothing. Finally, to carry out $U + W$ the left participant chooses which of U and W will happen, and the participants carry out that protocol. Dually, to carry out $U \times W$ the right participant chooses which protocol will happen. A cell of $\ulcorner\mathbb{A}\urcorner^{+\times}$ of type $(U \begin{smallmatrix} A \\ B \end{smallmatrix} W)$ describes a procedure for transforming the resources of A into the resources of B while acting as the left participant of the interaction protocol W, and as the right participant of U. For a more detailed account see [2].

The dynamics of programming languages are commonly specified by term rewriting systems. We require an analogous sort of rewriting system on a single-object double category. Term rewriting systems are a special case of what we will call *monoidal category rewriting systems*, which consist of a category $\mathbb{X}$ together with a generating binary relation $\to_R \subseteq \mathbb{X}(X, Y)$ for each $X, Y \in \mathbb{X}_0$ which is coherent with respect to composition and the tensor product. For example, we ask that if $f \to_R g$ then $f \otimes h \to_R g \otimes h$. The usual term rewriting systems form a category rewriting system on the category of tuples of terms. Monoidal category rewriting systems suggest a notion of rewriting system on a single-object double category $\mathbb{D}$, in which we have a binary relation $\to_R \subseteq \mathbb{D}(U \begin{smallmatrix} A \\ B \end{smallmatrix} W) \times \mathbb{D}(U \begin{smallmatrix} A \\ B \end{smallmatrix} W)$ for all $U, W \in \mathbb{D}_V$ and $A, B \in \mathbb{D}_H$ which is coherent with respect to vertical and horizontal composition. For example, we ask that if

*Niels Voorneveld is supported by the European Union under Grant Agreement No. 101087529. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

[1] In particular, a *strict* monoidal category. In this document, "monoidal category" means "strict monoidal category".

The elements of $\ulcorner \mathbb{A} \urcorner_V^{+\times}$ are formed according to the following rules:

$$\frac{}{I \in \ulcorner \mathbb{A} \urcorner_V^{+\times}} \qquad \frac{A \in \mathbb{A}_0}{A^\circ \in \ulcorner \mathbb{A} \urcorner_V^{+\times}} \qquad \frac{A \in \mathbb{A}_0}{A^\bullet \in \ulcorner \mathbb{A} \urcorner_V^{+\times}}$$

$$\frac{U, W \in \ulcorner \mathbb{A} \urcorner_V^{+\times}}{U \otimes W \in \ulcorner \mathbb{A} \urcorner_V^{+\times}} \qquad \frac{U, V \in \ulcorner \mathbb{A} \urcorner_V^{+\times}}{U + V \in \ulcorner \mathbb{A} \urcorner_V^{+\times}} \qquad \frac{U, V \in \ulcorner \mathbb{A} \urcorner_V^{+\times}}{U \times V \in \ulcorner \mathbb{A} \urcorner_V^{+\times}}$$

and are subject to the following equations:

$$(U \otimes W) \otimes V = U \otimes (W \otimes V) \qquad I \otimes U = U = U \otimes I$$

Figure 1: $\ulcorner \mathbb{A} \urcorner_V^{+\times}$, the vertical edge monoid of $\ulcorner \mathbb{A} \urcorner^{+\times}$.

$f \to_R g$ then $f \mid h \to_R g \mid h$. Our goal is to orient equations of the free cornering with protocol choice to obtain well-behaved rewriting systems of this kind.

We consider the rewriting system $\Rightarrow$ indicated in Figure 2 on the single-object double category obtained by retaining the rest of the equations. This rewriting system is terminating and confluent. Further, suppose that we are given a monoidal category rewriting system $\to_R$ on $\mathbb{A}$ representing the dynamics of $\mathbb{A}$ as a non-interactive programming language. Let $\Rightarrow_R$ consist of the rewrites of $\Rightarrow$ together with a rewrite $\ulcorner f \urcorner \Rightarrow_R \ulcorner g \urcorner$ for each rewrite $f \to_R g$. Then we have:

Theorem 1.

(i) If $\to_R$ is terminating then $\Rightarrow_R$ is terminating.

(ii) If $\to_R$ is confluent then $\Rightarrow_R$ is confluent.

Thus, our approach to the dynamics of interacting programs is modular in the sense that it is well-behaved with respect to the dynamics of the underlying programming language. Further, a careful inductive argument concerning the elimination of "internal" redexes yields a proof of a conjecture from [2]:

Theorem 2. *Let $\mathbb{A}$ be a monoidal category. Then there is an isomorphsim of categories $\mathbf{V}\ulcorner \mathbb{A} \urcorner^{+\times} \cong \mathbb{A}$, where $\mathbf{V}\ulcorner \mathbb{A} \urcorner$ is the category of* vertical cells *of $\ulcorner \mathbb{A} \urcorner$ (see e.g., [1]).*

In future work, we hope to extend our rewriting system to account for all of the structure added to the free cornering in [2]. This includes *active choice*, which models case statements in the non-interactive fragment of the system and their relationship to protocol choice, as well as *protocol iteration*, which allows protocols to be repeated an indefinite number of times based on the choices of the participants. Eventually, we would like to implement a simple interactive programming language based on these ideas. To that end, we will attempt to design an abstract machine describing the efficient implementation of the rewriting semantics presented here.

References

[1] C. Nester. Concurrent Process Histories and Resource Transducers. *Logical Methods in Computer Science*, Volume 19, Issue 1, January 2023.

The cells of $\ulcorner \mathbb{A} \urcorner^{+\times}$ are formed according to the following rules:

$$\frac{f \in \mathbb{A}(A,B)}{\ulcorner f \urcorner : \left(I \begin{smallmatrix} A \\ B \end{smallmatrix} I\right)} \qquad \frac{A \in \mathbb{A}_0}{1_A : \left(I \begin{smallmatrix} A \\ A \end{smallmatrix} I\right)} \qquad \frac{U \in \ulcorner \mathbb{A} \urcorner^{+\times}_V}{id_U : \left(U \begin{smallmatrix} I \\ I \end{smallmatrix} U\right)}$$

$$\frac{a : \left(U \begin{smallmatrix} A \\ B \end{smallmatrix} W\right) \qquad b : \left(U' \begin{smallmatrix} B \\ C \end{smallmatrix} W'\right)}{\frac{a}{b} : \left(U \otimes U' \begin{smallmatrix} A \\ C \end{smallmatrix} W \otimes W'\right)} \qquad \frac{a : \left(U \begin{smallmatrix} A \\ B \end{smallmatrix} W\right) \qquad b : \left(W \begin{smallmatrix} A' \\ B' \end{smallmatrix} V\right)}{a \mid b : \left(U \begin{smallmatrix} A \otimes A' \\ B \otimes B' \end{smallmatrix} V\right)}$$

$$\frac{A \in \mathbb{A}_0}{\mathsf{get}^A_{\mathsf{L}} : \left(A^\circ \begin{smallmatrix} I \\ A \end{smallmatrix} I\right)} \qquad \frac{A \in \mathbb{A}_0}{\mathsf{put}^A_{\mathsf{R}} : \left(I \begin{smallmatrix} A \\ I \end{smallmatrix} A^\circ\right)} \qquad \frac{A \in \mathbb{A}_0}{\mathsf{get}^A_{\mathsf{R}} : \left(I \begin{smallmatrix} I \\ A \end{smallmatrix} A^\bullet\right)} \qquad \frac{A \in \mathbb{A}_0}{\mathsf{put}^A_{\mathsf{L}} : \left(A^\bullet \begin{smallmatrix} A \\ I \end{smallmatrix} I\right)}$$

$$\frac{U, W \in \ulcorner \mathbb{A} \urcorner^{+\times}_V}{\pi_0^{U,W} : \left(U \times W \begin{smallmatrix} I \\ I \end{smallmatrix} U\right)} \qquad \frac{U, W \in \ulcorner \mathbb{A} \urcorner^{+\times}_V}{\pi_1^{U,W} : \left(U \times W \begin{smallmatrix} I \\ I \end{smallmatrix} W\right)} \qquad \frac{a : \left(V \begin{smallmatrix} A \\ B \end{smallmatrix} U\right) \qquad b : \left(V \begin{smallmatrix} A \\ B \end{smallmatrix} W\right)}{a \times b : \left(V \begin{smallmatrix} A \\ B \end{smallmatrix} U \times W\right)}$$

$$\frac{U, W \in \ulcorner \mathbb{A} \urcorner^{+\times}}{\varkappa_0^{U,W} : \left(U \begin{smallmatrix} I \\ I \end{smallmatrix} U + W\right)} \qquad \frac{U, W \in \ulcorner \mathbb{A} \urcorner^{+\times}}{\varkappa_1^{U,W} : \left(W \begin{smallmatrix} I \\ I \end{smallmatrix} U + W\right)} \qquad \frac{a : \left(U \begin{smallmatrix} A \\ B \end{smallmatrix} V\right) \qquad b : \left(W \begin{smallmatrix} A \\ B \end{smallmatrix} V\right)}{a + b : \left(U + W \begin{smallmatrix} A \\ B \end{smallmatrix} V\right)}$$

and are subject to the the following equations. Here we write $\Rightarrow$ to indicate those equations that become the generating rewrites of our rewriting system. That is, we both define the free cornering with protocol choice (in which every $a \Rightarrow b$ is taken as an equation) and also indicate the generating rewrites $a \Rightarrow b$ of our double category rewriting system.

$$\frac{\ulcorner f \urcorner}{\ulcorner g \urcorner} = \ulcorner f ; g \urcorner \qquad \ulcorner f \urcorner \mid \ulcorner g \urcorner = \ulcorner f \otimes g \urcorner \qquad 1_A = \ulcorner 1_A \urcorner \qquad \frac{1_A}{a} = a = \frac{a}{1_B}$$

$$id_U \mid a = a = a \mid id_W \qquad \frac{a}{\left(\frac{b}{c}\right)} = \frac{\left(\frac{a}{b}\right)}{c} \qquad (a \mid b) \mid c = a \mid (b \mid c) \qquad \frac{a \mid b}{c \mid d} = \frac{a}{c} \mid \frac{b}{d}$$

$$\mathsf{get}^A_{\mathsf{R}} \mid \mathsf{put}^A_{\mathsf{L}} \Rightarrow 1_A \qquad \frac{\mathsf{get}^A_{\mathsf{R}}}{\mathsf{put}^A_{\mathsf{L}}} \Rightarrow id_{A^\bullet} \qquad \mathsf{put}^A_{\mathsf{R}} \mid \mathsf{get}^A_{\mathsf{L}} \Rightarrow 1_A \qquad \frac{\mathsf{get}^A_{\mathsf{L}}}{\mathsf{put}^A_{\mathsf{R}}} \Rightarrow id_{A^\circ}$$

$$(a \times b) \mid \pi_0 \Rightarrow a \qquad (a \times b) \mid \pi_1 \Rightarrow b \qquad (h \mid \pi_0) \times (h \mid \pi_1) \Rightarrow h$$

$$\varkappa_0 \mid (a + b) \Rightarrow a \qquad \varkappa_1 \mid (a + b) \Rightarrow b \qquad (\varkappa_0 \mid h) + (\varkappa_1 \mid h) \Rightarrow h$$

We note that this definition differs from the one given in [2] in that we ask for a version of the surjective paring axiom for our binary $+$ and $\times$ operations instead of the corresponding uniqueness condition. The resulting double categories are identical.

Figure 2: The cells of $\ulcorner \mathbb{A} \urcorner^{+\times}$.

[2] C. Nester and N. Voorneveld. Protocol Choice and Iteration for the Free Cornering. *Journal of Logical and Algebraic Methods in Programming*, 137:100942, 2024.

DropShadow: Hypercontracts in Go*

Andreas Kjeldgaard Brandhøj, Dat Tommy Thanh Dieu, Kasper Vesteraa, Danny Bøgsted Poulsen, René Rydhof Hansen, and Kim Guldstrand Larsen

Department of Computer Science, Aalborg University, Aalborg, Denmark
tommydieu@outlook.dk, kasperwesteraa@gmail.com, {akbr, dannybpoulsen, rrh, kgl}@cs.aau.dk

Go is a widely used language for embedded devices, systems programming, cloud and network services. All require a high degree of quality assurance and, in some cases, service level agreements. The Go community has made significant efforts to provide testing tools, evident in features such as the built-in support for property-based testing and fuzzing. Developers often combine different approach to cover various aspects of their programs, but many systems require more thorough testing and documentation capabilities than the provided tools support. Contract-based programming is an approach with the potential to bridge this gap. However, contracts generally only consider a single execution of the function, severely limiting the properties which can be expressed in the contract and their corresponding tests. Here we present DropShadow, a novel tool for Go, which enables experimental generation of hypertests for hypercontracts, which universally quantifies over pairs of executions. To limit the overhead of testing the contracts, DropShadow automatically handles the full process of generating automatic hypertests and executing them to find counterexamples. We illustrate how DropShadow can specify certain hyperproperties such as the secure information flow policy non-interference to improve the software's security and quality. Experiments are promising, but service level agreements are not yet supported, and contract complexity is limiting the full potential.

Contract-based programming is an approach to developing software where developers specify the requirements on input to a function and the guarantees on the output of the function in terms of pre- and post-conditions [11]. Hyperproperties is an extension to trace properties and is a set of sets of traces (i.e. set of trace properties) [5]. Contracts can include hyperproperties by specifying a relation between multiple executions of the same function, in this case we call them hypercontracts. The extension to contracts is desirable since some properties can only be expressed as hyperproperties such as non-interference and observational determinism [5].

DropShadow enables developers to specify relations over inputs and outputs of pairs of executions, which allows specifying hyperproperties universally quantifying over pairs of finite traces. These hypercontracts define expected behavior, and DropShadow automatically integrates them into the code by adding assertions to catch any potential violations. Additionally, DropShadow generates hypertests and run them with input generators, which are user-defined constructs implementing both inclusion checks and the generation of included elements.

```
1 // assume: value = AnyNumber[int](0)
2 // expect: ret >= 0 && ((value < 0 && ret == -value) ||
3 //         (value >= 0 && ret == value && ))
4 func Absolute(value int) int {
5     if value < 0 {
6         return -value
7     }
8     return value
9 }
```

*This abstract reports on the work done on the MSc theis [2].

Existing tools such as grpc-go-contracts [8] and gocontract [15] enables developers to write pre- and post-conditions in function bodies. Whereas, go-contracts [13] and dbc4go [3] enables contractual specifications as function documentation over individual executions. The tools inject assertions and rely on the existing tests to check the contracts. Existing tools can be used for hyperproperties but requires the sequential self-composition to be manually constructed. The black-box fuzzer LeakFuzzer [1], is limited to testing non-interference by segmenting its input and keeping track of the public and private input and storing previous outputs. A common aspect of many fuzzers is the categorization of interesting inputs, which are inputs uncovering new paths of the fuzz target. A framework for hypertests was developed in [9] wherein they describe a coverage metric as the uncovering of the program as pairs of executions.

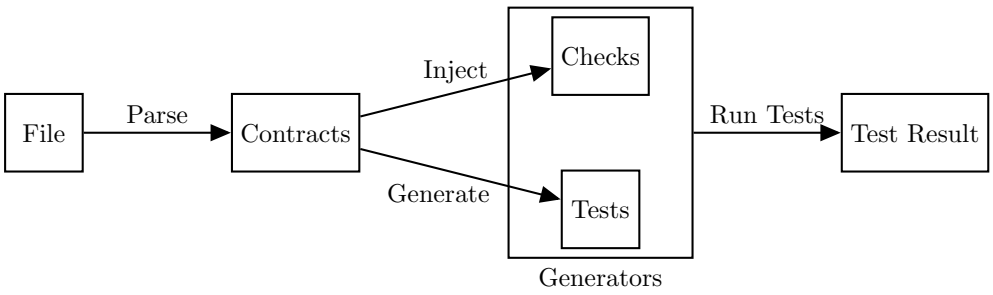


Figure 1: A summary of DropShadow's workflow, starting with parsing a Go source file, followed by injecting hyperproperty checks and generating tests, and lastly running them, potentially exposing counterexamples.

DropShadow begins by parsing the Go source files using the standard library parser. During parsing, the contracts for each function are identified as function documentation and parsed. The contracts are then used for two purposes: injecting checks (both assertions and sequential self-composition for the hyperproperty) and generating hypertests. Both tasks rely on generators, depicted in Figure 1. Tests use the generators' generation, whilst the injected assertions use inclusion checks. Once all tests are generated and assertions inserted, the tests are executed, potentially exposing counterexamples for the hyperproperty.

As similar existing contract tools do not handle the generation of tests and test inputs, they rely on users potentially exploring their contracts with tools such as rapid [14], gopter [10], or standard library quick [7] to perform property-based testing. In contrast, the generators in DropShadow serve a similar purpose to the ones in QuickCheck [4], but with inclusion checks. It relies on the generator's implementation to describe the required conditions of the input. The benefit of using generators is the potential for efficient user defined-custom generation, which supports any type, can be used in contracts, and test input generation. An example is the `GE[int](n)` generator, which randomly generates an `int` greater than or equal to n and can check if a number satisfies the constraint.

Through experimentation, we found the post-condition to often specify an outcome in specific bounds on the input - like that of partition testing. To alleviate this, we defined named regions to split the post-conditions depending on the input value. This allowed the specification of some contracts to be more readable. Depending on which regions the input belongs to, a specific post-condition must be satisfied. The contract is breached if any input either does not fall within any region or it falls within some regions but the post-condition is not satisfied. the contract for `Absolute` is an example of a subtle breach, which can be found by calling it with

the minimum integer value which overflows before returned.

```
// region: Positive
// assume: value = GE[int](0)
// expect: ret >= 0 && ret == value
// region: Negative
// assume: value = LT[int](0)
// expect: ret >= 0 && ret == -value
func Absolute(value int) int {
    if value < 0 {
        return -value
    }
    return value
}
```

Non-interference is a hyperproperty which states that two executions with the same low but different high inputs must have the same observable output. This is supported by DropShadow because the hypercontracts allow the specification of a proceeding call to the function where the inputs and outputs can have a defined relation. A violation of the non-interference property can be found by executing `Retain` twice, once with an even and an odd high input, and compare the outputs.

```
// assume: low = AnyNumber[int]()
// assume: high = AnyNumber[int]()
// hyper:
// assume: low_p = low
// assume: high_p = AnyNumber[int]().Next()
// expect: ret == ret_p
func Retain(low, high int) int {
    if high % 2 == 0 {
        return 0
    }
    return low
}
```

In general, we found examples suggesting the usefulness of the tool to effectively communicating contractual specifications for hyperproperties. However, it faces practical limitations, such as increased runtime, restrictions to pure functions, and the rapid growth of contracts. The use of generators allows for highly specific user-defined types and custom distributions. However, in some cases, using the generators as a part of the contracts can reduce the readability. Go, being a relatively new language, has limited tooling for formal methods. Nevertheless, exploring symbolic execution in combination with sequential self-composition of functions may offer a viable approach to verifying hypercontracts. Moreover, symbolic execution can potentially use the contracts to optimize execution paths by replacing function calls with their corresponding contracts if they have been verified. Some progress has been made in this area by extending Viper [12] for separation logic to support hyperproperties in Hypra [6], as well as the development of Gobra [16], a tool that translates Go code into Viper. However, service level agreements, such as the mean response time for requests with sizes greater than $100MB$ must not exceed $100ms$ which requires probabilistic hyperproperties and a time metric is yet unsupported.

References

[1] Daniel Blackwell, Ingolf Becker, and David Clark. Hyperfuzzing: black-box security hypertesting with a grey-box fuzzer. *CoRR*, abs/2308.09081, 2023.

[2] Andreas Kjeldgaard Brandhøj, Dat Tommy Thanh Dieu, Kasper Westeraa, Danny Bøgsted Poulsen, and René Rydhof Hansen. Dropshadow: Instrumenting Go with Contracts and Automatic Property-based Test Generation of Hyperproperties. Master's thesis, Aalborg University, June 2024.

[3] chavacava. dbc4go. https://github.com/chavacava/dbc4go. Accessed on: 21-10-2024.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[5] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, 2010.

[6] Thibault Dardinier, Anqi Li, and Peter Müller. Hypra: A deductive program verifier for hyper hoare logic. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):1279–1308, 2024.

[7] Google. quick. https://pkg.go.dev/testing/quick. Accessed on: 21-10-2024.

[8] Shayan Hosseini. grpc-go-contracts. https://github.com/shayanh/grpc-go-contracts. Accessed on: 21-10-2024.

[9] Johannes Kinder. Hypertesting: The case for automated testing of hyperproperties. In *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot)*, pages 1–8, 2015.

[10] leanovate. gopter. https://github.com/leanovate/gopter. Accessed on: 21-10-2024.

[11] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990.

[12] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Alexander Pretschner, Doron Peled, and Thomas Hutzelmann, editors, *Dependable Software Systems Engineering*, volume 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 104–125. IOS Press, 2017.

[13] Parquery. gocontracts. https://github.com/Parquery/gocontracts. Accessed on: 21-10-2024.

[14] Gregory Petrosyan. rapid. https://github.com/flyingmutant/rapid. Accessed on: 21-10-2024.

[15] Klassen Software Solutions. gocontracts. https://github.com/klassen-software-solutions/gocontract. Accessed on: 21-10-2024.

[16] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. Gobra: Modular specification and verification of go programs. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 367–379. Springer, 2021.

Strategy Synthesis for First-Order Agent Programs over Finite Traces

Till Hofmann[1] and Jens Claßen[2]

[1] RWTH Aachen University, Germany
till.hofmann@cs.rwth-aachen.de
[2] Roskilde University, Denmark
classen@ruc.dk

In this work,[1] we consider the task of synthesizing an execution strategy for an agent from a high-level description of the initial state of the world, the actions available to the agent, a control program, and a temporal goal. In particular, we look at the case of infinite-state systems with unbounded object domains, based on a specification formalism with first-order expressiveness, as well as exogenous events, triggered by the non-deterministic environment. More specifically, we use the agent programming language Golog [11]. Golog in turn is based on the situation calculus [13, 15], a first-order logic formalism for reasoning about change. Here, a situation calculus action theory (perhaps incompletely) describes the initial state of the world, together with the preconditions and effects of primitive actions at the agent's disposal, while Golog programs then combine primitive actions into more complex behaviours using sequence, iteration, non-deterministic branching, and concurrency [5]. Since both imperative and non-deterministic program constructs are included, this allows for combining programming and planning in a flexible manner.

As an example, adapted from [3], consider a robot that has to clean dirty dishes. It can move between a number of different rooms and the kitchen, load (an arbitrary number of) dirty dishes onto itself located in specific rooms, and unload dishes it carries into the dishwasher in the kitchen. Here we use a modal variant of the situation calculus called $\mathcal{ES}$ [10], where these actions would be represented through functions $goto(x)$, $load(x, y)$, and $unload(x)$, respectively, whereas properties that change due to actions are encoded by *fluent* predicates such as $At(x)$ or $OnRobot(x)$. An action theory $\mathcal{D} = \mathcal{D}_0 \cup \mathcal{D}_{\text{pre}} \cup \mathcal{D}_{\text{post}}$ then consists of three parts:

1. The *initial theory* $\mathcal{D}_0$ is a finite set of axioms that encodes what is true initially. For example, the robot might be in the kitchen, and not carry any dirty dishes yet:

$$At(kitchen) \wedge \neg\exists x\, OnRobot(x)$$

2. The *precondition axiom* $\mathcal{D}_{\text{pre}}$ states when each action a can be executed in terms of the special fluent $\text{Poss}(a)$. For instance, the robot can unload a dirty dish x into the dishwasher iff it is currently carrying x and it is located in the kitchen (the modal operator $\Box$ expresses that the subformula is true now and after any sequence of actions; free variable x is understood as implicitly $\forall$-quantified from the outside):

$$\Box\,\text{Poss}(unload(x)) \equiv OnRobot(x) \wedge At(kitchen)$$

3. The *successor state axioms* (SSAs) $\mathcal{D}_{\text{post}}$ express the changes to fluents' values due to actions. For example, the robot will hold dish x after action a just in case a was the action of loading x from room y, or the robot was already holding x previously and a was not the action of unloading x (the modal operator $[a]$ reads as "after action a"):

$$\Box[a]\, OnRobot(x) \;\equiv\; \exists y.\, a = load(x, y) \vee OnRobot(x) \wedge a \neq unload(x)$$

[1] This paper gives an overview. A full version with all formal details is available at [8].

Given such a theory, a Golog program for the robot could be the following:

loop:
while $\exists x.\, \mathit{OnRobot}(x)$ **do** $\pi x : \{d_1, \ldots, d_m\}.\, \mathit{unload}(x)$;
$\pi y : \{r_1, \ldots, r_n\}.\, \mathit{goto}(y)$;
while $\exists x.\, \mathit{DirtyDish}(x, y)$ **do** $\pi x : \{d_1, \ldots, d_m\}.\, \mathit{load}(x, y)$;
$\mathit{goto}(\mathit{kitchen})$

That is to say, the agent is instructed to iterate (**loop**) a subprogram where in each cycle, it first performs a loop to unload any dishes it is holding into the dishwasher. Afterwards, it chooses a room to move to, where it loads up all dirty dishes (if any) in another loop. Finally, it moves back to the kitchen. Here, π operators denote a finitary non-deterministic choice of argument; e.g., $\pi x : \{d_1, \ldots, d_m\}$ means "choose some x from among dishes $d_1, \ldots, d_m$", where each d_i is an $\mathcal{ES}$ constant.

The program thus defines a general structure for the robot's course of action, but leaves certain choices open, such as what room to go next to. Typically, it is assumed that the agent is in complete control, i.e., that all such non-determinism is "angelic". More recently, different forms of "demonic" non-determinism have been studied [4, 2], where actions have outcomes that are determined by the environment. For example, we might have another program running in parallel that occasionally triggers an action to place some new dirty dish x into some room y (to simulate a dynamic environment with people that use the dishes):

loop: $\pi x : \{d_1, \ldots, d_m\}, y : \{r_1, \ldots, r_n\}.\, \mathit{newDish}(x, y)$

Here we are particularly interested in scenarios where agent and environment do not act in turns, as is often assumed, but where they more realistically may act in arbitrary order, similar to *supervisory control* [14]. In this setting, program realization becomes a synthesis task. The goal is to determine a policy that executes the program, while also satisfying a temporal goal, independent of and reacting to all possible environment behaviors. Temporal formulas are expressed in terms of $\mathrm{LTL_f}$, a restriction of Linear Temporal Logic (LTL) to finite traces [6]. For example, we may want to require that eventually ($\mathcal{F}$) there will always ($\mathcal{G}$) be no more dirty dish:

$$\mathcal{F}\,\mathcal{G}\,\neg\exists x, y.\, \mathit{DirtyDish}(x, y) \tag{1}$$

The Golog language provides a large degree of expressiveness, in particular in terms of first-order quantification, allowing to represent infinite-state systems over unbounded domains. The synthesis problem is thus highly undecidable in general. In this work, we present a decidable approach that works on a non-trivial fragment. Specifically, exploiting results on decidable verification of Golog [17], we require that

- all non-modal subformulas fall into the (decidable) two-variable fragment of first-order logic with counting quantifiers [7],
- the non-deterministic choice of action arguments π only ranges over finite sets, and
- dependencies among fluent predicates in successor state axioms satisfy a certain acyclicity criterion.

We can then construct an abstract, finite "game arena" (a special form of transition system) that captures all possible program executions while also tracking the satisfaction of the temporal specification. Using an encoding of $\mathrm{LTL_f}$ formulas that interprets temporal formulas as propositional atoms [12], the construction works on-the-fly and avoids building irrelevant parts.

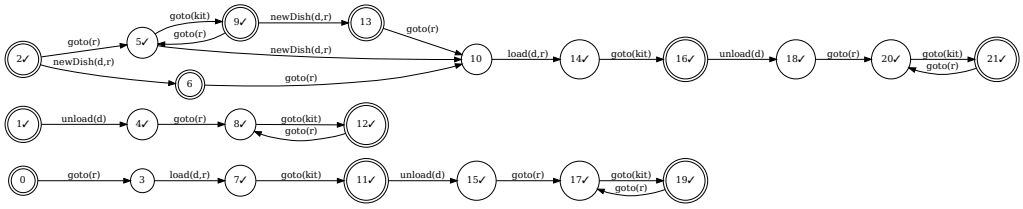


Figure 1: Example game arena with single room `r` and single dish `d`. Left are initial states with one new dish and none on robot (0), no new dish and one on robot (1), and no new dish and none on robot (2). Double circles indicate *final* states where program execution may terminate. Check marks indicate *accepting* states where the temporal goal is completed.

A game-theoretic approach can then be applied to synthesize a policy. Figure 1 shows an example game arena for the dishwasher robot if there is only one room r and one dish d, but where the initial state is underspecified so that the dish may initially be in the room, on the robot, or neither.

We implemented the method for the Golog interpreter `vergo` [1], which uses embedded theorem provers [16, 9] for first-order reasoning tasks and a first-order variant of binary decision diagrams for concisely representing formulas. We did an experimental evaluation on the dishwasher robot domain as well as a domain with a warehouse robot that moves boxes which may fall non-deterministically and break their contents. In the experiment, we varied the overall numbers of dishes, rooms, and boxes, and measured the method's runtime as well as the size of the resulting game arenas and extracted strategies. The set time-out of 1500 seconds was reached quickly for instances with 3 or more rooms, 3 or more dishes, and 3 or more boxes, yielding game arenas up to around 3000 states and transitions. Note that although decidable, the problem is very hard: In the worst case, the number of states in the abstract game arena is double exponential in the size of the input, and computing them involves consistency checks over sets of formulas that take up to double exponential time.

While the experiments thus demonstrate that the method works in principle, they also point out possible avenues of future work in terms of possible improvements. In particular, Golog programs often contain symmetries in the sense that there is a number of objects each of which need to be handled independently in the same way (e.g., the dishes in our example). For solving the task, the order of handling objects is hence irrelevant, yet the current method materializes all possible permutations, resulting in a severe blow-up of the size of the abstract transition system. It may therefore be interesting to study how the approach can be adapted to detect and deal with symmetries of this kind. Another limitation is the restriction that the π operator ranges over finite sets only. It can be shown that dropping this constraint altogether quickly results in undecidability, but the condition seems very harsh in light of the fact that non-finitary quantification is allowed in other places such as the action theory or the temporal goal. We are therefore interested in identifying perhaps "softer" restrictions that still guarantee decidability, yet allow for picking action arguments from potentially infinite sets.

References

[1] Jens Claßen. Symbolic verification of Golog programs with first-order BDDs. In Michael Thielscher, Francesca Toni, and Frank Wolter, editors, *Proceedings of the Sixteenth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2018)*, pages 524–529. AAAI Press, 2018.

[2] Jens Claßen and James P. Delgrande. An Account of Intensional and Extensional Actions, and its Application to Belief, Nondeterministic Actions and Fallible Sensors. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, volume 18, pages 194–204, September 2021.

[3] Jens Claßen, Martin Liebenberg, Gerhard Lakemeyer, and Benjamin Zarrieß. Exploring the boundaries of decidable verification of non-terminating Golog programs. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1012–1019. AAAI Press, 2014.

[4] Giuseppe De Giacomo and Yves Lespérance. The nondeterministic situation calculus. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, volume 18, pages 216–226. AAAI Press, September 2021.

[5] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121:109–169, 2000.

[6] Giuseppe De Giacomo and Moshe Y. Vardi. Synthesis for LTL and LDL on Finite Traces. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1558–1564. AAAI Press, 2015.

[7] Erich Grädel, M. Otto, and E. Rosen. Two-variable logic with counting is decidable. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 306–317, June 1997.

[8] Till Hofmann and Jens Claßen. LTLf synthesis on first-order action theories, 2024. arXiv:2410.00726.

[9] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *Proceedings of the Twentyfifth International Conference on Computer Aided Verification (CAV 2013)*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[10] Gerhard Lakemeyer and Hector J. Levesque. A semantic characterization of a useful fragment of the situation calculus with knowledge. *Artificial Intelligence*, 175(1):142–164, 2010.

[11] Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

[12] Jianwen Li, Geguang Pu, Yueling Zhang, Moshe Y. Vardi, and Kristin Y. Rozier. SAT-based explicit LTLf satisfiability checking. *Artificial Intelligence*, 289:103369, December 2020.

[13] John McCarthy and Patrick J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[14] P.J.G. Ramadge and W.M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, January 1989.

[15] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[16] Stephan Schulz, Simon Cruanes, and Petar Vukmirovic. Faster, higher, stronger: E 2.3. In Pascal Fontaine, editor, *Proceedings of the Twenty-Seventh International Conference on Automated Deduction (CADE 2019)*, volume 11716 of *Lecture Notes in Computer Science*, pages 495–507. Springer, 2019.

[17] Benjamin Zarrieß and Jens Claßen. Decidable verification of Golog programs over non-local effect actions. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1109–1115. AAAI Press, 2016.

Approximate Relational Reasoning for Higher-Order Probabilistic Programs

Philipp G. Haselwarter[1], Kwing Hei Li[1], Alejandro Aguirre[1], Simon Oddershede Gregersen[2], Joseph Tassarotti[2], and Lars Birkedal[1]

[1] Aarhus University, Denmark
[2] New York University, USA

Abstract

Properties such as provable security and correctness for randomized programs are naturally expressed relationally as approximate equivalences. As a result, a number of relational program logics have been developed to reason about such approximate equivalences of probabilistic programs. However, existing approximate relational logics are mostly restricted to first-order programs without general state.

In this paper we develop Approxis, a *higher-order approximate relational separation logic* for reasoning about approximate equivalence of programs written in an expressive ML-like language with discrete probabilistic sampling, higher-order functions, and higher-order state. The Approxis logic recasts the concept of *error credits* in the relational setting to reason about relational approximation, which allows for expressive notions of modularity and composition, a range of new approximate relational rules, and an internalization of a standard limiting argument for showing exact probabilistic equivalences by approximation. We also use Approxis to develop a logical relation model that quantifies over error credits, which can be used to prove *exact contextual equivalence*. We demonstrate the flexibility of our approach on a range of examples, including the PRP/PRF switching lemma, IND$-CPA security of an encryption scheme, and a collection of rejection samplers. All of the results have been mechanized in the Coq proof assistant and the Iris separation logic framework.

1 Introduction

Many important properties of probabilistic programs are naturally expressed as *approximate* equivalence of two programs. For example, provable security [GM84] compares an implementation of a cryptographic scheme to an idealized specification program that does not have access to any sensitive information, and aims to show that an adversary can only distinguish them with some small probability. In a similar spirit, many randomized algorithms and data structures can be specified by showing that they are approximately equivalent to their non-probabilistic counterparts. Consequently, it is important to be able to reason about approximate equivalences and so a number of relational program logics have been developed for first-order languages [BKOZB12, BGG+16, BEG+17] or higher-order languages with first-order global state [ABG+21].

In this work, we develop Approxis, a *higher-order approximate relational separation logic* for reasoning about approximate equivalence of RandML programs, an expressive ML-like language with discrete random sampling, higher-order functions, and higher-order dynamically-allocated state. A key point is that Approxis, inspired by the unary Eris logic [AHdM+24], introduces *error credits* in the relational setting to reason about approximation. Error credits are separation-logic resources that bound the maximum approximation error between two programs. We introduce a collection of novel *approximate coupling rules*, which consume error

credits in order to relate randomized transitions of two programs. By treating the relational approximation error as just another separation-logic resource, Approxis provides modular reasoning principles that enable more precise error accounting when composing proofs, much as Eris demonstrated in the non-relational setting.

Surprisingly, error credits not only allow us to prove approximate equivalences, they also allow us to prove *exact* equivalences that were beyond the scope of prior coupling-based relational program logics. Just as in real analysis, where one can prove two numbers are equal by showing that the distance between them is smaller than ε for all $\varepsilon > 0$, we can similarly show two probability distributions are equivalent by showing the distance between them is bounded by ε for all $\varepsilon > 0$. Using Approxis, we show how to recover this technique internally in the logic through *error amplification* [AHdM+24] and thus prove exact equivalence of probabilistic programs by means of approximation. Based on this, we develop a new binary logical relations model of a rich type system for RandML with recursive types and impredicative polymorphism. The model supports approximate reasoning and gives us a powerful and novel method for showing exact *contextual* equivalence of higher-order probabilistic programs. For other existing approaches, including both operational approaches, *e.g.*, Clutch [GAH+24], and denotational approaches, *e.g.*, pRHL [BGZB09] and HO-RHL [ABG+21], some of the examples that we consider would be very complicated—if not impossible—to handle.

We show that Approxis scales to more involved approximate reasoning by showing the classical PRP/PRF Switching Lemma [HWKS98, BR04] and IND$-CPA security of a PRF-based symmetric encryption scheme. Moreover, we apply error amplification and our logical relation to show contextual equivalences for a collection of rejection samplers, including a sampling scheme for drawing a random sample from a B+ tree [BM72].

Examples like the PRP/PRF Switching Lemma have been verified in many different settings, but we emphasize the rich programming language we consider here. While some of these examples might be expressible in simpler languages, features such as higher-order functions, higher-order state, and polymorphism are all found in general-purpose programming languages, and are needed for modern compositional software development. Moreover, cryptographic security can be more naturally expressed in such higher-order languages and avoids the need for syntactic restrictions on adversaries as seen, *e.g.*, in EasyCrypt [BDG+14]. As a consequence, verification frameworks must handle these language features to reason about large applications and realistic implementations. Higher-order separation logic is a powerful and well-tested abstraction for this purpose, and Approxis shows how to beneficially apply it for approximate relational reasoning. While the B+ tree case study, for example, is quite involved, the complexity is managed through mostly-standard separation-logic reasoning. We see this as a significant strength of our approach.

At a technical level, our development builds upon the (non-approximate) probabilistic coupling logic Clutch [GAH+24]. By incorporating error credits [AHdM+24] in the relational setting, our development generalizes the approach to approximate reasoning using approximate couplings. In addition, we introduce two new *coupling precondition* connectives and a notion of *erasability*. The erasability condition not only captures the soundness of asynchronous couplings [GAH+24] in a more semantic way, but also allows for a more principled approach to validating the new approximate and non-approximate coupling rules we introduce and which are critical for the examples that we consider.

Contributions In summary, we make the following contributions:

- The first higher-order approximate relational separation logic, Approxis, for reasoning about approximate equivalence of RandML programs, an expressive ML-like language

with probabilistic sampling, higher-order functions, and higher-order state,

- A logical internalization of a limiting argument that allows us to show exact equivalence of higher-order probabilistic programs through approximation,
- A class of new approximate and non-approximate coupling rules, including the *many-to-one* and *fragmented* coupling rules,
- A logical relations model of an expressive type system for RandML with recursive types and impredicative polymorphism, which allows us to show (exact) *contextual* equivalence of probabilistic programs through a limiting argument,
- A collection of case studies: the PRP/PRF Switching Lemma [HWKS98, BR04], IND$-CPA security of a PRF-based symmetric encryption scheme, and contextual equivalence of a selection of rejection samplers, including a sampling scheme for drawing a random sample from a B+ tree [BM72]. Several of these are, to the best of our knowledge, beyond the scope of previous techniques, in particular for expressive languages such as RandML.
- Full mechanization of all results in the Coq proof assistant [Tea24], building on top of the Iris separation logic framework [JKJ+18] and the Coquelicot [BLM15] library for real analysis.

Note: A full preprint of this work is available [HLA+24].

Acknowledgments This work was supported in part by the National Science Foundation, grant no. 2338317, the Carlsberg Foundation, grant no. CF23-0791, a Villum Investigator grant, no. 25804, Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, and the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[ABG+21] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, Shin-ya Katsumata, and Tetsuya Sato. Higher-order probabilistic adversarial computations: categorical semantics and program logics. *Proc. ACM Program. Lang.*, 5(ICFP), aug 2021.

[AHdM+24] Alejandro Aguirre, Philipp G. Haselwarter, Markus de Medeiros, Kwing Hei Li, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Error credits: Resourceful reasoning about error bounds for higher-order probabilistic programs. *Proc. ACM Program. Lang.*, 8(ICFP), aug 2024.

[BDG+14] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. *EasyCrypt: A Tutorial*, pages 146–166. Springer International Publishing, Cham, 2014.

[BEG+17] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving expected sensitivity of probabilistic programs. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[BGG+16] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, page 749–758, New York, NY, USA, 2016. Association for Computing Machinery.

[BGZB09] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.*, 44(1):90–101, jan 2009.

[BKOZB12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. *SIGPLAN Not.*, 47(1):97–110, jan 2012.

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Math. Comput. Sci.*, 9(1):41–62, 2015.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[BR04] Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. *IACR Cryptol. ePrint Arch.*, page 331, 2004.

[GAH+24] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. Asynchronous probabilistic couplings in higher-order separation logic. *Proc. ACM Program. Lang.*, 8(POPL):753–784, 2024.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.

[HLA+24] Philipp G. Haselwarter, Kwing Hei Li, Alejandro Aguirre, Simon Oddershede Gregersen, Joseph Tassarotti, and Lars Birkedal. Approximate relational reasoning for higher-order probabilistic programs, 2024.

[HWKS98] Chris Hall, David A. Wagner, John Kelsey, and Bruce Schneier. Building prfs from prps. In *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, pages 370–389, 1998.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018.

[Tea24] The Coq Development Team. The Coq Proof Assistant, June 2024.

Probably: A language with stochastic let-bindings

Stian Øverby[1] and Joachim Tilsted Kristensen[2]

[1] Department of Informatics, University of Oslo Oslo, Norway stiaov@ifi.uio.no
[2] Department of Informatics, University of Oslo Oslo, Norway joachkr@ifi.uio.no

Abstract

Probabilistic programming languages can be used for specifying experiments as computer programs. We address the problem of computing the underlying probability distribution of a small toy programming language with stochastic let bindings. Our main contribution is to equip the language with probability distribution in both terms and type system, so that we can reason about probabilistic computations.

Contents

1 Introduction

Rather than binding a specific value, discrete stochastic variables bind the outcome of *observing* an experiment, such as flipping a coin or measuring the number of packets dropped in a network.

In the context of this paper, we use the word *experiment* to mean any non-deterministic program that has a well-defined set of possible outcomes, known as the sample space. In that sense, executing a program corresponds to performing an experiment.

During such experiments, we are interested in the probability of an *event* (a subset of all possible outcomes) occurring. It is generally assumed that events come from a distribution, that is, a pair $(\mathcal{S}, \mathcal{P})$ consisting of a sample space $\mathcal{S}$ and a function $\mathcal{P} : \mathcal{S} \to [0, 1]$ that assigns probabilities to its members. We say that an event occurs if the outcome of an experiment is a member of that event [1].

A probabilistic programming language, is a language in which programs express experiments. Meaning that they have stochastic bindings and a way of *observing* their outcomes. The experiments one can express in such a language implicitly specify a sample space $\mathcal{S}'$, and the problem of inference is to decide a function $\mathcal{P}'$ that describe the distribution $(\mathcal{S}', \mathcal{P}')$.

In this work, we want to allow exact inference through a distribution decorated type system that the programmer can query. And, we want to provide the guarantee, that sampling (running the experiment/program) yields values according to the inferred distribution.

By decorating each term with distributions, inference can be done by defining what it means to combine the distribution of each term. We explore this approach with the following contributions:

- We give the syntax of a probabilistic programming language `Probably` (see section 2), and its sampling and semantics (what it means to perform an experiment).

- We suggest a type system decorated with distributions, that allow exact inference for `Probably` (see section 2.2).

2 `Probably` : The language stuff

A program in `Probably` is the specification of an experiment. Running the program corresponds to conducting that experiment, which yields the outcome we observe. The syntax for programs is similar to the REC language from FSoPL [7]. That is a program is a term, and the syntax for terms can be found in Figure 2. A term can be a number $\overline{n}$, a boolean value, a variable, an operation on terms, a function, a function application or a conditional. Finally, a term can be a stochastic let binding.

The stochastic let binding introduces non-determinism by binding outcomes of sampling a uniform distribution over some type $\vec{\tau}$. Furthermore, since the uniform distribution is not well-defined on all domains the type is restricted to a size type $\vec{\tau}$, which we will detail in Section 2.2.1. Inference of probability distributions happens through a combination of type inference, and syntactic composition of terms to determine some probability function.

`Probably` is deliberately a minimal language. Some terms and types present in general purpose languages are not included. A small language allows us to clearly focus our attention to the interesting ideas of how to reason about experiments, and makes it easier to deal with proofs. Our language is primarily created to prove a point, and not as a industry tool.

`Probably` does not have a recursive construct at this point. We plan to support recursion in later iterations of the language, but doing so require us to solve inference problems related to unbounded recursion that will be worthy of its own paper.

Logical `and`(&&), `or`(||) and `equality`(==) are syntactic sugar for some combinations of conditionals. We want to keep the language simple to make proving things about our programs easier in the future.

$$d ::= \textbf{uniform}\ \vec{\tau} \qquad \text{(uniform dist over size type)}$$

Figure 1: Syntax of distributions

$$
\begin{array}{rr}
t ::= \overline{n} & \text{(num literal)} \\
\mid \texttt{true} & \text{(bool literal)} \\
\mid \texttt{false} & \text{(bool literal)} \\
\mid \texttt{x} & \text{(var)} \\
\mid t_0 + t_1 & \text{(addition expr)} \\
\mid t_0 \leq t_1 & \text{(leq expr)} \\
\mid \lambda x.t_0 & \text{(lambda expr)} \\
\mid t_0 t_1 & \text{(function application)} \\
\mid \texttt{if } t_0 \texttt{ then } t_1 \texttt{ else } t_2 & \text{(conditional expr)} \\
\mid \texttt{let x} \sim d \texttt{ in } t_0 & \text{(stochastic let expr)}
\end{array}
$$

Figure 2: syntax of terms

$$
\begin{aligned}
[\![\ t_1 \ \&\&\ t_2\ ]\!] &::= \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } \textbf{false} \\
[\![\ t_1 \parallel t_2\ ]\!] &::= \texttt{if } t_1 \texttt{ then } \textbf{true} \texttt{ else } t_2 \\
[\![\ t_1 == t_2\ ]\!] &::= t_1 \leq t_2 \ \&\&\ t_2 \leq t_1
\end{aligned}
$$

Figure 3: syntactic sugar

2.1 `Probably` by example

Let us try to model the behavior of `insert` of a hash map. Since the elements are arbitrary, and we assume we have a good hash function that evenly distributes the elements, we can represent the resulting hash as picking an element from a uniform distribution of 15 unique elements.

```
let x ~ (uniform int 1 15) in
  let y ~ (uniform int 1 15) in
    x == y
λ> false
```

Listing 1: Example program 1

In example program 1, the experiment yields the outcome `false` since the two hashes were different. By repeating the experiment, we could get the same result, but at some point it should produce `true`. But how likely is our program to produce `true`? We certainty prefer that the probability of collisions of our hashing function to be reasonably low, so that we could avoid handling collisions as often as possible. Luckily for us, `Probably` provides a way of answering these type of questions.

2.2 Formalization of `Probably`

Every term is decorated with a *distribution type*. A distribution type $\hat{\tau}$ is a tuple $(\vec{\tau}, \hat{p})$ of a size type $\vec{\tau}$ and a collection of probability functions $\hat{p}$ that assign probabilities to specialized forms of events. It is called a distribution type because the two components describe a distribution, well enough to answer certain kinds of queries. $\vec{\tau}$ represents the possible outcomes of a given experiment, while $\hat{p}$ provide answers to various simple queries, such as "what is the probability of the outcome being less than 7", and the possible queries are those that can be achieved through the composition of such.

Size types have judgments rules (section 2.2.2), while probability functions have denotational semantics (section 2.2.3).

$$
\begin{aligned}
\hat{\tau} &::= \vec{\tau}, \hat{p} \\
\vec{\tau} &::= \textbf{int}\ \mathtt{n}\ \mathtt{m} \\
&\mid \textbf{bool} \\
&\mid \vec{\tau}_0 \to \vec{\tau}_1
\end{aligned}
$$

Figure 4: Syntax of distribution types ($\hat{\tau}$) and size types ($\vec{\tau}$)

2.2.1 Size types

A size type $\vec{\tau}$ is a simple type τ confided by a size. The syntax of size types can be found in figure 4. A size type can either be an integer with an inclusive lower and upper bound, a boolean, or a function type that maps a size type to any other size type. The size of an integer type is 1 more than its upper bound subtracted by its lower bound, the size of the boolean type is 2, and the size of a function type $\vec{\tau}_1 \to \vec{\tau}_2$ is $|\vec{\tau}_2|^{|\vec{\tau}_1|}$.

Size types completely avoid the issue of attempting to uniformly sample from an infinite sample space. As an example, let us try to sample a value from the uniform distribution of integers. The probability of sampling a number is equal to the cardinality of the occurrences of the number, divided by the cardinality of the type. Since the cardinality of the integer type is ∞, the probability of sampling any number will always be 0.

2.2.2 Judgement rules of size types

Judgment rules decides types of terms, and therefore their sample space. The judgment rules are similar to traditional functional languages, with the main difference being that they consider the size of the types.

All judgments for typing atomic terms with size types is an adaptation of the judgement rules for simple types. The only difference, is that T-NUM sets a lower and upper bound equal to its own value. The rules for functions, function application, $\leq$-expresssion and let-expressions are more or less equivalent to that of their simple type counter-part.

The judgement rule for addition combines the bounds of two sized integers in a clever way. The new lower bound is equal to the sum of the two smallest numbers of each sized integer, which corresponds to their lower bound. Equivalently, the new upper bound is the sum of the two upper bounds.

The judgment rule of conditionals depends on the probability function of its predicate. If the predicate is decorated with a boolean distribution that is certain to pick a specific truth value, the sample space can be reduced to that of the branch that will be chosen with probability 1. Otherwise, the domain is the union of the sample space of both branches.

Notice that sized integers are only considered to be of same type if they share the exact same size. This is not very useful, and we want to introduce subtyping in future work. Subtyping is not further discussed in this extended abstract.

$$\text{T-NUM}: \frac{}{\Gamma \vdash \overline{n} : \textbf{int}\ n\ n}$$

$$\text{T-TRUE}: \frac{}{\Gamma \vdash \textbf{true} : \textbf{bool}}$$

$$\text{T-FALSE}: \frac{}{\Gamma \vdash \textbf{false} : \textbf{bool}}$$

$$\text{T-VAR}: \frac{}{\Gamma \vdash x : \vec{\tau}} \Gamma(x) = \vec{\tau}$$

$$\text{T-PLUS}: \frac{\Gamma \vdash t_0 : \textbf{int}\ n_1\ m_1 \quad \Gamma \vdash t_1 : \textbf{int}\ n_2\ m_2}{\Gamma \vdash t_0 + t_1 : \textbf{int}\ (n_1 + n_2)\ (m_1 + m_2)}$$

$$\text{T-LEQ}: \frac{\Gamma \vdash t_0 : \textbf{int}\ n_1\ m_1 \quad \Gamma \vdash t_1 : \textbf{int}\ n_2\ m_2}{\Gamma \vdash t_0 \leq t_1 : \textbf{bool}}$$

$$\text{T-IF-NUM}: \frac{\Gamma \vdash t_0 : \textbf{bool} \quad \Gamma \vdash t_1 : \textbf{int}\ n_1\ m_1 \quad \Gamma \vdash t_2 : \textbf{int}\ n_2\ m_2}{\Gamma \vdash\ \textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : \vec{\tau}}$$

$$\vec{\tau} = \begin{cases} \textbf{int}\ n_1\ m_1 & [\![\ t_0\ ]\!]_{bool}(true\ ) == 1 \\ \textbf{int}\ n_2\ m_2 & [\![\ t_0\ ]\!]_{bool}(false) == 1 \\ \textbf{int}\ min(n_1, n_2)\ max(m_1, m_2) & \text{otherwise} \end{cases}$$

$$\text{T-IF-BOOL}: \frac{\Gamma \vdash t_0 : \textbf{bool} \quad \Gamma \vdash t_1 : \textbf{bool} \quad \Gamma \vdash t_2 : \textbf{bool}}{\Gamma \vdash\ \textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : \textbf{bool}}$$

$$\text{T-IF-LAM}: \frac{\Gamma \vdash t_0 : \textbf{bool} \quad \Gamma \vdash t_1 : \vec{\tau}_1 \to \vec{\tau}_2 \quad \Gamma \vdash t_2 : \vec{\tau}_1 \to \vec{\tau}_2}{\Gamma \vdash\ \textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 : \vec{\tau}_1 \to \vec{\tau}_2}$$

$$\text{T-LET}: \frac{\Gamma[x \to \vec{\tau}_1] \vdash t : \vec{\tau}_2}{\Gamma \vdash \textbf{let}\ x \sim \textbf{uniform}\ \vec{\tau}_1\ \textbf{in}\ t : \vec{\tau}_2}$$

$$\text{T-LAM}: \frac{\Gamma[x \to \vec{\tau}_1] \vdash t : \vec{\tau}_2}{\Gamma \vdash \lambda x.t : \vec{\tau}_1 \to \vec{\tau}_2}$$

$$\text{T-APP}: \frac{\Gamma \vdash t_1 : \vec{\tau}_1 \to \vec{\tau}_2 \quad \Gamma \vdash t_2 : \vec{\tau}_1}{\Gamma \vdash t_1 t_2 : \vec{\tau}_2}$$

Figure 5: Typing rules

2.2.3 Denotional semantics of probability functions

Denotational semantics allow us to specify how to derive the probability functions of our program. Every atomic term (numbers and booleans) has a predefined probability function, while composed terms has rules for how to compose their own probability functions from their subterms.

We define a function $[\![\cdot]\!]_{\vec{\tau}}$ by induction on the syntax of our programming language. This particular function is the "equals function", a function used to answer what the probability of a value being equal to the outcome of an experiment. Be aware that some of the queries have large overhead, in particular $\leq$. Assume an arithmetic expression $t_1 + t_2$ where the sub-expressions has type $\vec{\tau}_1$ and $\vec{\tau}_2$ respectively. The query requires every member of $\vec{\tau}_1$ to be paired and compared with every member of $\vec{\tau}_2$, resulting in a complexity of $\mathcal{O}(|\vec{\tau}_1| \cdot |\vec{\tau}_2|)$. Sadly, with large types this might be incredibly slow.

In the future, we want to define a collection of functions that are all optimized to answer specific queries and avoiding expensive operations. By defining such functions, we can compose these functions to answer more complex queries.

$$
\begin{aligned}
[\![v]\!]_{\vec{\tau}}(w) &= \begin{cases} 1 & w = v \\ 0 & \text{otherwise} \end{cases} \\
[\![x]\!]_{\vec{\tau}}(v) &= \begin{cases} \frac{1}{|\vec{\tau}|} & v \in E(\vec{\tau}) \\ 0 & \text{otherwise} \end{cases} \\
[\![t_1 + t_2]\!]_{\text{int } (m_1+n_1)\ (m_2+n_2)}(k) &= \sum_{i=n}^{m} [\![t_1]\!]_{\text{int } m_1\ m_2}(x-i) \cdot [\![t_2]\!]_{\text{int } n_1\ n_2}(i) \\
[\![t_1 \leq t_2]\!]_{\text{bool}}(true) &= \sum_{i \leq j \wedge \langle i,j \rangle \in E(\vec{\tau}_1) \times E(\vec{\tau}_2)} [\![t_1]\!]_{\vec{\tau}_1}(i) \cdot [\![t_2]\!]_{\vec{\tau}_2}(j) \\
[\![t_1 \leq t_2]\!]_{\text{bool}}(false) &= 1 - [\![t_1 \leq t_2]\!]_{\text{bool}}(true) \\
[\![\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3]\!]_{\vec{\tau}}(x) &= [\![t_1]\!]_{bool}(\texttt{true}) \cdot [\![t_2]\!]_{\vec{\tau}}(x) + [\![t_1]\!]_{bool}(\texttt{false}) \cdot [\![t_3]\!]_{\vec{\tau}}(x) \\
[\![\lambda x.t_1]\!]_{\vec{\tau}_1 \to \vec{\tau}_2}(\lambda y.t_2) &= \begin{cases} 1 & \bigwedge_{t \in E(\vec{\tau}_1)} t_1[x/t_1] = t_2[y/t_1] \\ 0 & \text{otherwise} \end{cases} \\
[\![(\lambda x.t_1)\ t_2]\!]_{\vec{\tau}}(v) &= [\![t_1[x/t_2]]\!]_{\vec{\tau}}(v) \\
[\![\texttt{Let } x \sim \texttt{uniform } \vec{\tau}_1 \texttt{ in } t]\!]_{\vec{\tau}_2}(v) &= [\![t]\!]_{\vec{\tau}_2}(v) \\
\text{Helper function:}^* E(\vec{\tau}) &= \{t | t \in \vec{\tau}\}
\end{aligned}
$$

Figure 6: Denotational semantics of probability function

3 The path forward

In conclusion, `Probably` is a probabilistic programming language designed to model and do experiments. By supporting stochastic let-bindings and having a type system decorated with distributions, it should allow for precise reasoning about the probabilistic behavior of experiments. Our current work paves the way for further exploration of what it means to have a language that deals with stochastic variables, and how to design languages that support them. For a full paper we want to explore optimized probability functions and subtyping. Hopefully, `Probably` will enable us to get greater insight into modeling complex stochastic computations.

4 Background and related work

A probability distribution is a mathematical function **P** that distributes a total probability of 1 among each member ω of a probability space Ω. The probability **P** (ω) must be a nonnegative real number, and the condition

$$\sum_{\omega \in \Omega} P(\omega) = 1 \tag{1}$$

must hold for every discrete probability space [3].

Probabilistic programming is about doing Bayesian inference using the tools of computer scientists: denoting probabilistic models in a programming language, and using statistical inference algorithms for computing the true posterior distribution of observed program output [6].

A probabilistic programming language should assist encoding complex probability distributions by writing computer programs. This is done with a combination of familiar programming language constructs, like conditionals and loops, and some additional ones making it easier to work with probability.

Probability distributions form a monad [2], allowing for a simple model for composing distributions. A distribution can be composed with another using the bind operator, and interesting queries can be answered through the distribution. Interesting queries might be the probability of a term yielding a specific value when evaluated, or even the expected value [5].

`Probably` can be viewed as a generalization of `Troll` [4], a programming language for specifying dice-rolls. `Troll` has stochastic let bindings that bind variables to the outcome of rolling an n sided die. The language has semantics for both rolling dice, and inspecting the probability distribution of a specified dice roll. The distributions are represented as finite maps. An eagerly evaluated map of all possible outcomes to their probability does not scale well to larger domains than dice. However, `Probably` uses a similar underlying map structure as `Troll`, with some additional information to avoid evaluating the complete map.

References

[1] Joseph K. Blitzstein and Jessica Hwang. *Introduction to Probability*. CRC Press, 2015.

[2] Martin Erwig and Steve Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *J. Funct. Program.*, 16(1):21–34, January 2006.

[3] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Massachusetts, 2008.

[4] Torben Ægidius Mogensen. Troll, a language for specifying dice-rolls. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 1910–1915, New York, NY, USA, 2009. Association for Computing Machinery.

[5] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. *SIGPLAN Not.*, 37(1):154–165, January 2002.

[6] Jan-Willem van de Meent, Brooks Page, Hongseok Yang, and Frank Wood. *An Introduction to Probabilistic Programming*. 2021.

[7] Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, London, 1994.

Modelling a Probabilistic Programming Language in Clocked Cubical Type Theory *

Philipp Jan Andries Stassen[1], Rasmus Ejlers Møgelberg[2], Maaike Zwart[3], Alejandro Aguirre[4], and Lars Birkedal[5]

[1] Aarhus University, Aarhus, Denmark
stassen@cs.au.dk
[2] IT University of Copenhagen, Copenhagen, Denmark
mogel@itu.dk
[3] IT University of Copenhagen, Copenhagen, Denmark
mazw@itu.dk
[4] Aarhus University, Aarhus, Denmark
alejandro@cs.au.dk
[5] Aarhus University, Aarhus, Denmark
birkedal@cs.au.dk

Abstract

We show how to use a metalanguage with guarded recursion to construct both denotational and operational semantics for a programming language combining recursive types and finite probabilistic choice. We construct a relation between the two and show that it is adequate for reasoning about contextual equivalence. Examples include the encoding of a fair coin from an unfair one, and the equivalence of two random walks.

1 Introduction

Contextual equivalence is often the right notion of equivalence for programs, but can be hard to reason about. Various techniques have been constructed for this purpose, including denotational semantics, models based on operational semantics, and bisimulation techniques. When the object language has recursion, the traditional approach is to use domain theory, but this can be quite challenging when the language also has computational effects such as probabilistic choice or advanced notions of store. More recently, step-indexed techniques have been applied especially to operational models. However, the explicit steps often obscure the picture, and the appealing simple idea of denotational semantics, that types are just sets and terms just functions, appears to be lost.

2 Guarded Recursion

Guarded recursion is an abstract approach to step-indexing. The idea is to work in a metalanguage capturing the essence of the step-indexed models using a modal type constructor $\triangleright$ (pronounced 'later') to encode a notion of step, a delay operation $\mathsf{next} : A \to \triangleright A$, a fixed point

*This work was supported in part by the Independent Research Fund Denmark grant number 2032-00134B, in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, and in part by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

operator $\mathsf{fix} : (\triangleright A \to A) \to A$, as well as guarded recursive types, i.e., solutions to equations such as $X \simeq A + \triangleright X$, where the recursive type variable only appears under a $\triangleright$. These construction can be justified by a semantic model like the topos of trees [1], but the metalanguage hides the intricacies of the model construction from the user, revealing a simple language for the modelling task at hand.

In this talk, the particular meta-language we use is Clocked Cubical Type Theory (CCTT) [12]. This is a dependent type theory combining features of multiclocked guarded recursion with Cubical Type Theory [4], which has been implemented in an experimental version of Cubical Agda[1], although the results presented here have not been formalised in that. The word *multiclocked* means that modality $\triangleright^\kappa$ is indexed by a clock κ, which can be quantified over to encode coinductive types. For example, one can define the *guarded delay monad* $L^\kappa(-)$ by defining $L^\kappa A$ to be the unique solution to the equation $L^\kappa A \simeq A + \triangleright^\kappa(L^\kappa A)$. The clock κ can then be universally quantified to yield the *delay monad* $L^\forall A \triangleq \forall\kappa.L^\kappa A$, which can be proved to be the coinductive solution to

$$L^\forall A \simeq A + L^\forall A \tag{1}$$

The delay monad $L^\forall$ has previously been used to model recursion in type theory [3], but unlike $L^\forall$, the guarded version allows for the definition of a fixed point operator of type

$$((A \to L^\kappa B) \to (A \to L^\kappa B)) \to A \to L^\kappa B.$$

This can be used, e.g., to give very simple denotational semantics to programming languages with recursive terms, in which the function type is interpreted using the standard (call-by-value) monadic interpretation: $[\![\sigma \to \tau]\!]^\kappa \triangleq [\![\sigma]\!]^\kappa \to L^\kappa[\![\tau]\!]^\kappa$. Quantification over clocks allows for programs to be run without delays in the form of $\triangleright^\kappa$. For example, a closed program of type Nat is interpreted as an element of type $L^\kappa(\mathbb{N})$, for any κ. By quantifying over κ, one obtains an element of $L^\forall\mathbb{N}$, which can then be run by the term

$$\mathsf{run} : \mathbb{N} \to L^\forall A \to A + 1$$

which unfolds its second argument using (1) the number of times given by the first argument. Similarly, one can define an operational semantics as functions of type

$$\mathsf{eval}^\kappa : \{\sigma : \mathsf{Ty}\} \to \mathsf{Tm}_\sigma \to L^\kappa(\mathsf{Val}_\sigma)$$
$$\mathsf{eval} : \{\sigma : \mathsf{Ty}\} \to \mathsf{Tm}_\sigma \to L^\forall(\mathsf{Val}_\sigma)$$

where Tm_σ is the type of terms of σ, and Val_σ is the type of values of type σ. The second of these is defined simply by abstracting over a clock in the first: $\mathsf{eval}\, M \triangleq \Lambda\kappa.\mathsf{eval}^\kappa\, M$.

3 Modelling probabilistic FPC

In this talk we extend the above idea to model Probabilistic FPC, a call-by-value lambda calculus with recursive types and a binary probabilistic choice operator. In order to do so, let $\mathcal{D}$ be the finite distributions monad, which can be defined in CCTT as a higher inductive type using constructors for dirac distributions and convex combinations of distributions as well as paths for idempotency, commutativity and associativity of convex combinations. Define the *guarded convex delay monad* D^κ by the guarded type equation

$$\mathsf{D}^\kappa A \simeq \mathcal{D}(A + \triangleright^\kappa(\mathsf{D}^\kappa A)).$$

[1] https://github.com/agda/guarded

An element of $\mathsf{D}^\kappa A$ is a finite distribution of elements that are either values of type A, or computations that can run for one more step. For example, one can define a geometric distribution

$$\mathsf{geo}_p^\kappa : \mathbb{N} \to \mathsf{D}^\kappa \mathbb{N}$$
$$\mathsf{geo}_p^\kappa\, n \triangleq (\delta^\kappa n) \oplus_p \mathsf{step}^\kappa(\mathsf{next}^\kappa(\mathsf{geo}_p^\kappa\,(n+1)))$$

where $\delta^\kappa : A \to \mathsf{D}^\kappa A$ and $\mathsf{step}^\kappa : \triangleright^\kappa(\mathsf{D}^\kappa A) \to \mathsf{D}^\kappa A$ are the obvious inclusions using the dirac distributions, and $\oplus_p$ is the convex combination of distributions. The term geo_p^κ is a recursive term that can be defined using the fixed point operator fix. The type $\mathsf{D}^\forall A \triangleq \forall\kappa.\mathsf{D}^\kappa A$ is the coinductive solution to $\mathsf{D}^\forall A \simeq \mathcal{D}(A + \mathsf{D}^\forall A)$.

The denotational and operational semantics can then be defined by extending the above idea from using L^κ to D^κ. The interpretation of recursive types is by using guarded recursive types

$$[\![\mu X.\tau]\!]^\kappa \triangleq \triangleright^\kappa [\![\tau[\mu X.\tau/X]]\!]^\kappa$$

We prove an adequacy result by constructing relations between syntax and semantics

$$\preceq_\sigma^{\kappa,\mathsf{Val}} \;: [\![\sigma]\!]^\kappa \to \mathsf{Val}_\sigma \to \mathsf{Prop}$$
$$\preceq_\sigma^{\kappa,\mathsf{Tm}} \;: \mathsf{D}^\kappa [\![\sigma]\!]^\kappa \to \mathsf{D}^\forall(\mathsf{Val}_\sigma) \to \mathsf{Prop}$$

by induction on the type σ. In the case of recursive types, this relation is defined by guarded recursion. We show that this relation is adequate for reasoning about contextual refinement, in the sense that if the denotation of M is related to $\mathsf{eval}\, N$, then N contextually refines M. The relation includes weak bisimilarity, so can be used to prove equivalence of programs that compute the same result in a different number of steps. Such programs do not have equal denotations since the model counts steps using the $\triangleright$ modality.

We give examples of how to use the relation $\preceq_\sigma^{\kappa,\mathsf{Tm}}$ to prove equivalences of probabilistic programs, including the encoding of a fair coin using an unfair coin, and the equivalence of two random walks. All of these examples use guarded recursion as part of the reasoning.

4 Conclusion

Probabilistic programming languages have received much attention lately, and there are other works on denotational semantics [11, 15, 9, 8, 7], operational techniques [10, 2, 6, 16, 17], and bisimulations [5, 13] The contribution of the present work is to show how working at the high level of abstraction that the metalanguage provides, can make the construction of denotational semantics fairly easy, and allow the user to focus on the essentials, as illustrated by the relative simplicity of the examples.

This abstract is based on a recent preprint [14] available on the arXiv.

References

[1] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.*, 8(4), 2012.

[2] Ales Bizjak and Lars Birkedal. Step-indexed logical relations for probability. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2015.

[3] Venanzio Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2), 2005.

[4] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017.

[5] Raphaëlle Crubillé and Ugo Dal Lago. On probabilistic applicative bisimulation and call-by-value λ-calculi. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2014.

[6] Ryan Culpepper and Andrew Cobb. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 368–392. Springer, 2017.

[7] Fredrik Dahlqvist and Dexter Kozen. Semantics of higher-order probabilistic programs with conditioning. *Proc. ACM Program. Lang.*, 4(POPL):57:1–57:29, 2020.

[8] Thomas Ehrhard, Michele Pagani, and Christine Tasson. Full abstraction for probabilistic PCF. *J. ACM*, 65(4):23:1–23:44, 2018.

[9] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.

[10] Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, pages 209–218. IEEE Computer Society, 2010.

[11] C. Jones and Gordon D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 186–195. IEEE Computer Society, 1989.

[12] Magnus Baunsgaard Kristensen, Rasmus Ejlers Møgelberg, and Andrea Vezzosi. Greatest hits: Higher inductive types in coinductive definitions via induction under clocks. In Christel Baier and Dana Fisman, editors, *LICS '22: 37th Annual ACM/IEEE Symposium on Logic in Computer Science, Haifa, Israel, August 2 - 5, 2022*, pages 42:1–42:13. ACM, 2022.

[13] Ugo Dal Lago, Davide Sangiorgi, and Michele Alberti. On coinductive equivalences for higher-order probabilistic functional programs. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 297–308. ACM, 2014.

[14] Philipp Stassen, Rasmus Ejlers Møgelberg, Maaike Zwart, Alejandro Aguirre, and Lars Birkedal. Modelling recursion and probabilistic choice in guarded type theory. *CoRR*, abs/2408.04455, 2024.

[15] Matthijs Vákár, Ohad Kammar, and Sam Staton. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.*, 3(POPL):36:1–36:29, 2019.

[16] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. ACM Program. Lang.*, 2(ICFP):87:1–87:30, 2018.

[17] Yizhou Zhang and Nada Amin. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022.

A type system to Ensure Non-Interference in ReScript

Benjamin Clausen Bennetzen[1], Emilie Sonne Steinmann[2], Loke Walsted[3], Nikolaj Rossander Kristensen[4], Peter Buus Steffensen[5], and Daniel Vang Kleist[6]

Aalborg University, Aalborg, Nordjylland, Danmark
{bbenne20[1],estein19[2],lwalst20[3],nrkr20[4],psteff19[5],dkleis20[6]}@student.aau.dk

1 Introduction

The property of non-interference was defined as early as 1982 by Goguen and Meseguer [2], and can be analyzed through information flow analysis. In their work, they present a formal notion for describing security policies and models.

Ensuring non-interference is especially important in web-based services, as these can easily be accessed by unauthorized observers. The most commonly used language throughout the last couple of years for development is JavaScript [5, 6, 7], and thus several solutions have been developed for performing information flow analysis on JavaScript code, as seen in, e.g., the work of Just et al. from 2011 [4] and Hedin, Bello, and Sabelfeld's work from 2016 [3]. Both articles suggest mechanisms for dynamic (run-time) information flow analysis, though recognize that static (compile-time) analysis is required to analyze implicit information flows. Information flow is generally categorized into two types: explicit and implicit [3]. Explicit information flow covers situations, where data is copied directly, e.g., via an assignment. Implicit information flow is slightly more complicated, as it describes situations where information is used to control program flow. Here, information about confidential data is indirectly revealed, as confidential data controls public aspects of the program behavior. We denote confidential data as `h`, and public data as `l`, for the code examples.

```
1 let h = true
2 let l = if (h) { 2 } else { 3 }
```

Listing 1: Implicitly revealing the value of `h` through a conditional assignment to `l`.

Listing 1 showcases an example of information flow we want to avoid in order to achieve non-interference.

We will look at a more recently developed programming language called ReScript where a non-interferent information flow could be relevant.

ReScript is a functional programming language that compiles to JavaScript, and its developers claim ReScript to be the fastest build system on the web [9]. As the language is still quite new, information flow analysis for ReScript is yet to be developed.

An interesting aspect of ReScript is that some imperative features are included in the otherwise functional programming language. These complicate the analysis of non-interference, e.g., through the possibility to use `ref` to create mutable bindings. This allows for other ways to create information flow as seen in Listing 2 and Listing 3.

```
1 let h = ref(2)
2 let l = h
```

Listing 2: Leaking confidential information from variable `h` to public variable `l` by copying the reference to the location of the value of `h`.

```
1 let l = ref(2)
2 let h = l
3 h := 4
```

Listing 3: Leaking confidential data by copying the reference of a public variable, `l`, to a confidential variable, `h`, and then updating the value of `h`.

In both cases, the confidential and public variables end up pointing to the same memory, which means that the confidential information in `h` is accessible through the public variable `l`. This clearly breaks with non-interference. To deal with this, a solution must be able to control references.

In [8], Sabelfeld and Myers describe how a type system is a natural way to implement information flow analysis, as existing type systems can easily be extended with security levels. As with many properties, non-interference is an undecidable problem. Therefore, any type system aiming to capture non-interference will be an approximation.

We will introduce a type system for ReScript similar to the security type system in [8], with the aim of ensuring non-interference for a subset of ReScript. To prove that this type system ensures non-interference, we have developed a proof of soundness that shows that if an expression is typeable it is also non-interferent. This method of proving non-interference is similar to the method by Volpano et al. in [10].

2 The Information Flow Type-system

Type judgments are of the form $\Gamma, pc \vdash e : t_1 @ t_2 \rhd \Gamma'$, read as: Given a type environment Γ and a security level pc, then the expression e has the type t_1, with the lowest side effect being t_2, and produce a new type environment Γ'. Here, type environments are partial functions from variables to types ($\mathbf{Var} \rightharpoonup \mathbf{T}$). A type can be a security-level, that being either High or Low. Confidential data should be typed as High, and public data should be typed as Low. Additionally $pc \in \{\mathsf{Low}, \mathsf{High}\}$, $e \in \mathbf{Exp}$ and $t_1, t_2 \in \mathbf{T}$. pc represents the security level of the context, meaning that if $pc = \mathsf{High}$, then a High value controls the program flow, when e is evaluated.

An example of a type-rule is given by:

$$\text{(Let-n)} \quad \frac{\Gamma, pc \vdash e : t_2 @ t_3 \rhd \Gamma_1}{\Gamma, pc \vdash \mathsf{Let}\ x_{t_1} = \ e\ : \mathsf{Low} @ t_4 \rhd \Gamma[x \to t_1]} \quad \begin{array}{r} t_1, t_2 \in \{\mathsf{Low}, \mathsf{High}\} \\ t_1 \sqsupseteq t_2 \text{ and } t_1 \sqsupseteq pc \\ t_4 = \sqcap\{t_3, t_1\} \end{array}$$

Here, we ensure that the data that gets assigned to a variable x, must be either of a lower security level or of the same security level as the variable itself. This ensures that there is no explicit information flow, where highly secure data gets assigned to a variable used for low security data. For all type rules we refer to the full paper [1].

Our type system ensures that all well-typed expressions maintain non-interference.

Definition 2.4 (Non-interference). Consider $e \in \mathbf{Exp}$ and $\Gamma, \Gamma' \in$ TypeEnviroments. We say that e upholds the non-interference property under Γ if the following condition holds, for any given pair of states s_1, s_2.

if	$\Gamma \vdash s_1 =_{\mathsf{Low}} s_2$
and	$\langle e, s_1 \rangle \rightarrow \langle v_1, s_1' \rangle$
and	$\langle e, s_2 \rangle \rightarrow \langle v_2, s_2' \rangle$
then	$\Gamma' \vdash s_1' =_{\mathsf{Low}} s_2'$

The following definition precisely defines the meaning of being well-typed in the context of this paper:

Definition 2.5 (Well-typed). An expression e is well-typed in regards to a type environment Γ and a security level pc, if: $\Gamma, pc \vdash e : t_1 @ t_2 \rhd \Gamma'$

Our proof of soundness states that any given well-typed expression will maintain non-interference. This means, that if an expression can be typed with our type rules, there is no information flow from High variables to Low variables. For the full proof of every type-rule we refer to the full paper [1].

3 Conclusion

We have explored the integration of a type system, that ensures the non-interference property, within a subset of the programming language ReScript. A proof of soundness was used to validate that the type system ensures that every well-typed expression is non-interferent. As an addition, the type system has also been implemented as a type checker in Haskell. While the type system does enhance the security of programs, it also imposes limitations in the way you can use certain language constructs. An example of this is the inability of if-expressions to evaluate to anything other than Low or High.

As the type system only works for a subset of ReScript, the obvious extension would be a type system that encompasses all of ReScript. Most notably, we are missing records, lists, arrays, destructuring, switches and "try-catch" constructs. Additionally, there is a significant number of binary and unary operations currently missing, although their inclusion should be a relatively straightforward extension.

Building on this, another place that merits attention is the simplicity of our security levels. One could easily imagine scenarios where more complex security levels would be needed. Consider as an example the management of exam materials in a university. In such a scenario, students should not have access to exams before the scheduled date. Department heads should have access exclusively to their department's exams, while the university head should be able to access all exams. A very useful extension would therefore be one where the security levels are a lattice described by a set of formation rules, which allow for the vertical and lateral extension of the lattice.

References

[1] Benjamin Clausen Bennetzen, Daniel Vang Kleist, Emilie Sonne Steinmann, Loke Walsted, Nikolaj Rossander Kristensen, and Peter Buus Steffensen. A type system to ensure non-interference in rescript. 2023.

[2] J. A. Goguen and J. Meseguer. Security policies and security models. In 1982 IEEE Symposium on Security and Privacy, pages 11–11, 1982.

[3] Daniela Hedin, Lucianoa Bello, and Andreia Sabelfeld. Information-flow security for javascript and its apis. Journal of Computer Security, 24(2), 2016.

[4] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information flow analysis for javascript. In Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients, PLASTIC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[5] Stack Overflow. Developer survey 2020, 2020. Accessed: 2024-09-04.

[6] Stack Overflow. Developer survey 2021, 2021. Accessed: 2024-09-04.

[7] Stack Overflow. Developer survey 2023, 2023. Accessed: 2024-09-04.

[8] A. Sabelfeld and A.C. Myers. Language-based information-flow security. IEEE Journal on Selected Areas in Communications, 21(1):5–19, 2003.

[9] ReScript Team. Rescript, n.d. ReScript Documentation, Accessed: 2024-09-04.

[10] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. Journal of Computer Security, 4, 08 2000.

Towards text extraction learning with regular expressions extended with complement, intersection and lookarounds

Ksenija Kivojenko, Edwin Smagin, Ian Erik Varatalu, and Juhan Ernits

Tallinn University of Technology, Tallinn, Estonia
(kskivo|edsmag|ian.varatalu|juhan.ernits)@taltech.ee

1 Introduction

Learning regular expressions has various flavours. The task we focus on here is based on inference of regular expressions for text extraction by (Bartoli, Lorenzo, Medvet, & Tarlao, 2016) which is accompanied by a dataset with text to be extracted clearly annotated. We show how using a regular expression engine extended with intersection, complement and restricted lookarounds (Varatalu, Veanes, & Ernits, 2024) makes learning more efficient over traditional regex engines supporting concatenation, Kleene star and alteration. It is important to note that the aim is to use the functionality of regular expressions beyond deterministic finite automata, as there are ample examples of learning the latter. We are interested in learning regular expressions that support zero-width lookaround assertions (Goyvaerts, 2024) for context. An example of a lookaround for detecting author name in a bibtex entry might be `(?<=author.*).*` stating that the string `author` must precede the matched text on the current line.

We refer the reader to (Bartoli et al., 2016) for an overview of how the text extraction problem has evolved. We use the results from there as baseline. The regular expression evolution rules are replaced with heuristic search for regexes supporting intersection, complement and lookarounds interleaved with simplification and generalization steps.

For up-to-date context, inference of regular expressions has been used in the literature for different tasks, which we group into two distinct categories (omitting active learning approaches, which require user interaction and generation of regular expressions from natural language descriptions accompanied by samples):

1. Existence of a match (positive/negative examples);
2. Context-aware substring extraction (finding match positions with lookarounds).

In the current work we concentrate on algorithmic learning of regular expression as opposed to using large language models to generate candidate regular expressions.

1.1 Learning from positive and negative examples (*IsMatch*)

The first category of regular expression learning has received by far the most attention in the literature, with tools centered around finding only the existence of a match. AlphaRegex (Lee, So, & Oh, 2016) a tool with accompanying benchmark for synthesizing regular expressions from positive and negative examples.

RFixer (Pan, Hu, Xu, & D'Antoni, 2019), takes an approach where given a regular expression and positive and negative examples, it produces a fix as small distance from the original expression as possible which constitutes a repair.

(Chen, Wang, Ye, Durrett, & Dillig, 2020) propose a DSL for describing regular expressions. As the authors encode regular expression constraints into SMT context, they support and use

intersection and (approximated) complement, but there is no support for lookarounds for text extraction. The tool works on positive/negative examples given by the user.

(Kim, Hu, D'Antoni, & Reps, 2021) propose SemGuS, which can also be instantiated to synthesize regular expressions from positive and negative examples.

(Li, Xu, Cao, Chen, Ge, Cheung, & Zhao, 2020) FlashRegex was developed for ReDos regex generation. It is also based on learning from positive and negative examples by reducing the ambiguity of these regexes and using SAT techniques.

(Valizadeh, Gorinski, Iacobacci, & Berger, 2024) propose the Regular Expression Inference Challenge, a program optimization task of finding minimal regular expressions from examples, which can be very efficiently solved on GPUs, as has been shown in (Valizadeh & Berger, 2023).

1.2 Context-aware substring extraction

The regexes learned for extraction tasks have semantic differences from those learned for match existence tasks e.g. the pattern `.*a.*` is fundamentally indistinguishable from `a` for match existence, as `.*` does not impose any additional existence constraints on its own. For match extraction the former may retrieve a longer substring within the input.

For substring extraction tasks, the training data consists of marked regions of input, e.g. extracting only a small relevant part within the original input.

(Bartoli et al., 2016) presents a genetic programming based approach to generating regular expressions for text extraction from such marked examples. The tool is capable of synthesizing regular expressions for both tasks, which it supports via lookaround assertions.

The substring extraction task has a much larger search space than match existence, but it has extended capabilities as well. We argue that regular expressions for substring extraction tasks can reach some capabilities similar to deep learning models like BERT (Devlin, Chang, Lee, & Toutanova, 2019), which is designed with an attention mechanism for learning contextual relations between words. The task of context-aware text extraction requires a similar attention mechanism, which can be supported within regex syntax as lookarounds.

While the learning mechanism for tasks such as Text Summarization (e.g. marking the part of text that answers the question), or Named Entity Recognition (e.g. labeling and classifying individual parts of the input) is different from transformers, such as BERT (Devlin et al., 2019), it makes little difference whether the result comes from a probabilistic source, or a large dictionary of learned patterns, as both can be generalized or over-fitted the same.

We aim to show that intersection and complement in regular expressions have properties well suited for inference and machine learning tasks, where the intersections make the pattern modular and composable. A key difference of learning regexes this way, as opposed to neural networks, is that the final learned regular expression can sustain a vastly higher throughput of up to 1 GB/s on a single CPU thread.

2 Experiments

Our tool ExRegExEx excels at the substring extraction task, where we utilize heuristics combined with constructs supported by the RE# engine (Varatalu et al., 2024) (Table 1), where the *wildcard* `_*` matches *all strings*. We illustrate the efficacy of our approach on context-aware substring extraction by comparison to RegexGenerator++ (Bartoli et al., 2016), (see Table 3) where having support for intersections and complement allows for shorter inference times and overall more concise regexes learned. The *File* column of the Table 3 refers to the benchmark task, *len* column to the length of the learned regular expressions in characters, the *%* to the

Table 1: Basic constructs and their meaning in the extended regex syntax in **RE#**.

Lookarounds	*Prefixes/Suffixes*	*Other*
`(?<=`R`)_*` : preceded by R	R`_*` : starts with R	`_*`R`_*` : contains R
`(?<!`R`)_*` : not preceded by R	`~(`R`_*)` : does not start with R	`~(_*`R`_*)` : does not contain R
`_*(?=`R`)` : followed by R	`_*`R : ends with R	R`|`S : either R or S
`_*(?!`R`)` : not followed by R	`~(_*`R`)` : does not end with R	R`&`S : both R and S

Table 2: Advanced constructs in the extended regex syntax of **RE#**.

	Regex	*Notes*
Difference	$L \not\rightarrow R$	L but not R (`L&~`R)
Implies (*Negated Difference*)	$L \rightarrow R$	if L then R (`~`L`|`R)
XOR (*Symmetric Difference*)	$L \oplus R$	exactly one of L, R (`L&~`R`|~L&`R)
XNOR (*Neg. Symm. Diff.*)	$L \odot R$	both or none of L, R (`L&`R`|~L&~`R)
Window	`(?<=`L`~(_*`R`_*))_*`	located in a window between L and R
Between	`(?<=`L`)~(_*`L`|`R`_*)(?=`R`)`	between the boundaries of L and R

proportion of samples covered by the resulting regular expression, and the *Time* column to time taken to acquire the result. All experiments are run on a single thread. The fact that RegexGenerator++ utilizes genetic programming means that it is not guaranteed to generate a correct solution, which may often take a very long time to find, as is illustrated on the BibTeX title and BibTeX author tasks in Table 3. Just as a check for validity, we also ran our algorithm on AlphaRegex and FlashRegex benchmarks and achieved regexes matching the samples with 100% accuracy in the majority of cases with short run times. But the key benefit is in achieving superior results in context aware string extraction learning tasks with utilizing a combination of heuristic search for regular expressions and simplification steps.

Table 3: Regular expressions found by RegexGenerator++

	RegexGenerator++			**ExRegExEx**		
File	**len**	**%**	**Time**	**len**	**%**	**Time**
References_Lead-Author.json	40	100	00:55:36	37	100	00:00:04
Bibtex_Title.json	247	28.797	06:00:15	21	100	00:00:24
Bibtex_Author.json	264	66,975	05:47:01	58	100	00:01:17
reduced.json	29	100	00:01:45	9	100	00:00:03
Web-HTML_Heading.json	-	-	-	35	100	00:00:22
Web-HTML_Heading-Content.json	-	-	-	155	100	00:03:39

2.1 Exotic constructs within regular expressions

A key feature of having intersection and complement within regex syntax, is that it makes way for a whole effective boolean algebra of regexes, which allows programming constructs such as if-then implication ($\rightarrow$) or exclusive or ($\oplus$) to be encoded entirely within regex syntax. The RE# engine supports such constructs (see Table 2) without any search-time penalties, which when paired with the modular nature of regex intersections, creates a framework that could be competitive in the machine-learning space. Given the strong results in relatively little computational time in Table 3, we are eager to see which similar tasks are suitable for regex inference in the given framework.

References

Bartoli, A., Lorenzo, A. D., Medvet, E., & Tarlao, F. (2016). Inference of regular expressions for text extraction from examples. *IEEE Trans. Knowl. Data Eng.*, *28*(5), 1217–1230.

Chen, Q., Wang, X., Ye, X., Durrett, G., & Dillig, I. (2020). Multi-modal synthesis of regular expressions. In Donaldson, A. F., & Torlak, E. (Eds.), *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pp. 487–502. ACM.

Devlin, J., Chang, M., Lee, K., & Toutanova, K. (2019). BERT: pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., & Solorio, T. (Eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 4171–4186. Association for Computational Linguistics.

Goyvaerts, J. (2024). Lookarounds.. https://www.regular-expressions.info/lookaround.html.

Kim, J., Hu, Q., D'Antoni, L., & Reps, T. W. (2021). Semantics-guided synthesis. *Proc. ACM Program. Lang.*, *5*(POPL), 1–32.

Lee, M., So, S., & Oh, H. (2016). Synthesizing regular expressions from examples for introductory automata assignments. In Fischer, B., & Schaefer, I. (Eds.), *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pp. 70–80. ACM.

Li, Y., Xu, Z., Cao, J., Chen, H., Ge, T., Cheung, S., & Zhao, H. (2020). Flashregex: Deducing anti-redos regexes from examples. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pp. 659–671. IEEE.

Pan, R., Hu, Q., Xu, G., & D'Antoni, L. (2019). Automatic repair of regular expressions. *Proc. ACM Program. Lang.*, *3*(OOPSLA), 139:1–139:29.

Valizadeh, M., & Berger, M. (2023). Search-based regular expression inference on a GPU. *Proc. ACM Program. Lang.*, *7*(PLDI), 1317–1339.

Valizadeh, M., Gorinski, P. J., Iacobacci, I., & Berger, M. (2024). Correct and optimal: The regular expression inference challenge. In Larson, K. (Ed.), *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pp. 6486–6494. International Joint Conferences on Artificial Intelligence Organization. Main Track.

Varatalu, I. E., Veanes, M., & Ernits, J.-P. (2024). Re#: High performance derivative-based regex matching with intersection, complement and restricted lookarounds.. Conditionally accepted to POPL 2025, https://doi.org/10.48550/arXiv.2407.20479.

Cost Analysis for Import and Export Using an Abstract Machine

Benjamin Clausen Bennetzen[1], Emilie Sonne Steinmann[2], Loke Walsted[3], Nikolaj Rossander Kristensen[4], Peter Buus Steffensen[5], and Daniel Vang Kleist[6]

Aalborg University, Aalborg, Nordjylland, Danmark
{bbenne20[1],estein19[2],lwalst20[3],nrkr20[4],psteff19[5],dkleis20[6]}@student.aau.dk

1 Introduction

When working in web design, many components used by the programmer or designer are imported from other sources. These dependencies can have their own dependencies, and thereby vary in size. Therefore, the computational cost of inclusion of a dependency can also vary. This becomes an issue, when for instance a program, such as a web-based design tool, has to operate at 60 frames pr. second for an optimal user experience. This smooth experience could be challenged when including a new element that is not already imported and is computationally expensive. Analyzing the cost of such an import could help prove what impact it has on the experience. In this paper we examine a method that could be used for cost analysis.

A lot of previous work has been done tackling similar problems. Avanzini and Lago [1] introduces a method for complexity analysis for a functional programming language using a type system and Hoffmann et al. [5] gives a system for finding the worst-case resource bound.

One method that is recurring in some of these analyses is using a type system with sized types. Sized types as introduced by Hughes et al. [7] were used for proving properties such as liveness in reactive systems. This was later extended by Hughes and Pareto [6] to approximate stack and heap cost. Sized types are useful as they can be used to carry information about objects. We will make use sized types to carry information about the computational cost of files.

We will show an analysis method for a subset of JavaScript XML inspired by Baillot and Ghyselen [2], where they examines a method for finding the time complexity for a process in the π-calculus, by introducing semantics using a tick-notation to denote operations with an impact on the time complexity, as well as a type system using sized types. Diehl et al. [4] give an overview of different abstract machines and how they are used. We will introduce an abstract machine similar to the one introduced by Montenegro et al. [9]. Our abstract machine is in the style of Landin's SECD machine [8]. We will annotate its transitions with a cost using the tick-notation. In our cost model, we are interested in import and export so we will use the tick to mark reductions that will have an impact on this. We will also use tick to notify writing to memory, to show that programs without import and export also might have a substantial cost.

2 Abstract Machine

The abstract machine can be described using a runtime syntax, a machine configuration and a set of reduction rules. The machine configuration can be defined as in definition 2.1.

Definition 2.1 (Machine Configuration). The configuration of the abstract machine is defined as a 6-tuple:

$$(InstructionStack, fg, ls, es, vs, ss)$$

Where $InstructionStack$ is the set that ranges over all the possible formations of the runtime syntax. fg (file getter) is a partial function from identifiers to source files. ls (locals) is a partial function from a variable identifier and a scope identifier to a runtime value. es (exports) is a partial function from a variable identifier to a runtime value. vs (values) is a stack of runtime values. ss (scope stack) is a stack of scope identifiers. For a more formal introduction to these we refer to the full paper [3].

Every reduction $\stackrel{c}{\Rightarrow}$ has an associated cost c. As an example, $\stackrel{0}{\Rightarrow}$ denotes a reduction with a cost of 0. An example of a reduction rule can be seen below.

$$\begin{aligned}\langle \overrightarrow{id}\ f :: rest, fg, ls, es, vs, ss\rangle \stackrel{2}{\Rightarrow} \langle fg(f) :: PopScope ::\\ BindSelected\ id_1 :: \cdots :: BindSelected\ id_n :: EmptyExports :: rest, fg, ls, es, f :: ss\rangle\end{aligned}$$

This reduction rule describes how when one imports a list of variables from another file, it can be decomposed into the execution of said file, followed by the binding of a subset of the files exported variables to the local scope. One can read the reduction rule as follows: Given that a list of identifiers and a file name identifier is on top of the instruction stack, the configuration can be reduced with a cost of 2. The configuration will be reduced such that the instruction stack now contains the contents of the file f, a $PopScope$ command, a list of $BindSelected$ commands, and an $EmptyExports$ command. Also note that the file name identifier is pushed on top of the scopes stack.

3 Type System

The type judgements are of the form $Env \vdash e : t$ for expressions and $Env \vdash S : t \rhd Env'$ for statements, imports and exports. Env is a partial function from variables to types. Env has a special location ϵ where the types of exported variables are stored. t are the types of our type system, which are defined through the formation rules below.

$$\begin{array}{lcll} t & := & t_{Stm} | t_{Expr} & \\ t_{Stm} & := & n & \text{(number)} \\ & | & x & \text{(variable)} \\ & | & t_{Stm} + t_{Stm} & \text{(addition)} \\ & | & t_{Stm} \cdot t_{Stm} & \text{(multiplication)} \\ & | & t_{Stm} \uparrow t_{Stm} & \text{(maximum)} \\ t_{Expr} & := & t_{Stm} & \text{(actual cost)} \\ & | & t_{Expr} \to t & \text{(component)} \\ & | & \{id_i : t_{Expr_i}\}_{i \in 1 \ldots n} & \text{(record)} \end{array}$$

To type an import, one can use the typing rule below. The rule states that an import of a list of variables from a file can be typed as the addition of the type of the file, plus an amount n equaling the number of variables imported plus the constant cost of 2. The type environment now also contains the types of all the variables imported, as well as removing all the types

within the exports location ϵ.

$$\text{(T-ImportSelected)}\ \frac{[] \vdash fg(File) : t \rhd \Gamma'[\epsilon \to \{id_i : t_i\}_{i \in 1 \ldots n \ldots m}]}{\Gamma \vdash Import\ \overrightarrow{id}^{n}\ File : t + n + 2 \rhd \Gamma[id_1 \to t_1, \ldots, id_n \to t_n, \epsilon \to \{\}]}$$

Soundness in our type-system states that the type cost is greater than or equal to the cost of an execution of the program on the abstract machine. The problem here is that there exists a set of free variables $\vec{x}$ in the type t, one variable for each while loop. It must be the case that $\exists \sigma \in \vec{x} \to \mathbb{N}$ which is an instantiation of the free variables. Finding the function σ is incalculable in the case of free variables generated by while loops, as the number of iterations is only known at runtime. For the full Soundness proof of the Type System, we refer to the full paper [3].

4 Results

With the proof of soundness, we have shown that the type system is sound, meaning that the cost of all well-typed programs will be over-approximated. Furthermore, if the file does not include while loops, but only uses for-loops, which has an explicit number of iterations, the cost estimate will not need to introduce variables, and therefore, we can find an over-approximation without unknowns. We have also implemented the abstract machine and a constraint-gatherer, to infer the type of a program which can also be found on GitHub [3].

Since JavaScript XML is primarily used for internet and web-development, use cases for this type system could be to preemptively infer the cost of executing files, so that you could begin executing files with a high cost, before they are needed. This could potentially make a user interface more responsive, as perhaps the change of a button press could already begin to be calculated, when the user is hovering over the button, but before they press it. Pre-loading is used in other places, such as browsers loading the web page when the cursor is close to its hyperlink. It could also be used to find the more costly parts of a program, indicating which part of a code base that have the most potential to be improved upon. It can also give an estimate on a complexity of the algorithms in a file, as the type might include variables, that could indicate what degree of a polynomial the complexity of a file is. The complexity would not be sound, as we have no way for example to create exponential complexity with our type system.

5 Future Work

A simple extension of our current abstract machine is to include scopes in loops and conditionals.

Another extension of our work, would be to expand the smaller language, so that it becomes the full JavaScript XML language. This means that our type system would be able to be implemented and actually cost-approximate all of a JavaScript XML file.

Another way to extend this work would be to include a file-environment. This is because some files might already be loaded, and their exported values are already found. This means that we do not need to execute the imported file, as we already have the result of executing it. Therefore, a file that imports already loaded files should have its cost set to a much lower value, when over-approximating the cost of executing a file.

References

[1] Martin Avanzini and Ugo Dal Lago. Automating sized-type inference for complexity analysis. Proc. ACM Program. Lang., 1(ICFP), aug 2017.

[2] Patrick Baillot and Alexis Ghyselen. Types for parallel complexity in the pi-calculus. CoRR, abs/1910.02145, 2019.

[3] Benjamin Bennetzen. abstract-machine. https://github.com/BenjaminCB/abstract-machine, may 2024.

[4] Stephan Diehl, Pieter H. Hartel, and Peter Sestoft. Abstract machines for programming language implementation. Future Gener. Comput. Syst., 16(7):739–751, 2000.

[5] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for ocaml. CoRR, abs/1611.00692, 2016.

[6] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ml programming. ACM SIGPLAN Notices, 34, 07 1999.

[7] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '96, page 410–423, New York, NY, USA, 1996. Association for Computing Machinery.

[8] P. J. Landin. The mechanical evaluation of expressions. Comput. J., 6(4):308–320, 1964.

[9] Manuel Montenegro, Ricardo Peña-Marí, and Clara Segura. A resource-aware semantics and abstract machine for a functional language with explicit deallocation. In Moreno Falaschi, editor, Proceedings of the 17th International Workshop on Functional and (Constraint) Logic Programming, WFLP 2008, Siena, Italy, July 3-4, 2008, volume 246 of Electronic Notes in Theoretical Computer Science, pages 167–182. Elsevier, 2008.

Simple Worst-Case Optimal Joins

Fritz Henglein*, Changjun Li, and Mikkel Kragh Mathiesen

DIKU, University of Copenhagen

Abstract

We show that worst-case optimal joins, also for cyclic joins, are easy to program using basic programming techniques. They only require straightforward dictionaries, iterating over the smallest set when intersecting multiple sets and nested iteration over the variables in a join query in any order. We sketch a proof of worst-case optimality by amortization, where the execution cost is allocated to the generated output, but of padded input. We point out that modern database systems (still) generate asymptotically inferior code on cyclic joins.

1 Introduction

Let U be an infinite universe of values.[1] A *conjunctive query* is a first-order logic formula Φ with free variables $x_1, \ldots, x_k$ for some $k \geq 0$ built exclusively from predicate symbols applied to variables, $\wedge$ (conjunction) and $\exists$ (first-order quantification). Φ induces a function from *finite models* $\mathcal{M}$, interpretations of the predicate symbols by finite relations over U, to the set of environments ρ that satisfy Φ [2]. More precisely, Φ is semantically interpreted as

$$\mathcal{E}[\![\Phi]\!](\mathcal{M}) = \{(\rho(x_1), \ldots, \rho(x_k)) \mid \mathcal{M}, \rho \models \Phi\}.$$

We usually write $\{(x_1, \ldots, x_k) \mid \Phi\}$ for the output set of the query. Note that the output is a set of tuples, that is a relation, and indeed a finite relation.

A *join query* is a conjunctive query without existential quantifiers.

Example 1. *A directed graph can be represented by a binary relation R on U. Its* paths of length 3 *and* triangles *are definable as join queries:*

$$P_3 = \{(x_1, x_2, x_3) \mid R(x_1, x_2) \wedge R(x_2, x_3)\} \quad (1)$$

$$T = \{(x_1, x_2, x_3) \mid R(x_1, x_2) \wedge R(x_2, x_3) \wedge R(x_3, x_1)\} \quad (2)$$

Acyclic joins constitute a subclass of joins that can be efficiently evaluated, in time that is (quasi)linear in the sum of the sizes of the input and the output [4, 7, 6, 18, 22] by decomposing the query into subqueries in a certain way. These techniques yield *worst-case optimal* execution plans; in particular, their execution time is linear in the sum of the sizes of the input and the largest output for any input of that size. The remaining class of joins, *cyclic joins*, constitute the hard core of join queries where classical query optimization techniques provably do not yield worst-case optimal implementations [3, 15, 16], but asymptotically inferior implementations.

Example 2. *Given R of size (cardinality) N, paths of a length 3, P_3, is an acyclic join with maximal output size $\Theta(N^2)$. They can be computed in time $O(N^2)$ by straightforward nested iteration. This implementation is thus worst-case optimal. The maximal output size of triangles,*

*Corresponding author

[1] The following can be developed with multiple sorts/types of values. For simplicity we put them all into the same universe.

T, is only $O(N^{\frac{3}{2}})$, however, which follows from its fractional edge covering bound [3, 9]. It is usually implemented in database query engines by first computing P_3 and then filtering all triples (x_1, x_2, x_3) away where $(x_3, x_1) \notin R$. This takes $O(N^2)$ time, but since the maximal output of the query is asymptotically smaller, only $O(N^{\frac{3}{2}})$, this is not *a worst-case optimal join algorithm for computing triangles.*

Intuitively, the hard part of cyclic joins is *not* generating intermediate results that will eventually be filtered away.

Two distinct general worst-case optimal join algorithms were independently discovered only about a dozen years ago. One algorithm performs careful counting of sizes of intermediate tables, emulating the Grohe-Marx fractional edge covering bounds [17, 1]. These must be computed before-hand for each query, a nontrivial task in itself. Another approach introduces *leap-frog tries* and a non-constructive proof of worst-case optimality [21, 8], which seems to be practically superior. Both methods are rather involved, however; they are not implemented in mainstream databases.

2 Simple worst-case optimal joins

We show that, surprisingly, worst-case optimal joins are stunningly easy to implement. They require only three standard ingredients:

- dictionaries with constant-time size (length), lookup and domain iteration;
- iterating over the smallest set when intersecting multiple sets; and
- straightforward nested iteration over the join variables *in any order*.

Our join algorithm works, informally, as follows.[2]

1. Put the join query into the form

$$\{(x_1, \ldots, x_k) \mid C_1(x_1, \ldots) \wedge \ldots \wedge C_k(x_1, \ldots) \wedge D_{k+1}(\ldots) \wedge \ldots \wedge D_n(\ldots)\}$$

   where the C_i are the predicate symbols in the query with one occurrence of some freely chosen designated variable x_1 and the D_j are the remaining ones, without an occurrence of x_1.[3] The ellipses $\ldots$ stand for any sequence of variables not containing x_1.

2. Build dictionaries $\bar{C}$ such that $\bar{C}(x_1) = \{(x_2, \ldots, x_l) \mid C(x_1, x_2, \ldots, x_l)\}$.

3. For each $x_1 \in dom(\bar{C}_{i_0})$, where $\bar{C}_{i_0}$ is a $\bar{C}_i$ with the smallest domain, do

   (a) compute $D_1 = \bar{C}_1(x_1), \ldots, D_k = \bar{C}_k(x_k)$;

   (b) compute the join $\{(\ldots) \mid D_1(\ldots) \wedge \ldots \wedge D_k(\ldots) \wedge D_{k+1}(\ldots) \wedge \ldots \wedge D_n(\ldots)\}$ in the same fashion;

   (c) return $(x_1, \ldots)$ for each such result.

This unrolls into a nested loop over all the variables occurring in the query, which is straightforwardly expressed using comprehension notation as in SETL [19], Haskell [14] or Python [20].

[2] We use x_i both as metavariables for variables in formulae and for values in the models. Likewise we use C both as predicate symbol and for the relation that interprets it in a model.

[3] The case of multiple occurrences of x_1 is left out; it is straightforward to handle.

```
triangles rel = [ (x1, x2, x3) |
     let s = dict rel,           -- maps x to {y | (x,y) in rel}
     let sTransp =               -- maps y to {x | (x,y) in rel}
         dict [(y, x) | (x, y) <- rel]
     x1 <- minDom s sTransp,     -- iterates over dictionary with smallest domain
     let t1 = apply s x1,        -- {y | (x1,y) in rel}
     let t3 = apply sTransp x1,  -- {z | (z, x1) in rel}
     x2 <- minDom t1 s,          -- iterates over t1 or s, whichever is smallest
     contains t1 x2,             -- checks whether x2 is in t1
     let u2 = apply s x2,        -- {y' | (x2,y) in rel}
     x3 <- minDom u2 t3,         -- iterates over set u2 or t3, whichever is smallest
     contains u2 x3,             -- checks whether x3 is in u2
     contains t3 x3              -- checks whether x3 is in t3
     ]                           -- adds (x1, x2, x3) to the solution
```

Figure 1: Worst-case optimal triangles, in Haskell

Example 3. *Triangles can be computed monadically in Haskell using list comprehension notation. See Figure 1.*

Theorem 1. *Our join algorithm is worst-case optimal.*

Proof. (Sketch) We require dictionaries with constant-time lookup. A dictionary for a relation given as a list of pairs can be constructed in linear time.[4] It is important that the length of a dictionary be memoized to have constant-time complexity. This facilitates finding the dictionary with the smallest domain in constant time[5]. Then each constant-time step in the algorithm can be allocated to an output tuple that is eventually produced *if* a fixed fraction of the elements of the smallest domain $\bar{C}_{i_0}$ chosen to iterate over are also contained in the other domains. That is accomplished by padding the input relations to the query with *shadow elements*. This only doubles the input size, but ensures that each step in the algorithm is amortized over the output produced, which eventually yields that the algorithm is worst-case optimal. See [12] for a generalized application of this argument. □

It can be shown that choosing *any* static strategy for fixing the $\bar{C}_i$ to iterate over rather than choosing dynamically the $\bar{C}_{i_0}$ with smallest domain leads to asymptotically inferior performance.

Example 4. *The datasets* $R_n = \{(1,i) \mid 0 < i \leq n\} \cup \{(i,1) \mid 0 < i \leq n\}$ *have* $3n-2$ *triangles. Our join algorithm in Example 3 executes in* $O(n)$ *time whereas all variations that statically fix the dictionary to iterate over execute in* $\Theta(n^2)$ *time. See Table 2.*

Similarly, MySQL, SQLite and Postgres evaluate triangles on the R_n *datasets in* $\Theta(n^2)$ *time and have execution times comparable to rows 2-5 in Figure 2.*

Note that our algorithm beats the fractional covering bound of $O(n^{\frac{3}{2}})$ *in this case; this is because the dataset is already padded and thus executes in time linear in the size of the output.*

[4] Hashing-based dictionaries are standard; trie-based implementations tend to be better; comparison-based dictionaries/search trees worse.

[5] That is, dependent on the number of clauses in the query, but independent of the number of inputs in the relations

Iteration domain/R_n	1000	2000	4000	8000	16000	32000	...	16384000
Smallest domain	0.015	0.031	0.050	0.081	0.125	0.203	...	90.204
First/first domain	0.254	0.944	3.695	15.097	58.755	238.082		($\approx$ 2 yrs)
First/second domain	0.520	2.044	8.057	29.878	115.906	454.015		
Second/first domain	0.231	0.908	3.606	13.560	51.538	203.103		
Second/second domain	0.487	1.930	7.662	28.598	109.035	450.836		

Table 1: Examples of triangle computation times in seconds on $R_n = \{(1,i) \mid 0 < i \leq n\} \cup \{(i,1) \mid 0 < i \leq n\}$ using ghci, Version 9.4.7, on MacBook Air M1 (2020). The top row is for the code in Figure 1; the remaining four lines are for fixed choices of choosing `x2` and `x3`.

3 Discussion and future work

This is preliminary work. Next steps consist of systematic empirical evaluations with random and synthesized data sets as well as more comparisons with commercial and research database systems. Given the simplicity of implementing worst-case optimal joins demonstrated in this paper, a puzzling question is why almost all relational database systems, including all commonly used ones, to this day do not implement this. Since dictionaries are commonly used as index data structures in database systems, we conjecture that this is because they may have forgotten to employ the standard trick to always choose the smallest set to iterate over when intersecting sets. They likely use a statically fixed argument to iterate over, say the first argument, which demonstrably yields asymptotically suboptimal performance; see rows 2-5 in Table 2. Another reason could be the use of sort-merge join rather than hash table or radix-tree based joins. Sort-merge join correspond to traversing all the elements in the two input sets when computing their intersection rather than just the smallest and are thus asymptotically inferior on cyclic queries.

As it turns out, worst-case optimality can be extended to *algebraic joins* [12], which operate on potentially infinite relations and algebraic generalizations of relations.[6] The algebraic generalizations open avenues to expressive semantic frameworks for analytic queries, machine learning [5], reversible logic programming [10], and even quantum computing. Additionally, they provide a promising toolbox for efficient data structure and algorithm design, including parallel execution [11, 13].

References

[1] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 13–28, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences among relational expressions. *SIAM Journal on Computing*, 8(2):218–246, 1979.

[3] Albert Atserias, Martin Grohe, and Daniel Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42:739 – 748, 11 2008.

[4] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM (JACM)*, 30(3):479–513, 1983.

[6] Indeed, the present work has arisen by removing the generalization and specializing the proof of optimality to classical relational joins.

[5] Martin Elsman, Fritz Henglein, Robin Kaarsgaard, Mikkel Kragh Mathiesen, and Robert Schenck. Combinatory adjoints and differentiation. In Jeremy Gibbons and Max New, editors, *Proc. 9th Workshop on Mathematically Structured Functional Programming (MSFP)*, Electronic Proceedings in Theoretical Computer Science, 2022.

[6] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, November 2002.

[7] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.

[8] Todd J. Green. LogiQL: A declarative language for enterprise applications. In *Proceedings of the 34th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '15, pages 59–64, New York, NY, USA, 2015. ACM.

[9] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms (TALG)*, 11(1):1–20, 2014.

[10] Fritz Henglein, Robin Kaarsgaard, and Mikkel Kragh Mathiesen. Algeo: An algebraic approach to reversibility. In *International Conference on Reversible Computation*, pages 128–145. Springer, 2022.

[11] Fritz Henglein, Robin Kaarsgaard, and Mikkel Kragh Mathiesen. The programming of algebra. In *Proc. 9th Workshop on Mathematically Structured Functional Programming (MSFP)*, Munich, Germany, April 2022. Electronic Proceedings in Theoretical Computer Science (EPTCS).

[12] Fritz Henglein and Mikkel Kragh Mathiesen. Worst-case optimal algebraic joins. Unpublished manuscript, December 2020.

[13] Fritz Henglein and Mikkel Kragh Mathiesen. Synthetic algebraic programming. In Annie Liu, editor, *Proc. Logic and Practice of Programming (LPOP)*, December 2022.

[14] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language Haskell: a non-strict, purely functional language (version 1.2). *ACM SigPlan notices*, 27(5):1–164, 1992.

[15] H.Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. In *Proceedings of the 31st Symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[16] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

[17] Hung Q Ngo, Christopher Re, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *arXiv preprint arXiv:1310.3314*, 2013.

[18] Anna Pagh and Rasmus Pagh. Scalable computation of acyclic joins. *Journal of the ACM*, 2006.

[19] J. T. Schwartz, R. B. K. Dewar, E. Schonberg, and E. Dubinsky. *Programming with Sets: An Introduction to SETL*. Monographs in Computer Science. Springer New York, NY, 1986.

[20] Guido Van Rossum, Fred L Drake, et al. *Python reference manual*, volume 111. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[21] Todd L. Veldhuizen. Leapfrog triejoin: a worst-case optimal join algorithm. *CoRR*, abs/1210.0481, 2012.

[22] Dan E. Willard. An algorithm for handling many relational calculus queries efficiently. *Journal of Computer and System Sciences*, 65(2):295 – 331, 2002.

UI Test Automation Framework for Energy Analysis: Exploring Energy Compositionality of Actions*

Timmie M. R. Lagermann[1], Kristina Sophia Carter[1],
Su Mei Gwen Ho[1], Maja H. Kirkeby[1]

Roskilde University
timmiel,kcarter,smgho,majaht@ruc.dk

Abstract

UI Testing Frameworks have been suggested for evaluating the energy consumption of software. Previous research has studied how automation frameworks can increase the energy consumption from the individual actions, compared to a human produced baseline. In this study, we present the first explorations on the dependence between the individual actions. We explore the Selenium Test Framework's actions Input, i.e., inputting text into a text field, and Click, i.e., the action of clicking a button. These initial results show that the energy consumed by the individual Framework actions are dependent, and they are order-independent.

1 Introduction

UI Testing frameworks such as Selenium, Cypress, Nightwatch, Puppeteer, or Playwright, has been proposed for energy analysis of user tasks (user flows that solve a task) in software [1]. When measuring the energy consumption of a system, everything running on the system is included in the measurement. This presents an issue, as the energy overhead of the chosen UI testing framework is included in the energy measurement of a user task, making it difficult to isolate the energy consumption of the user flows themselves. Therefore, it is important to determine the overhead introduced by the framework. This has been studied for individual actions, such as "Click," "Input," "Double Click," and "Drag-and-Drop" on mobile phones [1]. However, it has not yet been studied how the energy overhead behaves in the case of action sequences, which is necessary for analyzing the energy consumption of user tasks. In this pre-study, we have chosen one UI testing framework, namely Selenium, to study the composition of actions of two selected actions: Click and Input (for inputting text into a text field) and their combinations. We will examine whether the energy consumption of the Selenium test framework's actions, Type Text and Click Button, is independent of each other and whether they are order-independent.

2 Methodology

Our aim is to explore the dependency between actions' overhead in energy consumption. The measurement setup is illustrated in Figure 1 and further specified in Table 1. It consists of five parts:

- Computer for Data collection: A laptop with TestController, to initiate the scripts and to collect and monitor the data of the experiment.

*This research was supported by the Independent Research Fund Denmark Project no. 2102-00281B

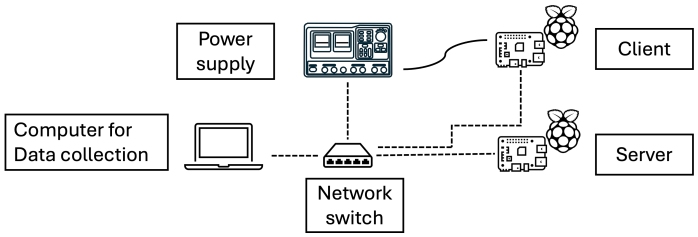


Figure 1: The Measurement setup.

- Switch: A network switch connects the components in a local network, via ethernet cables.
- Power supply: A Siglent (model SPD3303X-E) provides power to the Client and collects the energy data.
- Client: A Raspberry Pi with an installation of Selenium and where the scripts 4 reside namely: Click, Input, Click+input and Input+Click.
- Server: A Raspberry Pi to host the website. The web browser in use is Chromium, which is a bare bone version of Google Chrome.

The experiment and the measurements are started and monitored from the laptop running TestController[1]. The laptop initiates the Selenium scripts through SSH on the client, which runs the scripts headless on the server hosting a local website. The power supply provides a constant voltage with varying current and power, depending on the task of the client. These voltage, current and power are monitored and collected via the laptop.

In order to strengthen the reliability of our data, we conducted each experiment 35 times, while making sure that the setup each time were identical and independent. This also makes the energy measurement less prone to random fluctuations from various sources such as the underlying operation system and the measurement equipment [1, 2].

We execute our experiment in a controlled environment, to minimize potential side effects from user interaction with the UI [1]. To further control the underlying processes we disable some of the services with the following commands: "sudo apt remove unattended-upgrades" and "sudo rfkill block 0 1". We will not need to consider whether to include the idle energy usage, since this preliminary study is on the dependency between the actions' energy consumption.

Platform	Raspberry Pi 4 Model B Rev 1.5
Operation System	Linux version 6.1.21-v8+ Debian 11, Bullseye
Node Specifications	Node: v20.17.0, nvm: 0.40.0, npm: 10.08.02
Automation Framework	selenium-webdriver@4.24.1
Script repetitions (approx 20 sec)	Click: 120, Click;Input: 50, Input;Click: 50, Input: 70

Table 1: Platform specifications and script repetitions.

3 Results

Figure 2 shows the histogram and probability density function (left) and box plots of the 4 tests (right). A Shapiro-Wilk test indicated that the cleaned energy consumption data for 'Click'

[1] https://lygte-info.dk/project/TestControllerIntro%20UK.html

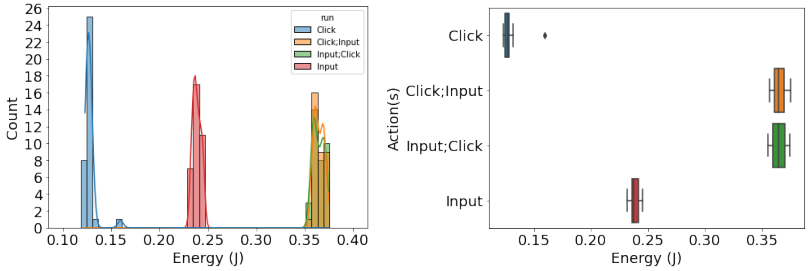


Figure 2: The Energy Consumption of actions and sequences.

(W = 0.977, p = 0.662) approximated normality, while the 'Input' data approached normality but remained borderline (W = 0.941, p = 0.062). However, the energy consumption of the two sequences are both bimodal. The distributions of Click and Input are significantly different, and each of them are significantly different from the compositions, i.e., the Mann-Whitney U Statistic: 35.0 with P-value: $1.22 * 10^{-11}$ below 0.05. In addition, the Spearman Correlation is 0.050 with P-value: 0.774 demonstrating that there is no significant relationship between them.

The individual actions are both consuming less energy than their sequences, which is following the expectation, that the combined test of both Click and Input should consume more than the individual actions. Indeed, summing the medians of the individual actions (Click: 0.1268, Input: 0.2376) to 0.3644 is close to the median of both the sequences (Click;Input: 0.3645, and Input;Click: 0.3644). The distributions of the sequences are bimodal and thus not normal distributed indicating that Click and Input are not independent in sequences [3]. While independence would have simplified the analysis, it is not strictly required for composition.

A visual inspection indicates that the two sequence distributions have similar tendencies, which suggests that the order of the actions are irrelevant. In addition, there is not enough evidence to reject that they come from the same distribution using Mann-Whitney U(Statistic: 646.0, P-value: 0.698). The P-value is above 0.05 and there is no significant difference between the two sequences. Thus, the evidence indicates that the order does not matter.

4 Conclusion

In this prestudy we have provided a setup to measure the energy consumption of the actions, Click and Input, for the automation Test Framework of Seleniums and the compositions of the actions. Our results show that the energy consumption of the actions consumes almost the same regardless of the ordering of the actions. This suggest that the order of the actions of Click and Input are independent. However, the results also indicate that the actions are dependent on each other when in sequence.

References

[1] Luís Cruz and Rui Abreu. On the energy footprint of mobile testing frameworks. *IEEE Transactions on Software Engineering*, 47(10):2260–2271, 2021.

[2] Alla G. Kravets and Vitaly Egunov. The software cache optimization-based method for decreasing energy consumption of computational clusters. *Energies*, 15, 10 2022.

[3] Eric W. Weisstein. Normal sum distribution, 2024. From MathWorld–A Wolfram Web Resource.

Bayesian Energy Profiler for Java (Extended Abstract)*

Joel Nyholm[1], Wojciech Mostowski[1], and Christoph Reichenbach[2]

[1] Halmstad University, Halmstad, Sweden
joel.nyholm@hh.se, wojciech.mostowski@hh.se
[2] Lund University, Lund, Sweden
christoph.reichenbach@cs.lth.se

1 Introduction & Motivation

Energy efficiency is a key concern in the design of mobile software, but no less significant for understanding the environmental impact of backend software in data centers. However, understanding and improving the energy usage of even simple software systems can be challenging, and modern language run-time systems with complex, dynamic optimizations exacerbate this challenge. Consider Java: Java stores executable code in a platform-independent intermediate representation, Java bytecode, that is executed on platform-specific Java Virtual Machines (JVMs). Energy consumption of Java code thus depends not only on the dynamic behavior of the just-in-time compiler and garbage collector, but also on peculiarities of the target hardware, operating system, and JVM implementation. To understand the energy impact of some software design decision, a software engineer today would need detailed knowledge that penetrates all these abstraction layers and accounts for the peculiarities of all intended target platforms.

To aid developers, we propose to offer tools that support the *Static Analysis of Energy Usage in Software*[1], that can both infer energy usage properties through scalable and explainable analysis techniques [2, 13], and verify explicit developer claims via theorem-proving based verification, extending prior work on energy usage modeling [6] to the KeY system [1].

We split the challenge of building such tools into modeling (1) the mapping from static program structure to dynamic traces, and (2) the energy consumption of dynamic traces. We here report on our initial exploration of this second aspect: building a statistical energy consumption model for Java bytecode in a controlled setting, on the Raspberry Pi 5 platform.

2 Statistical Energy Profiler for Java

Our energy profiler uses a Java bytecode pattern as its input and outputs a probability distribution of its corresponding energy cost, and examine the bytecode patterns emitted by the OpenJDK javac compiler (Figure 1).

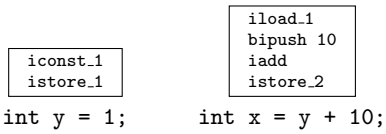


Figure 1: Bytecode patterns for two variable declarations and one addition

*This work is supported by ELLIIT funding for project D12.
[1] https://elliit.se/project/static-analysis-of-energy-usage-in-software/

Our energy model consists of two components: the measurement and the modeling component. The measurement component measures the energy of bytecode patterns through a current sensor connected to the execution platform and a power supply. The Voltage V from the power supply is constant, the current sensor measures the Amperage I, and the application measures the execution time t of the bytecode patterns. Thus, we can calculate the energy cost $E = V \times I \times t$. To account for statistical variation, we collect n_{total} samples for each pattern on our execution platform (Section 3). We divide these n_{total} samples across n_{div} subsets, which execute in stochastic order, shuffling the subsets of patterns. We currently draw the patterns from five categories: assignments, arithmetic, logic, conversion, and jumps, all of which we parameterize by any types and input sizes that appear in the patterns. We consider primitive types and one- and two-dimensional arrays, measuring all permutations.

The modeling component, implemented in R [14], models the energy usage for each bytecode pattern into probability distributions. We use Bayesian statistics, meaning initializing our models with explicit prior distributions that repeatedly update with the observed data. We use the following priors, based on what we have observed as typical current (I), voltage (V), and execution time (t) measurements, with a particularly wide standard deviation for t:

$$
\begin{aligned}
E &\sim \mathcal{N}(V \times I \times t, \sigma) \\
t &\sim \mathcal{N}(5e^{-4}, 1.5e^{-4}) \\
V &\sim \mathcal{N}(5, 0.05) \\
I &\sim \mathcal{N}(0.6, 0.1) \\
\sigma &\sim \mathcal{N}(1e^{-5}, 1e^{-6})
\end{aligned}
$$

Since the statistical model creates a Gaussian probability distribution for the energy cost of each bytecode pattern, we can utilize convolution to create a probability distribution consisting of several bytecode pattern's energy distributions. Hence, we can statically estimate the energy cost of a Java program considered as a bytecode collection matching our measured patterns.

3 Experimental Setup

Since time and energy consumption can be affected by a large number of parameters, we fix several parameters that we can control directly and account for variation in others. As our execution platform, we use the Raspberry Pi 5 with its CPU frequency fixed to 1.5 GHz, on Raspberry Pi OS Lite 64-bit 6.6.20 and OpenJDK 17.0.10, with garbage collection and JIT disabled. Since energy consumption can vary between different instances of the same hardware platform, due to variations in its electrical components, such as resistors [12] and PCBs [3], we measure on two (functionally identical) instances of the execution platform.

4 Related Work

Perhaps the closest work to ours is Hao et al.'s *eLens* system [4], which statically and statistically predicts energy usage of Android applications. Other work for Android has examined the connection between API usage and energy consumption [9] and strategies for associating energy usage and source code [8]. Prior work on Java has explored a number of other dimensions, including the role of the JVM on energy consumption [11].

The accuracy of these approaches hinges on the accuracy of their approach to energy profiling. Approaches with hardware-based sampling often attach an external power meter [9, 11]

or similar hardware device, while others, such as GreenMiner [5], use a custom measurement platform. Software-based sampling avoids the need for specialized hardware and instead utilizes monitors internal to the execution platform, e.g. via performance counters [7], though it is unclear how accurately these software metrics reflect actual power consumption [10].

5 Conclusions and Future Work

We have briefly presented our in-progress work on constructing Bayesian energy consumption models for Java. Our approach aims at obtaining large numbers of high-frequency samples to allow us to build precise statistical samples. Since the quality of our initial results has been hampered by limitations of our hardware power-usage monitors, we are currently evaluating alternative options for cost-effective power sensors.

References

[1] D. Bruns, W. Mostowski, and M. Ulbrich. Implementation-level Verification of Algorithms with KeY. *International Journal on Software Tools for Technology Transfer*, 17:729–744, Nov. 2015.

[2] A. Dura, C. Reichenbach, and E. Söderberg. JavaDL: Automatically Incrementalizing Java Bug Pattern Detection. In *Proceedings of the ACM on Programming Languages*. ACM, Sept. 2021.

[3] S. Ghosh and K. Roy. Parameter Variation Tolerance and Error Resiliency: New Design Paradigm for the Nanoscale Era. *Proceedings of the IEEE*, 98(10):1718–1751, 2010.

[4] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *ICSE 2013*, pages 92–101, 2013.

[5] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky. Greenminer: a hardware based mining software repositories software energy consumption framework. MSR 2014, page 12–21, New York, NY, USA, 2014. Association for Computing Machinery.

[6] R. Kersten, P. P. Toldin, B. van Gastel, and M. van Eekelen. A Hoare Logic for energy consumption analysis. In *Foundational and Practical Aspects of Resource Analysis*, LNPSE 8552. Springer, 2014.

[7] M. Kumar and W. Shi. Energy consumption analysis of java command-line options. In *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, 2019.

[8] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. ISSTA 2013, page 78–89, 2013.

[9] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: an empirical study. MSR 2014, page 2–11, New York, NY, USA, 2014. Association for Computing Machinery.

[10] J. Mair, D. Eyers, Z. Huang, and H. Zhang. Myths in power estimation with performance monitoring counters. *Sustainable Computing: Informatics and Systems*, 4(2):83–93, 2014. Special Issue on Selected papers from EE-LSDS2013 Conference.

[11] H. Oi. Power-performance analysis of JVM implementations. In *ICIMU 2011 : Proceedings of the 5th international Conference on Information Technology & Multimedia*, pages 1–7, 2011.

[12] V. K. Pandey and C. M. Tan. Effect of resistor tolerance on the performance of resistor network—An application of the statistical design of experiment. *International Journal of Circuit Theory and Applications*, 50(1):175–182, 2022.

[13] I. Riouak, C. Reichenbach, G. Hedin, and N. Fors. A precise framework for source-level control-flow analysis. In *SCAM 2021*. IEEE Computer Society, Sept. 2021.

[14] The R Foundation. The R Project for Statistical Computing. https://www.r-project.org/. Online; accessed 2024-09-20.

Input Reduction Revisited

Christian Gram Kalhauge

DTU Compute, Kgs. Lyngby, Denmark
chrg@dtu.dk

Abstract

Given an input that crashes your program, finding a minimal working example is crucial to enable easy debugging. We refer to this as the input reduction problem. Current state of the art techniques are either syntax oriented, which means they cannot do interesting transformations, or they are very painstakingly written to fit a specific input type. In this series of work we present a novel technique, we call Reduction Trees. It is a technique that allows the user to map out all possible sub-inputs of an input in an ordered binary tree. Theoretically, this technique allows for fast solutions to the input reduction problem, while being flexible enough to enable transformations. However, it has one big flaw: the trees had to be specified lazily because otherwise they were too big to fit in memory. This made them hard to encode in non-lazy languages. During last years talk on Input Reduction, many great questions were asked. The best, by far, was "Is it possible to write your reductions in a language like Python?" The answer turns out to be yes! and to great effect. Given a relatively modest implementation effort we are able to produce reduction results equivalent to the state of the art in a fraction of the time.

1 Input Reduction and Reduction Trees

Imagine you are writing a compiler for a programming language, and some user of that language has the audacity to run the compiler with an input that crashes the compiler. The first thing you will ask the user to do, is to provide an MWE – a minimal working example. And, when the user eventually fails at this, you manually and laboriously have to do this reduction yourself. This is the **Input Reduction Problem** [3, 8, 1, 2, 7].

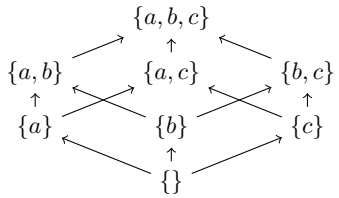
Defintion 1 (The Input Reduction Problem)**.** *Given $(I, \preceq_I, P, i)$, where i is an input from a partially ordered set (poset) of inputs $(I, \preceq_I)$, and a polynomial-time predicate P where $P(i)$, find $j \in I$ st. $j \preceq_I i$ st. it is a local minimum: $\forall k \in I.\ k \prec_I j \implies \neg P(k)$.*

Here the poset is the ordered set of all possible sub inputs of the original failing input, and the predicate P is running the compiler on the input to see if it fails. The local minimum is desirable, because then we know that every part of this input is important to recreate the bug.

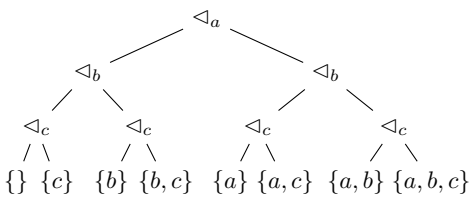
Reduction Trees Since it is unrealistic to store the reduction poset of even medium sized inputs in memory, we use a novel technique called reduction trees. Reduction trees both encode the poset which we want to do reduction over, as well as how to do the reduction.

Defintion 2 (Reduction Tree)**.** *A labeled reduction tree $r \in \mathbf{RTree}_{(I,\preceq_I)}$ from the poset $(I, \preceq_I)$ is a binary tree, with leafs from the poset $i \in I$, and with labeled nodes $r_1 \lhd_l r_2$. We say $i \in r$, if $r = i$ or $r = r_1 \lhd_l r_2$ and $i \in r_1 \vee i \in r_2$. We also refer to the rightmost element as the extracted element $\lceil r \rceil$: $\lceil r \rceil = i$ if $r = i$ and $\lceil r \rceil = \lceil r_2 \rceil$ if $r = r_1 \lhd_l r_2$. Finally each node $r_1 \lhd_l r_2$ must be consistent with the poset:*

$$r_1, r_2 \in \mathbf{RTree}_{(I,\preceq_I)} \quad \wedge \quad \lceil r_1 \rceil \preceq_I \lceil r_2 \rceil \quad \wedge \quad \forall j \in r.\ j \preceq_I \lceil r_1 \rceil \implies j \in r_1.$$



(a) The Hesse diagram.



(b) A reduction tree

```python
def reduce_abc(check) -> list:
  result = []
  for x in ['a', 'b', 'c']:
    if not check(f"remove {x}?"):
      result.append(x)
  return result
```

(c) The reduction tree as a Python program.

```
>>> p = lambda i: 'a' in i
>>> reduce(p, reduce_abc)
remove a? .. test ['b', 'c'] .. False
remove b? .. test ['a', 'c'] .. True
remove c? .. test ['a']      .. True
{'a'}
```

(d) Reduction given the predicate $P(i) \equiv a \in i$.

Figure 1: Reduction of $(2^{\{a,b,c\}}, \subseteq)$ using reduction trees encoded as Python program.

The intuition behind the tree is best captured by an example (see Figure 1). It depicts the Hesse diagram (fig. 1a) for the poset $(2^{\{a,b,c\}}, \subseteq)$ and a corresponding reduction tree (fig. 1b). The largest element of the tree is the extracted element $\lceil r \rceil = \{a, b, c\}$. The four interesting properties that the tree must have is that: 1) every leaf has to be in the poset; 2) every branch of a node has to be a reduction tree from the same poset, e.g., $\{\} \lhd_c \{c\} \in \mathbf{RTree}_{(2^{\{a,b,c\}}, \subseteq)}$; 3) the extracted element of a left branch has to be smaller that the right branch, e.g., $\{\} \subseteq \{c\} \subseteq \{b, c\} \subseteq \{a, b, c\}$; and 4) if an element in the tree is smaller than the extracted element of the left branch, it has to (also) be in the right branch, e.g., $(\{\} \lhd \{c\}) \lhd (\{\} \lhd \{a, c\})$ is a valid reduction tree, even though $\{\} \subseteq \{c\}$ because it is also on the left side.

Solving the Input Reduction Problem Assuming that the extracted element of the reduction tree satisfies the predicate $P(\lceil r \rceil)$, a reasonable assumption given that we start with an input that crashes the compiler, and it is of finite depth, we can easily find a smaller input that also satisfy the predicate:

$$\lfloor i \rfloor_P = i \quad \lfloor r_1 \lhd_l r_2 \rfloor_P = \texttt{if } P(\lceil r_1 \rceil) \texttt{ then } \lfloor r_1 \rfloor_P \texttt{ else } \lfloor r_2 \rfloor_P.$$

Let us illustrate the algorithm on the example (in Figure 1) with the predicate $P(i) \equiv a \in i$. The algorithm would first check the extracted element of the left branch of the top node $P(\{b, c\})$ (labeled a). This would fail, so we continue our reduction on the right branch (labeled b). Here we check $P(\{a, c\})$. It is true, which means we reduce to the left branch (labeled c). Finally, $P(\{a\})$ is true which means we reduce to the left branch. It is a leaf $\{a\}$ which we return.

If the reduction tree is of polynomial depth and if P is monotone over the poset $\preceq_I$, then this algorithm is a polynomial-time solution to the Input Reduction Problem.

2 Reduction Trees as Python Programs

The big limitation of the approach above is that it require us to specify a very big tree of inputs in some way. Previously, we used the lazyness of Haskell to encode the trees, but in

Listing 1: The reduction algorithm in Python.

```python
def reduce(predicate, rtree):
    path: list[bool] = []
    labels: list[str] = []
    def check(label):
        labels.append(label)
        return path[len(labels) - 1] if len(path) >= len(labels) else False
    i = rtree(check)
    while len(path) < len(labels):
        path.append(True); labels.clear()
        if t := predicate(j := rtree(check)): i = j
        else: path[-1] = False
        print(f"{labels[len(path) - 1]} .. test {j!r:<10} .. {t}")
    return i
```

this section, we will address this by encoding them as Python programs. The idea is simple: we can encode the tree as the paths of a Python program that given a source of external branching – a subprocedure that given a label can produce a boolean ($\mathbb{L} \rightsquigarrow \mathbb{B}$) – produce inputs: $(\mathbb{L} \rightsquigarrow \mathbb{B}) \rightsquigarrow I \supseteq \mathbf{RTree}_{(I, \preceq_I)}$. We call the source of external branches `check`, as it reads as checking if the label is true or false. In Figure 1c, we have encoded the reduction tree as a Python program. The program reads quite naturally, for every element in the set $\{a, b, c\}$, check if we should remove it, if not then add it to the results.

Underneath the covers (see Listing 1), reduction implemented as maintaining two lists, a `path` through the tree and a list of `labels`. `check` adds the label to a list while it direct the program down the correct path given the number of checks done so far (length of `labels`). If we are out of decisions in the path, we simply extract the right most element by returning false from then on. Reduction is now trivial, first extract the right most element of the tree, then until we have given every path a value (e.i., reached a leaf), we try to go down the true branch, essentially test $P(r_1)$, if that succeeds update i and continue down that branch, otherwise revert that choice to false, and go down the right branch. When we reach a leaf, return the input.

In Figure 1d, we use the algorithm in Listing 1 to reduce the tree.

3 Preliminary Results

The outstanding problems is now to do the encoding. Building reduction trees of poset that make real predicates relatively monotone is challenging, but not impossible. We have build encodings in three different languages, Haskell, Python, and LEAN. Each requires different extensions of the core idea presented here We are currently in the process of evaluating them.

We have built a reducer of C programs that competes with two state of the art reducers, C-Reduce [4] and Perses [5]. On a limited set of benchmarks we can do 99.8% and 99.97% of the reduction of C-Reduce and Perses in 10% and 26% of the single-threaded time. We can achieve this speedup because our reduction technique allows for precise reduction definitions and make the predicate relatively monotone.

We are also in the process of building a reducer for LEAN4, a theorem prover with extendable syntax. LEAN4 is a compiler in fast development and use. It happens that a change in the compiler produce a regression in the very big library mathlib4. This case study poses an interesting problem, because current syntax based approaches [3, 5, 6, 9] would not work as the syntax of a Lean program is first known at runtime.

References

[1] C. G. Kalhauge and J. Palsberg. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 556–566, 2019.

[2] C. G. Kalhauge and J. Palsberg. Logical bytecode reduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1003–1016, 2021.

[3] G. Misherghi and Z. Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151, 2006.

[4] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[5] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*, pages 361–371, 2018.

[6] Z. Xu, Y. Tian, M. Zhang, G. Zhao, Y. Jiang, and C. Sun. Pushing the limit of 1-minimality of language-agnostic program reduction. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):636–664, 2023.

[7] A. Zeller. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, 24(6):253–267, 1999.

[8] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[9] M. Zhang, Z. Xu, Y. Tian, Y. Jiang, and C. Sun. Ppr: Pairwise program reduction. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 338–349, 2023.

A Simple Method for Inverting Tail-Recursive Functions

Torben Ægidius Mogensen

DIKU, University of Copenhagen
torbenm@di.ku.dk

Abstract

Program inversion has long been studied. For functional programming languages, tail-recursive functions are an issue, as they are most often not reversible on their own, but only in specific calling contexts. We present a simple method for inverting functions in a subset of Haskell, and show that this can not handle tail recursion. We then propose a solution to this that involves a temporary rewrite of calls to tail-recursive functions to a functional iterative form, inverting this, and then rewrite back into a tail-recursive form.

1 Introduction

Inverting programs to compute the (possibly partial) inverse function of the original program has long been studied [1–6, 8, 10, 11].

The equivalent of loops in functional languages are tail-recursive functions, as these can be implemented without an unbounded stack. However, tail-recursive functions are most often not reversible on their own, but only in specific calling contexts. A paper on semi-inversion of first-order functional programs [9] proposed a simple solution to (fully) inverting tail-recursive functions. The main limitation of this method is that it does not handle tail-recursive functions with multiple non-recursive base cases, as it uses the single base case for initialising the inverse function. A recent paper [7] presents a more general inversion transformation that can handle multiple base cases in tail-recursive function definitions. It is, however, fairly complex and increases the size and computation time of the inverted program, as it introduces a reified continuation datatype that separates input, step, and output. We will extend the simple approach, but borrow the idea of separating the input, step, and output parts of tail-recursive functions and their calling context.

2 Program Inversion for a Subset of Haskell

We consider a small subset of Haskell extended with an iteration construct. The choice of language is not important, we would have used ML instead. In this subset, we can easily invert a single function rule. Fig. 1 shows both the syntax and the inversion rules. The inversion rules are unchanged from previous work [9], but the translation of calls to tail-recursive functions below is extended by having separate in, out, and step functions, where the original only had a step function.

Given a calling context of the form $\texttt{let}\ p_0 = f\ p_1\ \texttt{in}\ e_0$, where f is a tail-recursive function, we transform this into

$$\texttt{let}\ x = f_{in}\ p_1'\ \texttt{in let}\ y = \texttt{iter}\ f_{step}\ x\ \texttt{in let}\ p_0 = f_{out}\ y\ \texttt{in}\ e_0$$

where x and y are new variables and p_1' is a pattern that just contains the free variables of p_1 (in a tuple if more than one). We, additionally, add a definition $f_{in}\ p_1' = p_1$, rename the non-recursive (base-case) rules for f to f_{out}, rename the tail-recursive rules to f_{step}, and transform

in these rules the tail-recursive calls `let` $x = f\ p$ `in` x into just p. Note that f_{in}, f_{step} and f_{out} correspond to input, iteration and output configurations in [7].

We can now apply the inversion rules on both the calling context and the rules for f_{in}, f_{step}, and f_{out}.

What remains is to transform the rules back into a form that doesn't use the `iter` construct. The inverse of the calling context that uses the `iter` construct is of the form

$$\texttt{let}\ y = f_{out}^{-1}\ p_0\ \texttt{in let}\ x = \texttt{iter}\ f_{step}^{-1}\ y\ \texttt{in let}\ p_1' = f_{in}^{-1}\ x\ \texttt{in}\ e$$

and is transformed into

$$\texttt{let}\ y = f_{out}^{-1}\ p_0\ \texttt{in let}\ x = f^{-1}\ y\ \texttt{in let}\ p_1' = f_{in}^{-1}\ x\ \texttt{in}\ e$$

We, additionally, transform the right-hand side of the rules for f_{step}^{-1} by rewriting their output patterns p_i to tail calls: `let` $z = f^{-1}\ p_i$ `in` z, where z is a new variable. We then rename the rules for f_{in}^{-1} and f_{step}^{-1} to f^{-1}. The rules for f_{out}^{-1} are untouched.

This transformation does not guarantee that the inverted rules of functions will have disjoint input patterns, which is required to guarantee correctness of the inverse functions. So we can not invert all programs, but we can invert more than without the transformation.

3 Example with Multiple Base Cases

Consider the following variant of the reverse function, that in addition to returning the reversed list also returns an indication of whether the length of the list is odd or even.

```
reverseP xs = let z = revP (xs, []) in z
revP ([], ys) = (ys, Even)
revP ([x], ys) = (x:ys, Odd)
revP (x1:x2:xs, ys) = let z = revP (xs, x2:x1:ys) in z
```

Note the multiple non-recursive rules in `revP`. We first transform into iterative form:

```
reverseP xs
  = let x = revP_in xs in
    let y = iter revP_step x in
    let z = revP_out y in z
revP_in xs = (xs, [])
revP_out ([], ys) = (ys, Even)
revP_out ([x], ys) = (x:ys, Odd)
revP_step (x1:x2:xs, ys) = (xs, x2:x1:ys)
```

The program is inverted to

```
reverseP^-1 z
  = let y = revP_out^-1 z in
    let x = iter revP_step^-1 y in xs
    let xs = revP_in^-1 x in xs
revP_in^-1 (xs, []) = xs
revP_out^-1 (ys, Even) = ([], ys)
revP_out^-1 (x:ys, Odd) = ([x], ys)
revP_step^-1 (xs, x2:x1:ys) = (x1:x2:xs, ys)
```

$$
\begin{array}{lcl}
\mathit{Function} & \rightarrow & \textbf{Fid}\ \mathit{Pattern} = \mathit{Expression} \\
\\
\mathit{Pattern} & \rightarrow & \textbf{Vid} \mid \textbf{Cid} \mid \textbf{Cid}\ \mathit{Pattern} \mid (\mathit{Pattern}, \ldots, \mathit{Pattern}) \\
\\
\mathit{Expression} & \rightarrow & \mathit{Pattern} \\
\mathit{Expression} & \rightarrow & \texttt{let}\ \mathit{Pattern} = \mathit{Pattern}\ \texttt{in}\ \mathit{Expression} \\
\mathit{Expression} & \rightarrow & \texttt{let}\ \mathit{Pattern} = \textbf{Fid}\ \mathit{Pattern}\ \texttt{in}\ \mathit{Expression} \\
\mathit{Expression} & \rightarrow & \texttt{let}\ \mathit{Pattern} = \texttt{iter}\ \textbf{Fid}\ \mathit{Pattern}\ \texttt{in}\ \mathit{Expression}
\end{array}
$$

where **Fid** is a function identifier, **Vid** is a variable identifier, and **Cid** is a constructor identifier. We will use Haskell-like shorthands for lists. We apply the restriction that patterns are linear and variables are used exactly once in their scope. A tail call in this language is of the form $\texttt{let}\ x = f\ p\ \texttt{in}\ x$, where x is variable identifier, f is a function identifier, and p is a pattern. The semantics of $\texttt{let}\ p_1\ \texttt{=}\ \texttt{iter}\ f\ p_0\ \texttt{in}\ e$, where p_1 is a non-variable pattern, is that, f is called with the value v_0 of p_0 as input, giving a value v_1. If p_1 matches v_1, the matching is used in e (as in a normal let-expression). If p_1 does match v_1, v_1 is fed as input to f, and the output v_2 is matched against p_1 as above, continuing until a match is found.
If p_1 is a variable pattern y, we need a larger context:
$\texttt{let}\ y\ \texttt{=}\ \texttt{iter}\ f\ p_0\ \texttt{in let}\ p_1\ \texttt{=}\ g\ y\ \texttt{in}\ e$
Here, iteration continues until a rule in g matches y. This allows multiple different patterns to match y, where the simple form above allows only one pattern. This is an extension compared to previous work [9].

$$
\begin{array}{lcl}
\mathit{InvF}(f\ p_0 = e_0) & = & f^{-1}\ p_1 = e_1,\ \mathit{where}\ (p_1, e_1) = \mathit{Inv}(e_0, p_0) \\
\\
\mathit{Inv}(p, e) & = & (p, e) \\
\mathit{Inv}(\texttt{let}\ p_0 = p_1\ \texttt{in}\ e_1, e_0) & = & \mathit{Inv}(e_1, \texttt{let}\ p_1 = p_0\ \texttt{in}\ e_0) \\
\mathit{Inv}(\texttt{let}\ p_0 = g\ p_1\ \texttt{in}\ e_1, e_0) & = & \mathit{Inv}(e_1, \texttt{let}\ p_1 = g^{-1}\ p_0\ \texttt{in}\ e_0) \\
\mathit{Inv}(\texttt{let}\ p_0 = \texttt{iter}\ g\ p_1\ \texttt{in}\ e_1, e_0) & = & \mathit{Inv}(e_1, \texttt{let}\ p_1 = \texttt{iter}\ g^{-1}\ p_0\ \texttt{in}\ e_0)
\end{array}
$$

Figure 1: Simple function rules and their inversion

which we transform back to tail-recursive style:

```
reverseP⁻¹ z
  = let y = revP_out⁻¹ z in let xs = revP⁻¹ y in xs
revP_out⁻¹ (ys, Even) = ([], ys)
revP_out⁻¹ (x:ys, Odd) = ([x], ys)
revP⁻¹ (xs, []) = xs
revP⁻¹ (xs, x2:x1:ys)
  = let w = revP⁻¹ (x1:x2:xs, ys) in w
```

4 Conclusion

We have presented a program inversion method for a subset of Haskell and extended this with a method for inverting tail-recursive functions that can handle functions with multiple non-

recursive base cases. We have implemented the method, except transformation back to tail-recursive form. Compared to other methods for inverting tail-recursive functions with multiple base cases, we find our method relatively simple. The method can be extended to include non-linear variables and arithmetic, both with some restrictions.

References

[1] Edsger W. Dijkstra. Program inversion. In F. L. Bauer and M. Broy, editors, *Program Construction: International Summer School*, LNCS 69, pages 54–57. Springer-Verlag, 1978.

[2] David Eppstein. A heuristic approach to program inversion. In *Int. Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 219–221. Morgan Kaufmann, Inc., 1985.

[3] David Gries. *The Science of Programming*, chapter 21 Inverting Programs, pages 265–274. Texts and Monographs in Computer Science. Springer-Verlag, 1981.

[4] P. G. Harrison and H. Khoshnevisan. On the synthesis of function inverses. *Acta Informatica*, 29:211–239, 1992.

[5] H. Khoshnevisan and K. M. Sephton. InvX: An automatic function inverter. In Nachum Dershowitz, editor, *Rewriting Techniques and Applications. Proceedings*, LNCS 355, pages 564–568. Springer-Verlag, 1989.

[6] Richard E. Korf. Inversion of applicative programs. In *Int. Joint Conference on Artificial Intelligence (IJCAI-81)*, pages 1007–1009. William Kaufmann, Inc., 1981.

[7] Joachim Tilsted Kristensen, Robin Kaarsgaard, and Michael Kirkedal Thomsen. Tail recursion transformation for invertible functions. In *Reversible Computation*, pages 73–88. Springer Nature Switzerland, 2023.

[8] Torben Æ. Mogensen. Semi-inversion of guarded equations. In *GPCE'05*, Lecture Notes in Computer Science 3676, pages 189–204. Springer-Verlag, 2005.

[9] Torben Æ. Mogensen. Report on an implementation of a semi-inverter. In *PSI'06*, Lecture Notes in Computer Science 4378, pages 322–334. Springer-Verlag, 2007.

[10] A. Y. Romanenko. The generation of inverse functions in Refal. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 427–444. North-Holland, 1988.

[11] Jan L. A. van de Snepscheut. Program inversion (chapter 11). In *What Computing is All About*, Texts and Monographs in Computer Science, pages 219–251. Springer-Verlag, 1993.

Saving memory by collapsing sparse lists

Joachim Tilsted Kristensen[1] and Matthias Pall Gissurarson[2]

[1] University of Oslo, Norway joachkr@ifi.uio.no
[2] Chalmers Institute of Technology, Sweden pall@chalmers.se

Abstract

Classic texts on algorithms and data structures, such as CLRS[1], focuses on the imperative programming paradigm, which favours ephemeral structures and destructive operations. – Modern programming paradigms such as the functional paradigm, does not enjoy such a rich literature, and it remains challenging to design space efficient persistent data structures that perform as well as their ephemeral counterparts[2].

In this paper, present a sparse representation of lists that achieves constant time and space complexity for append operations and linear time for traversal in Haskell. This structure is particularly effective in contexts like merge sort, where an efficient implementation of append is essential, but it also generalises to monadic contexts such as the writer monad, that maintains an appendable structure, which is usually only traversed once, after all other computation has completed.

1 Introduction

Pure functional programs produce immutability data only, and while there are undoubtedly things to be gained from such a strict discipline, it also causes efficiency problems. For instance, append for lists cannot not share the first argument although it the result initially contains precisely the same elements. As illustrated in Figure 1, it is customary to append two lists by copying the elements of the first list with the alteration that the last pointer points to the beginning of the appended list rather than nil.

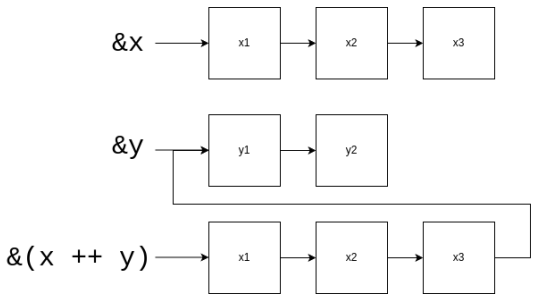


Figure 1: Append of `x` and `y` must make a copy of `x` to be persistent.

The downside implementing append in this way, is that it leads to an unexpected explosion in space complexity for algorithms that use it extensively. Take for instance the following, naive, implementation of Quicksort, that splits a list of n elements, until it is split into at most $2n$ lists, each containing 0 or 1 element

```
sort :: (Ord a) => [a] -> [a]
sort [    ] = []
sort (x:xs) = sort [a | a <- xs, a <= x] ++ [x] ++ sort [a | a <- xs, x < a]
```

It is well known, that this function is highly likely to complete this splitting in $O(n \cdot lg(n))$ comparison operations, and it is generally assumed that this is the most expensive part of its work. But this assumption relies on the ability to do all of the appends in $O(n)$ time. However, if we assume the best case (always split in the middle), then append is $O(n^2)$ both space and time complexity[1]. This is perhaps not completely surprising as the worst case time complexity of Quicksort is $O(n^2)$, but it does not improve at all for *Merge*sort.

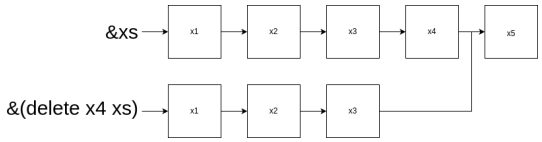


Figure 2: Deleting an element from `xs` requires us to make a copy of all elements that occur prior to the element we want to delete.

It isn't just the append operation (`++`) that unexpected amounts of space, any operation that needs to move a pointer, has to copy everything that comes before it. Take for instance `delete` in Figure 2.

2 Implmentation

A potential solution to our pain, could be to introduce the notion of a sparse list. That is a list, containing a pointer to a list, and an `Int`, specifying how many elements to take from there. For instance, the append in Figure 1 could be represented with the list `[(x,3), (y,2)]`, and the result of delete in Figure 2 could be represented by `[(xs, 3), (x4, 1)]`.

Traversing this list will still be linear, but it is still a list, and appending such sparse lists is still expensive, so we suggest the structure:

```
data SparseList a
  = FromList
      [a]              -- A native list source.
      Int              -- How many elements
  | Append
      (SparseList a)   -- First  operand
      (SparseList a)   -- Second operand
      (SparseList a)   -- Lazy collapse
      !Int             -- Number of elements
      !Int             -- Number of fragments
```

It repesents lists by a pointer to a native list, and a number of elements to take from there (also lets us implement `take` in constant time), or by the append of two sparse list together with the result of actually appending them.

However, since Haskell is a lazy functional programming language, the result of appending is not computed until it is actually needed (if it is needed). For instance, in the Quicksort example, we only look at the collapse in the last version of the structure, and for Mergesort, we only need to traverse the collapse of the resulting list.

[1] The latter, because you cannot use quadratic space in subquadratic time

Futhermore, this structure obvious instances of various Haskell type classes, such as `Functor`, `Applicative` and `Monad`, `Traversable` and `Foldable`, the latter of which lets us specify the collapse of a sparse list is ideally implemented traversing its elements with `foldr`. The definition is as follows following definition of `collapse`

```
collapse :: SparseList a -> SparseList a
collapse xs@(FromList     _ _) = xs
collapse (Append xs ys _ e _ ) = FromList c e
  where c = foldr (:) (foldr (:) [] ys) xs
```

We notice that, since a sparse list is defined in terms of a list and an `Int` describing how many elements to take, a program that uses our API will always express a finite list (even if the underlying native lists are infinite). Further more, since the above implementation of `collapse` does not inspect the result of collapsing the structure, we can be used it to implement append for sparse list without risking the hazard of infinite recursion like so:

```
(++) :: SparseList a -> SparseList a -> SparseList a
(++) xs ys =
  let c = Append xs ys (collapse c) (size xs + size ys) (frag xs + frag ys) in c
```

3 Discussion

Currently, we are formulating a library `Data.Sparse.List` for Haskell, which will be published along with an eventual full paper on the topic. The sparse list seems promising for certain algorithms in a preliminary benchmark, but we want to measure more thoroughly before making any bold claims.

Future work includes extending the technique of embedding a “collapsing” on a monoidal structure into the structure itself in such a way that we do not need to see all of the intermediate representations of the structure unless we need to. Potential usecases for this work are program analysis that output a set of constraints (e.g. by writing sets in the writer monad, which is the same problem but using union rather than append).

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[2] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, USA, 1998.