Russian Academy of Sciences
Ailamazyan Program Systems Institute

Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia

Proceedings
Pereslavl-Zalessky, Russia, July 1–5, 2010

Издательство Университет города Переславля
Pereslavl-Zalessky

УДК 004.42(063)
ББК 22.18

В874

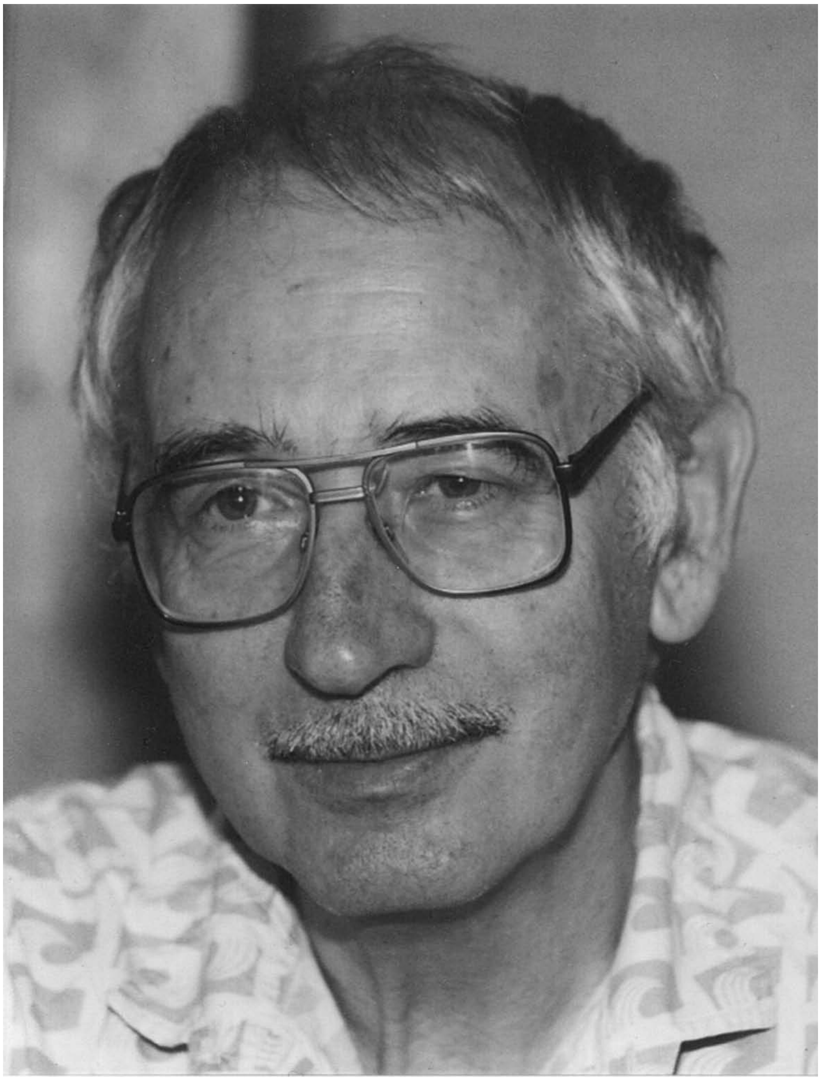
Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia // Proceedings of the second International Valentin Turchin Memorial Workshop on Metacomputation in Russia. Pereslavl-Zalessky, Russia, July 1-5, 2010 / *Edited by A. P. Nemytykh.* — Pereslavl Zalessky: Ailamazyan University of Pereslavl, 2010, **186** p. — 978-5-901795-21-7

Второй международный памяти В.Ф. Турчина семинар по метавычислениям в России // Сборник трудов Второго международного памяти В.Ф. Турчина семинара по метавычислениям в России, г. Переславль-Залесский, 1-5 июля 2010 г. / *Под редакцией А. П. Немытых.* — Переславль-Залесский: «Университет города Переславля», 2010, **186** с. *(англ).* — 978-5-901795-21-7

© Ailamazyan Program Systems Institute of RAS 2010
Институт программных систем им. А.К. Айламазяна РАН

© Ailamazyan University of Pereslavl 2010
НОУ «Институт программных систем —
„Университет города Переславля“» им. А.К. Айламазяна

ISBN 978-5-901795-21-7



Valentin Turchin
(1931-2010)

Workshop Organization

Honorary Chairman

$\boxed{\text{Valentin Turchin}}$, Professor Emeritus of The City University of New York, USA

Workshop Chair

Sergei Abramov, Program Systems Institute of RAS, Russia

Program Committee Chair

Andrei Nemytykh, Program Systems Institute of RAS, Russia

Program Committee

Mikhail Bulyonkov, A.P. Ershov Institute of Informatics Systems of RAS, Russia
Robert Glück, University of Copenhagen, Denmark
Geoff Hamilton, Dublin City University, Republic of Ireland
Viktor Kasyanov, A.P. Ershov Institute of Informatics Systems of RAS, Russia
Andrei Klimov, Keldysh Institute of Applied Mathematics of RAS, Russia
Alexei Lisitsa, Liverpool University, Great Britain
Johan Nordlander, Luleå University of Technology, Sweden
Sergei Romanenko, Keldysh Institute of Applied Mathematics of RAS, Russia
Claudio Russo, Microsoft Research Cambridge, United Kingdom
Peter Sestoft, IT University of Copenhagen, Denmark
Morten Sørensen, Formalit, Denmark

Invited Speakers

Neil D. Jones, Professor emeritus of University of Copenhagen, Denmark
Simon Peyton-Jones, Microsoft Research Ltd, Cambridge, England

Sponsoring Institutions

The Russian Academy of Sciences

Russian Foundation for Basic Research (№ 10-07-06029-г)

The Scientific and Technical Project SKIF-GRID of the Union of Russia and Belarus

Table of Contents

Programming in Biomolecular Computation 9
Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen

Approaches to Supercompilation 29
Simon Peyton Jones

Preliminary Report on a Self-Applicable Online Partial Evaluator for Flowchart 30
Robert Glück

A Graph-Based Definition of Distillation 47
G. W. Hamilton, G. Mendel-Gleason

Strengthening Supercompilation For Call-By-Value Languages 64
Peter A. Jonsson, Johan Nordlander

Towards Higher-Level Supercompilation 82
Ilya Klyuchnikov, Sergei Romanenko

A Simple Supercompiler Formally Verified in Coq 102
Dimitur Krustev

Supercompilation and Normalisation By Evaluation 128
Gavin E. Mendel-Gleason, Geoff Hamilton

A Method of Verification of Functional Programs Based on Graph Models 146
Andrew M. Mironov

Supercompilation and the Reduceron 159
Jason S. Reich, Matthew Naylor, Colin Runciman

A Note on three Programming Paradigms 173
N. V. Shilov

Programming in Biomolecular Computation

Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen*

{hartmann,neil,simonsen}@diku.dk
Department of Computer Science, University of Copenhagen (DIKU),
Copenhagen, Denmark

Abstract. Our goal is to provide a top-down approach to biomolecular computation. In spite of widespread discussion about connections between biology and computation, one question seems notable by its absence: **Where are the programs?** We introduce a model of computation that is evidently *programmable*, by programs reminiscent of low-level computer machine code; and at the same time *biologically plausible*: its functioning is defined by a single and relatively small set of chemical-like reaction rules. Further properties: the model is *stored-program*: programs are the same as data, so programs are not only executable, but are also compilable and interpretable. It is *universal*: all computable functions can be computed (in natural ways and without arcane encodings of data and algorithm); it is also *uniform*: new "hardware" is not needed to solve new problems; and (last but not least) it is *Turing complete* in a strong sense: a universal algorithm exists, that is able to execute any program, and is not asymptotically inefficient.
A prototype model has been implemented (for now in silico on a conventional computer). This work opens new perspectives on just how computation may be specified at the biological level.

Keywords: biomolecular, computation, programmability, universality.

1 Biochemical universality and programming

It has been known for some time that various forms of biomolecular computation are Turing complete [7,8,10,12,25,29,32,33]. The net effect is to show that any computable function can be computed, in some appropriate sense, by an instance of the biological mechanism being studied. However, the arguments for Turing universality we have seen are less than compelling from a *programming* perspective. This paper's purpose is to provide a better computation model where the concept of "program" is clearly visible and natural, and in which Turing completeness is not artificial, but rather a natural part of biomolecular computation. We begin by evaluating some established results on biomolecular computational completeness from a programming perspective; and then constructively provide an alternative solution. The new model seems biologically plausible, and usable for solving a variety of problems of computational as well as biological interest.

* © 2010 Published by Elsevier Science B. V.

It should be noted that while our model can support full parallelism (as often seen in biologically-inspired computing), it is not the foci of the paper, which are completeness and universality: we consider *one* program running on *one*, contiguous piece of data.

The central question: can program execution take place in a biological context? Evidence for "yes" includes many analogies between biological processes and the world of programs: *program-like behavior*, e.g., genes that direct protein fabrication; "switching on" and "switching off"; processes; and reproduction.

A clarification from the start: this paper takes a *synthetic* viewpoint, concerned with building things as in the engineering and computer sciences. This is in contrast to the ubiquitous *analytic* viewpoint common to the natural sciences, concerned with finding out how naturally evolved things work.

The authors' backgrounds lie in the semantics of programming languages, compilers, and computability and complexity theory; and admittedly not biology. We focus on the synthetic question **can**, rather than the usual natural scientists' analytical question **does**.

Where are the programs? In existing biomolecular computation models it is very hard to see anything like a program that realises or directs a computational process. For instance, in cellular automata the program is expressed only in the initial cell configuration, or in the global transition function. In many biocomputation papers the authors, given a problem, cleverly devise a biomolecular system that can solve this particular problem. However, the algorithm being implemented is hidden in the details of the system's construction, and hard to see, so the program or algorithm is in no sense a "first-class citizen". Our purpose is to fill this gap, to establish a biologically feasible framework in which programs are first-class citizens.

2 Relation to other computational frameworks

We put our contributions in context by quickly summarising some other computational completeness frameworks. **Key dimensions**: uniformity; programmability; efficiency; simplicity; universality; and biological plausibility. (Not every model is discussed from every dimension, e.g., a model weak on a dimension early in the list need not be considered for biological plausibility.)

Circuits, BDDs, finite automata. While well proven in engineering practice, these models don't satisfy our goal of computational completeness. The reason: they are *non-uniform* and so not Turing complete. Any single instance of a circuit or a BDD or a finite automaton has a control space and memory that are both finite. Consequently, any *general but unbounded* computational problem (e.g., multiplying two arbitrarily large integers) must be done by choosing one among an infinite family of circuits, BDDs or automata.

The Turing machine. *Strong points.* Highly successful for theoretical purposes, the Turing model is uniform; there exists a clear concept of "program";

and the "universal Turing machine" from 1936 is the seminal example of a self-interpreter. The Turing model has fruitfully been used to study computational complexity problem classes as small as PTIME and LOGSPACE.

Weak points. Turing machines do not well model computation times small enough to be realistically interesting, e.g., near-linear time. The inbuilt "data transport" problems due to the model's one-dimensional tape (or tapes, on a multi-tape variant) mean that naturally efficient algorithms may be difficult to program on a Turing machine. E.g., a time $O(n)$ algorithm may suffer *asymptotic slowdown* when implemented on a Turing machine, e.g., forced to run in time $O(n^2)$ because of architectural limitations. A *universal Turing machine* has essentially the same problem: it typically runs quadratically slower than the program it is simulating. Stiull greater slowdowns may occur if one uses smaller Turing complete languages, for instance the counter or Minsky register machines as used in [7,8,12,22].

Other computation models with an explicit concept of program. Numerous alternatives to the Turing machine have been developed, e.g., the Tag systems studied by Post and Minsky, and a variety of register or counter machines. Closer to computer science are recursive functions; the λ-calculus; functional programming languages such as LISP; and machines with randomly addressable memories including the RAM and, most relevant to our work, its stored-program variant the RASP [19]. These models rate well on some of the key dimensions listed above. However they are rather complex; and were certainly not designed with biological plausibility in mind.

Cellular automata. John von Neumann's pathbreaking work on cellular automata was done in the 1940s, at around the time he also invented today's digital computer. In [29] computational completeness was established by showing that any Turing machine could be simulated by a cellular automaton. Further, it was painstakingly and convincingly argued that a cellular automaton could achieve self-reproduction.Von Neumann's and subsequent cellular automaton models, e.g., LIFE and Wolfram's models[15,8,32], have some shortcomings, though. Though recent advances have remedied the lack of asynchronous computations [23], a second, serious drawback is the lack of *programmability*: once the global transition function has been selected (e.g., there is only one such in LIFE) there is little more that the user of the system can do; the only degree of freedom remaining is to choose the initial configuration of cell states. There is no explicit concept of a program that can be devised by the user. Rather, any algorithmic ideas have to be encoded in a highly indirect manner, into either the global transition function or into the initial cell state configuration; in a sense, the initial state is *both* program and input, but in the zoo of cellular automata proven to be universal, there seems to be no clear way to identify which parts of the initial state of the CA corresponds to, say, a certain control structure in a program, or a specific substructure of a data structure such as a list.

Biomolecular computation frameworks. We will see that the Turing-typical asymptotic slowdowns can be avoided while using a biomolecular computing model. This provides an advance over both earlier work on automata-based computation models (Turing machines, counter machines, etc.), and over some other approaches to biomolecular computing

A number of contributions exist in this area; a non-exhaustive list: [1,3,7,10,8,11,12,17,20,21,25,26,30,31,5,33] The list is rather mixed: Several of the articles describe concrete finite-automaton-like computations, emphasising their realisation in actual biochemical laboratory contexts. As such their emphasis is not on general computations but rather on showing feasibility of specific computations in the laboratory. Articles [7,8,12,20,33] directly address Turing completeness, but the algorithmic or programming aspects are not easy to see.

How our approach is different: Contrary to several existing models, our atomic notion (the "blob") carries a *fixed* amount of data and has a *fixed* number of possible interaction points with other blobs. Further, one *fixed* set of rules specify how local collections of blobs are changed. In this sense, our setup resembles specific cellular automata, e.g. Conway's game of life where only the initial state may vary. Contrary to cellular automata, there is both programs and data are very clearly identified ensembles of blobs. Further, we use a textual representation of programs closely resembling machine code such that each line essentially corresponds to a single blob instruction with parameters and bonds. The resulting code conforms closely to traditional low-level programming concepts, including use of conditionals and jumps.

Outline of the paper: Section 3 introduces some notation to describe program execution. Section 4 concerns the *blob model* of computation, with an explicit program component. Section 5 relates the blob model to more traditional computation models, and Section 6 concludes. Appendix A has more discussion of computational completeness; and Appendix B shows how a Turing machine may be simulated in the blob model – doable within a constant slowdown because of the flexibility of blobs when considered as data structures. Appendix C discusses the blob model's realisability in 3-dimensional space.

3 Notations: direct or interpretive program execution

What do we mean by a program (roughly)? An answer: a *set of instructions* that specify *a series (or set) of actions on data.* Actions are carried out when the instructions are executed (activated,...) Further, a program is software, not hardware. Thus a program should itself be a *concrete data object that can be replaced* to specify different actions.

Direct program execution: write $[\![\texttt{program}]\!]$ to denote the meaning or net effect of running `program`. A program meaning is often a function from data input values to output values. Expressed symbolically:

$$[\![\texttt{program}]\!](\texttt{data}_{in}) = \texttt{data}_{out}$$

The `program` is *activated* (run, executed) by applying the semantic function $[\![\_]\!]$. The *task of programming* is, given a desired semantic meaning, to find a program that computes it. Some mechanism is needed to execute `program`, i.e., to compute $[\![\texttt{program}]\!]$. This can be done either by hardware or by software.

Interpretive program execution: Here `program` is a passive data object, but it is now activated by running the *interpreter program.* (Of course, some mechanism will be needed to run the interpreter program, e.g., hardware or software.) An equation similar to the above describes the effect of interpretive execution:

$$[\![\texttt{interpreter}]\!](\texttt{program}, \texttt{data}_{in}) = \texttt{data}_{out}$$

Note that `program` is now used as data, and not as an active agent. Self-interpretation is possible and useful [18]; the same value $\texttt{data}_{out}$ can be computed by:

$$[\![\texttt{interpreter}]\!](\texttt{interpreter}, (\texttt{program}, \texttt{data}_{in})) = \texttt{data}_{out}$$

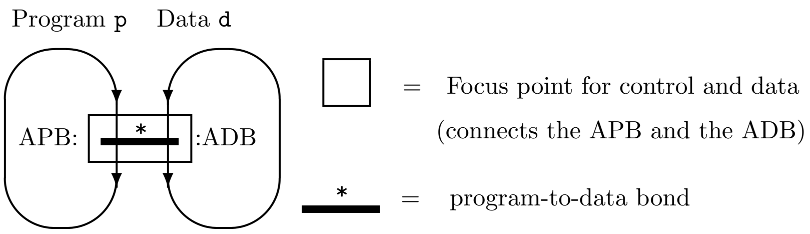
4 Programs in a biochemical world

Our goal is to express programs in a biochemical world. Programming assumptions based on silicon hardware must be radically re-examined to fit into a biochemical framework. We briefly summarize some qualitative differences.

- **There can be no pointers to data:** addresses, links, or unlimited list pointers. In order to be acted upon, a data value must be *physically adjacent* to some form of actuator. A biochemical form of adjacency: a chemical bond between program and data.
- **There can be no action at a distance**: all effects must be achieved via chains of local interactions. A biological analog: signaling.
- **There can be no nonlocal control transfer**, e.g., no analog to GOTOs or remote function/procedure calls. However some control loops are acceptable, provided the "repeat point" is (physically) near the loop end. A biological analog: a bond between different parts of the same program.
- On the other hand there exist available **biochemical resources** to tap, i.e., free energy so actions can be carried out, e.g., to construct local data, to change the program control point, or to add local bonds into an existing data structure. Biological analogs: Brownian movement, ATP, oxygen.

The above constraints suggest how to structure a biologically feasible model of computation. The main idea is to keep both program control point and the current data inspection site always close to a *focus point* where all actions occur. This can be done by continually shifting the program or the data, to keep the active program and data always in reach of the focus. The picture illustrates this idea for direct program execution.

Running program p, i.e., computing ⟦p⟧(d)


Program p
Data d
APB:
*
:ADB
= Focus point for control and data (connects the APB and the ADB)
= program-to-data bond

4.1 The Blob model

We take a very simplified view of a (macro-)molecule and its interactions, with abstraction level similar to the Kappa model [12,7,14]. To avoid misleading detail questions about real molecules we use the generic term “blob” for an abstract molecule. A collection of blobs in the biological “soup” may be interconnected by two-way *bonds* linking the individual blobs’ *bond sites*.

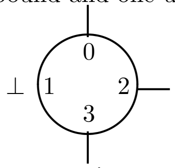
A *program p* is (by definition) a connected assembly of blobs. A data value *d* is (also) by definition a connected assembly of blobs. At any moment during execution, i.e., during computation of $[\![p]\!](d)$ we have:

- *One* blob in *p* is active, known as the *active program blob* or APB.
- *One* blob in *d* is active, known as the *active data blob* or ADB.
- A bond $*$, between the APB and the ADB, is linked at bond site 0 of each.

The data view of blobs: A blob has several bond sites and a few bits of local storage limited to fixed, finite domains. Specifically, our model will have *four bond sites*, identified by numbers $0, 1, 2, 3$. At any instant during execution, each can hold a bond – that is, a link to a (different) blob; or a bond can hold $\perp$, indicating unbound.

In addition each blob has 8 *cargo bits* of local storage containing Boolean values, and also identified by numerical positions: $0, 1, 2, \ldots, 7$. When used as program, the cargo bits contain an instruction (described below) plus an activation bit, set to 1. When used as data, the activation bit must be 0, but the remaining 7 bits may be used as the user wishes.

A blob with 3 bond sites bound and one unbound:



Since bonds are in essence two-way pointers, they have a “fan-in” restriction: a given bond site can contain at most one bond (if not $\perp$).

The program view of blobs: Blob programs are sequential. There is no structural distinction between blobs used as data and blobs used as program.

A single, fixed set of instructions is available for moving and rearranging the cursors, and for testing or setting a cargo bit at the data cursor. Novelties from a computer science viewpoint: there are no explicit program or data addresses, just adjacent blobs. At any moment there is only *a single program cursor* and *a single data cursor*, connected by a bond written `*` above.

Instructions, in general. The blob instructions correspond roughly to "four-address code" for a von Neumann-style computer. An essential difference, though, is that a bond is a *two-way link between two blobs*, and is not an address at all. It is not a pointer; there exists no address space as in a conventional computer. A blob's 4 bond sites contain links to other instructions, or to data via the APB-ADB bond `*`.

For program execution, one of the 8 cargo bits is an "activation bit"; if 1, it marks the instruction currently being executed. The remaining 7 cargo bits are interpreted as a 7-bit instruction so there are $2^7 = 128$ possible instructions in all. An instruction has an *operation code* (around 15 possibilities), and 0, 1 or 2 *parameters* that identify single bits, or bond sites, or cargo bits in a blob. See table below for current details. For example, `SCG v c` has 16 different versions since `v` can be one of 2 values, and `c` can be one of 8 values.

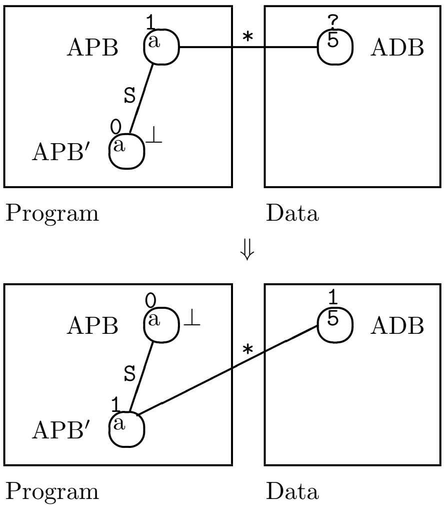
Why exactly 4 bonds? The reason is that each instruction must have a bond to its predecessor; further, a test or "jump" instruction will have two successor bonds (true and false); and finally, there must be one bond to link the APB and the ADB, i.e., the bond `*` between the currently executing instruction and the currently visible data blob. The `FIN` instruction is a device to allow a locally limited fan-in.

A specific instruction set (a bit arbitrary)**.** The formal semantics of instruction execution are specified precisely by means of a set of 128 biochemical reaction rules in the style of [12]. For brevity here, we just list the individual instruction formats and their informal semantics. Notation: `b` is a 2-bit bond site number, `c` is a 3-bit cargo site number, and `v` is a 1-bit value.

Numbering convention: the program APB and the data ADB are linked by bond `*` between bond sites 0 of the APB and the ADB. An instruction's predecessor is linked to its bond site 1; bond site 2 is the instruction's normal successor; and bond site 3 is the alternative "false" successor, used by jump instructions that test the value of a cargo bit or the presence of a bond.

Instruction	Description	Informal semantics (:=: is a two-way interchange)
`SCG v c`	Set CarGo bit	ADB.c := v; APB := APB.2
`JCG c`	Jump CarGo bit	if ADB.c = 0 then APB := APB.3 else APB := APB.2
`JB b`	Jump Bond	if ADB.b = ⊥ then APB := APB.3 else APB := APB.2
`CHD b`	CHange Data	ADB := ADB.b; APB := APB.2
`INS b1 b2`	INSert new bond	new.b2 :=: ADB.b1; new.b1 :=: ADB.b1.bs; APB := APB.2 Here "new" is a fresh blob, and "bs" is the bond site to which ADB.b1 was bound before executing `INS b1 b2`.
`SWL b1 b2`	SWap Links	ADB.b1 :=: ADB.b2.b1; APB := APB.2
`SBS b1 b2`	SWap Bond Sites	ADB.b1 :=: ADB.b2; APB := APB.2
`SWP1 b1 b2`	Swap bs1 on linked	ADB.b1.1 :=: ADB.b2.1; APB := APB.2
`SWP3 b1 b2`	Swap bs3 on linked	ADB.b1.3 :=: ADB.b2.3; APB := APB.2
`JN b1 b2`	Join b1 to linked b2	ADB.b1 :=: ADB.b1.b2; APB := APB.2
`DBS b`	Destination bond site	Cargo bits 0,1 := bond site number of destination for ADB.b
`FIN`	Fan IN	APB := APB.2 (bond site 3 is an alternative predecessor)
`EXT`	EXiT program	

An example in detail: the instruction `SCG 1 5`, as picture and as a rewrite rule. SCG stands for "set cargo bit". The effect of instruction `SCG 1 5` is to change the 5-th cargo bit of the ADB (active data blob) to 1. First, an informal picture to show its effect:



Note: the APB-ADB bond `*` has moved: Before execution, it connected APB with ADB. After execution, it connects APB′ with ADB, where APB′ is the next instruction: the successor (via bond `S`) of the previous APB. Also note that the activation bit has changed: before, it was 1 at APB (indicating that the APB was about to be executed) and 0 at ADB′. Afterwards, those two bit values have been interchanged.

Syntax: Code the above instruction as an 8-bit string: $\overbrace{1}^{a}\overbrace{100}^{SCG}\overbrace{1}^{v}\overbrace{101}^{c}$. Here activation bit $a = 1$ indicates that this is the current instruction (about to be executed). Operation code `SCG` (happens to be) encoded as 100; and binary numbers are used to express the new value: $v = 1$, and the number of the cargo bit to be set: $c = 5$.

The instruction also has four bond sites: $*PS\bot$. Here P is a bond to the predecessor of instruction `SCG 1 5`, S is a bond to its successor, and bond site 3 is not used. The full instruction, with 8 cargo sites and four bond sites can be written in form[1]: $B[11001101](*PS\bot)$.

Semantics: Instruction `SCG 1 5` transforms the three blobs APB, APB′ and ADB as in the picture above. This can be expressed more exactly using a rewrite rule as in [12] that takes three members of the blob species into three modified ones. For brevity we write "-" at bond sites or cargo sites that are not modified by the rule. Remark: the labels APB, ADB, etc. are not part of the formalism, just labels added to help the reader.

$$\overbrace{B[1\ 100\ 1\ 101](*\text{-}S\text{-})}^{APB},\ \overbrace{B[0\text{-}\text{-}\text{-}\text{-}\text{-}\text{-}\text{-}](\bot S\text{-}\text{-})}^{APB'},\ \overbrace{B[0\text{-}\text{-}\text{-}\text{-}x\text{-}\text{-}](*\text{-}\text{-}\text{-})}^{ADB}$$
$$\Rightarrow$$
$$\underbrace{B[0\ 100\ 1\ 101](\bot\text{-}S\text{-})}_{APB},\ \underbrace{B[1\text{-}\text{-}\text{-}\text{-}\text{-}\text{-}\text{-}](\bot S\text{-}\text{-})}_{APB'},\ \underbrace{B[0\text{-}\text{-}\text{-}\text{-}1\text{-}\text{-}](*\text{-}\text{-}\text{-})}_{ADB}$$

5 The blob world from a computer science perspective

First, an operational image: Any well-formed blob program, while running, is a collection of program blobs that is adjacent to a collection of data blobs, such that there is *one* critical bond (`*`) that links the APD and the ADB (the active program blob and the active data blob). As the computation proceeds, the program or data may move about, e.g., rotate as needed to keep their contact points adjacent (the APB and the ADB). For now, we shall not worry about the thermodynamic efficiency of moving arbitrarily large program and data in this way; for most realistic programs, we assume them to be sufficiently small (on the order of thousands of blobs) that energy considerations and blob coherence are not an issue.

5.1 The blob language

It is certainly small: around 15 operation codes (for a total of 128 instructions if parameters are included). Further, the set is irredundant in that no instruction's effect can be achieved by a combination of other instructions. There are easy

[1] B stands for a member of the blob "species".

computational tasks that simply cannot be performed by any program without, say, `SCG` or `FIN`.

There is certainly a close analogy between blob programs and a *rudimentary machine language*. However a bond is not an address, but closer to a two-way pointer. On the other hand, there is *no address space*, and *no address decoding hardware* to move data to and from memory cells. An instruction has an unusual format, with 8 single bits and 4 two-way bonds. There is no fixed word size for data, there are no computed addresses, and there are no registers or indirection.

The blob programs has some similarity to *LISP or SCHEME*, but: there are no variables; there is no recursion; and bonds have a "fan-in" restriction.

5.2 What can be done in the blob world?

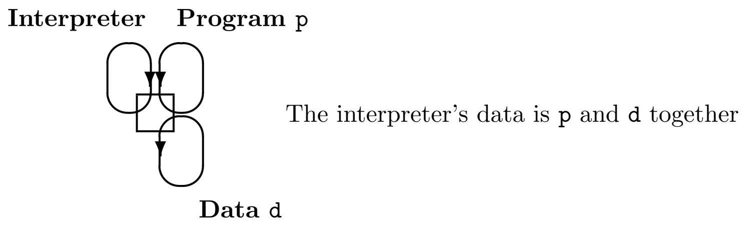
In principle the ideas presented and further directions are clearly expressible and testable in Maude or another tool for implementing term rewriting systems, or the kappa-calculus [7,9,12,14]. Current work involves programming a blob simulator. A prototype implementation has been made, with a functioning self-interpreter.

The usual programming tasks (appending two lists, copying, etc.) can be solved straightforwardly, albeit not very elegantly because of the low level of blob code. Appendix B shows how to generate blob code from a Turing machine, thus establishing Turing-completeness.

It seems possible to make an analogy between universality and self-reproduction that is tighter than seen in the von Neumann and other cellular automaton approaches. It should now be clear that familiar Computer Science concepts such as interpreters and compilers also make sense also at the biological level, and hold the promise of becoming useful operational and utilitarian tools.

5.3 Self-interpretation in the blob world

The figure of Section 4 becomes even more interesting when a program is executed interpretively, computing $[\![\texttt{interpreter}]\!](\texttt{p}, \texttt{d})$.


Interpreter **Program** `p`
The interpreter's data is `p` and `d` together
Data `d`

We have developed a "blob universal machine", i.e., a self-interpreter for the blob formalism that is closely analogous to Turing's original universal machine.

6 Contributions of This Work

We have for the first time investigated the possibility of programmable bio-level computation. The work sketched above, in particular the functioning of blob code, can all be naturally expressed in the form of abstract biochemical reaction rules. Further, we have shown molecular computation to be universal in a very strong sense: not only can every computable function be computed by a blob program, but this can all be done using *a single, fixed, set of reaction rules*: it is not necessary to resort to constructing new rule sets (in essence, new biochemical architectures) in order to solve new problems; it is enough to write new programs.

The new framework provides Turing-completeness efficiently and without asymptotic slowdowns. It seems possible to make a tighter analogy between universality and self-reproduction than by the von Neumann and other cellular automaton approaches.

It should be clear that familiar Computer Science concepts such as interpreters and compilers also make sense also at the biological level, and hold the promise of becoming useful operational and utilitarian tools.

References

1. L. M. Adleman. On constructing a molecular computer. In *DIMACS: series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1996.
2. R. Backofen and P. Clote. Evolution as a computational engine. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic*, pages 35–55. Springer-Verlag, 1996.
3. D. Beaver. Computing with DNA. *Journal of Computational Biology*, 2(1):1–7, 1995.
4. Y. Benenson. Biocomputers: from test tubes to live cells. *Molecular BioSystems*, 5(7):675–685, 2009.
5. Y. Benenson, R. Adar, T. Paz-Elizur, Z. Livneh, and E. Shapiro. DNA molecule provides a computing machine with both data and fuel. In *Proc Natl Acad Sci U S A*, volume 100 of *Lecture Notes in Computer Science*, pages 2191–2196, 2003.
6. L. Cardelli and G. Zavattaro. On the computational power of biochemistry. In *AB '08: Proceedings of the 3rd international conference on Algebraic Biology*, pages 65–80, Berlin, Heidelberg, 2008. Springer-Verlag.
7. L. Cardelli and G. Zavattaro. Turing universality of the biochemical ground form. *Mathematical Structures in Computer Science*, 19, 2009.
8. P. Chapman. Life universal computer. *http://www.igblan.free-online.co.uk/igblan/ca/*, (November), 2002.
9. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.
10. A. Danchin. Bacteria as computers making computers. *FEMS Microbiology Reviews*, 33(1):3 – 26, 2008.

11. V. Danos, J. Feret, W. Fontana, and J. Krivine. Abstract interpretation of cellular signalling networks. In *VMCAI*, volume 4905 of *VMCAI, Lecture Notes in Computer Science*, pages 83–97, 2008.
12. V. Danos and C. Laneve. Formal molecular biology. *Theor. Comp. Science*, 325:69 – 110, 2004.
13. P. Degano and R. Gorrieri, editors. *Computational Methods in Systems Biology, 7th International Conference, CMSB 2009, Bologna, Italy, August 31-September 1, 2009. Proceedings*, volume 5688 of *Lecture Notes in Computer Science*. Springer, 2009.
14. G. Delzanno, C. D. Giusto, M. Gabbrielli, C. Laneve, and G. Zavattaro. The *kappa*-lattice: Decidability boundaries for qualitative analysis in biological languages. In Degano and Gorrieri [13], pages 158–172.
15. M. Gardner. Mathematical recreations. *Scientific American*, October 1970.
16. M. L. Guerriero, D. Prandi, C. Priami, and P. Quaglia. Process calculi abstractions for biology. Technical report, University of Trento, Italy, Jan. 01, 2006.
17. M. Hagiya. Designing chemical and biological systems. *New Generation Comput.*, 26(3):295, 2008.
18. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice Hall, 1993.
19. N. D. Jones. *Computability and complexity: from a programming perspective*. MIT Press, Cambridge, MA, USA, 1997.
20. L. Kari. Biological computation: How does nature compute? Technical report, University of Western Ontario, 2009.
21. L. Kari and G. Rozenberg. The many facets of natural computing. *Commun. ACM*, 51(10):72–83, 2008.
22. M. Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.
23. C. L. Nehaniv. Asynchronous automata networks can emulate any synchronous automata network. *International Journal of Algebra and Computation*, 14(5-6):719–739, 2004.
24. T. Ran, S. Kaplan, and E. Shapiro. Molecular implementation of simple logic programs. *Nat Nano*, 4(10):642–648, Oct 2009.
25. E. Shapiro. Mechanical Turing machine: Blueprint for a biomolecular computer. Technical report, Weizmann Institute of Science, 1999.
26. E. Shapiro and Y. Benenson. Bringing DNA computers to life. *Scientific American*, 294:44–51, 2006.
27. C. Talcott. Pathway logic. In *Formal Methods for Computational Systems Biology*, volume 5016 of *LNCS*, pages 21–53. Springer, 2008. 8th International School on Formal Methods for the Design of Computer, Communication, and Software Systems.
28. A. Turing. On computable numbers with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936-7.
29. J. von Neumann and A. W. Burks. *Theory of Self-Reproducing Automata*. Univ. Illinois Press, 1966.
30. E. Winfree. Toward molecular programming with DNA. *SIGOPS Oper. Syst. Rev.*, 42(2):1–1, 2008.
31. E. Winfree, X. Yang, and N. C. Seeman. Universal computation via self-assembly of DNA: Some theory and experiments. In *DNA Based Computers II, volume 44 of DIMACS*, pages 191–213. American Mathematical Society, 1996.
32. S. Wolfram. *A New Kind of Science*. Wolfram Media, January 2002.

33. P. Yin, A. J. Turberfield, and J. H. Reif. Design of an autonomous dna nanomechanical device capable of universal computation and universal translational motion. In *Tenth International Meeting on DNA Based Computers (DNA10)*, volume 3384 of *Lecture Notes in Computer Science*, pages 426–444, 2005.

A More on Turing completeness

How to show Turing completeness of a computation framework. This is typically shown by *reduction* from another problem already known to be Turing complete. Notation: let L and M denote languages (biological, programming, whatever), and let $[\![p]\!]^L$ denote the result of executing L-program p, for example an input-output function computed by p. Then we can say that language M is at least as powerful as L if

$$\forall p \in L\text{−programs } \exists q \in M\text{−programs } (\ [\![p]\!]^L = [\![q]\!]^M\ )$$

A popular choice is to let L be some very small Turing complete language, for instance Minsky register machines or two-counter machines (2CM). The next step is to let M be a biomolecular system of the sort being studied. The technical trick is to argue that, given any L-instance of (say) a 2CM program, it is possible to construct a biomolecular M-system that faithfully simulates the given 2CM.

Oddly enough, Turing completeness is not often used to show that certain problems *can* be solved by M-programs; but rather only to show that, say, the *equivalence* or *termination problems of M-programs are algorithmically undecidable* because they are undecidable for L, and the properties are preserved under the construction. This discussion brings up a central issue:

Simulation as opposed to interpretation. Arguments to show Turing completeness are (as just described) usually by *simulation*: for each problem instance (say a 2CM) one somehow constructs a biomolecular system such that . . . (the system in some sense solves the problem). However, in many papers for each problem instance the construction of the simulator is done by hand, e.g., by the author writing the article. In effect the existential quantifier in $\forall p \exists q ([\![p]\!]^L = [\![q]\!]^M)$ is computed by hand. This phenomenon is clearly visible in papers on cellular computation models: completeness is shown by simulation rather than by interpretation.

In contrast, Turing's original "Universal machine" simulates by means of *interpretation*: a stronger form of imitation, in which the existential quantifier is realised by machine. Turing's "Universal machine" is capable of executing an arbitrary Turing machine program, once that program has been written down on the universal machine's tape in the correct format, and its input data has been provided. Our research follows the same line, applied in a biological context: we show that simulation can be done by general interpretation, rather than by one-problem-at-a-time constructions.

Self-interpretation without asymptotic slowdown. The blob self-interpreter overcomes a limitation that seems built-in to the Turing model. Analysis of its running time reveals that the time taken to interpret one blob instruction is bounded by a constant that is *independent of the program being interpreted.* Intuitively, there are two reasons for this: First, there are no variables, pointers, or other features that add program-dependent amounts of time to the central self-interpretion loop. Second, the fact that every transfer of control in the interpreted program is to an adjacent program blob means that no program-dependent amount of time is spent on fetching the next instruction.

One consequence is that *constant time factors do matter*: the "linear hierarchy" results developed in [19] (Section 19.3 for the I language) also hold for the blob language. (The linear hierarchy result sounds intuitively obvious and natural, but in fact does not hold for many of the traditional models of computation, in particular does not hold for Turing machines with arbitrarily large alphabets or number of tapes.)

B Turing completeness of the blob model

We prove that any one-tape Turing machine with a single read/write head may be simulated by a blob program. The tape contents are always finite and enclosed between a left endmarker $\lhd$ and a right endmarker $\rhd$.

B.1 Turing machine syntax

A Turing machine is a tuple $Z = (\{0,1\}, Q, \delta, q_{start}, q_{halt})$. The tape and input alphabet are $\{0,1\}$. (Blanks are not included, but may be encoded suitably by bits.) Q is a finite set of *control states* including distinct start and halting states $q_{start}, q_{halt} \in Q$. The *transition function* has type

$$\delta : \{0, 1, \lhd, \rhd\} \times Q \to \mathcal{A} \times Q$$

where an *action* is any $A \in \mathcal{A} = \{L, R, W0, W1\}$. Notation: we write a Turing machine instruction as

$$\delta(q, b) \to (A, r)$$

meaning "In state q, reading bit b, perform action A and move to state r". Actions $L, R, W0, W1$ mean informally "move Left, move Right, Write 0, Write 1", respectively. For simplicity we assume that Turing machines may not both move and write on the tape in the same atomic step. (A "write-and-move" action may easily be implemented using two states and two steps.)

We also assume that every Turing machine satisfies the following *consistency assumptions*:

- If $\delta(q, \lhd) \to (A, r)$ is an instruction, then $A \in \{R\}$ (i.e. the machine never moves to the left of the left endmarker and cannot overwrite the endmarker).
- If $\delta(q, \rhd) \to (A, r)$ then $A \in \{L, W0, W1\}$ (i.e. the machine never moves to the right of the right endmarker, but *can* overwrite the endmarker).

Definition 1. *Let M be a Turing machine. The* state graph *of M is the directed graph where the nodes are the states of M and there is a directed edge from q to r annotated (b, A) if there is an instruction $\delta(q, b) \to (A, r)$.*

B.2 Turing machine semantics

A total state has the form

$$\lhd\ b_1 \dots \overset{q}{b_i} \dots b_n\ \rhd$$

where the b_j are tape symbols, and q is a control state. We define the *tape contents* of the machine to be everything enclosed between $\lhd$ and $\rhd$.

The Turing machine defines a one-step transition relation between total states in the expected way (not spelled out here). Tapes may only grow to the right, not the left. We assume that if there is an instruction of the form $\delta(q, \rhd) \to (W0, r)$ or $\delta(q, \rhd) \to (W1, r)$ (i.e. the right endmarker is overwritten), then the tape is automatically extended to the right with a new endmarker to the immediate right of the previous endmarker.

Remark: the tape contents will always be finite after a finite number of computation steps.

Input/Output: A Turing machine Z computes a partial function

$$[\![Z]\!] : \{0,1\}^* \rightharpoonup \{0,1\}^*$$

- *Input*: The machine is in its start state with the tape head on the tape cell to the *immediate right* of the left endmarker $\lhd$. The input is the contents of the tape.
- *Output*: The machine is in its halt state. The output is the contents of the tape.

B.3 Compiling a Turing machine into a blob program

We describe a way to compile any Turing machine $Z = (\{0,1\}, Q, \delta, q_{start}, q_{halt})$ into blob program code *code*(Z) that simulates it. Compilation of a Turing machine into blob code is as follows:

- Generate blob code for each instruction $\delta(q, b) \to (A, r)$.
- Collect blob code for all the states into a single blob program.

Before describing the compilation algorithm, we explain how the blob code realises a step-by-step simulation of the Turing machine Z.

Turing machine representation by blobs At any time t in its computation, the Turing machine's tape $b_1 \dots b_i \dots b_n$ will represented by a finite sequence $B_1 \dots B_i \dots B_n$ of blobs. If at time t the Turing machine head is scanning tape symbol b_i, the active data blob will be the blob B_i. Arrangement: each B_i is linked to its predecessor via bond site 1, and to its successor via bond site 2.

The Turing machine's control state will correspond to the active program blob in *code*(Z).

The cargo bits of the "data blobs" are used to indicate the contents of the the tape cell:

- Cargo bit 0 is unused in the simulation.
- Cargo bit 1 is used to hold the bit occupied by the tape cell (if the blob represents either $\lhd$ or $\rhd$, the contents of cargo bit 1 is irrelevant).
- Cargo bit 2 is '1' iff the blob represents the *left* endmarker $\lhd$.
- Cargo bit 3 is '1' iff the blob represents the *right* endmarker $\rhd$.

Syntax of the generated code We will write the generated blob target program as straightline code with labels. For every instruction, the "next" blob code instruction to be executed is the one linked to the active program blob by the latter's "successor" bond site 2. Thus, in

```
SCG 0 5
EXT
```

the blob corresponding to `SCG 0 5` has its bond site 2 linked to the "predecessor" bond site 1 of the blob corresponding to `EXT`.

Code generation for each state Let $q \neq q_{halt}$ be a state. The four possible kinds of transitions on state q are:

$$\begin{array}{c}\delta(q,0) \rightarrow (A0,q0)\\ \delta(q,1) \rightarrow (A1,q1)\\ \delta(q,\lhd) \rightarrow (AL,qL)\\ \delta(q,\rhd) \rightarrow (AR,qR)\end{array}$$

where $q0,q1,qL,qR \in Q$, $A0,A1 \in \{L,R,W0,W1\}$, and $AL,AR \in \{L,W0,W1\}$.

We generate code for q as follows. For typographical reasons, $\lhd$ = EL and $\rhd$ = ER. The action code notations `[A0]` etc, is explained below, as is the label notation `<label>`. The initial `FIN` code may be safely ignored on the first reading.

```
Generate i-1 FIN  // Assume program port 2 is always "next" operation
                  // Each FIN is labeled as noted below
                  // The last FIN is bound (on its bond site 2) to
                  // the blob labeled 'Q' below.

Q:   JCG  2  QLE  // If 1, We're at left tape end
                  // By convention, bond site 3 of the APB is
                  // bound to the blob labeled QLE
     JCG  3  QRE  // If 1, We're at right tape end
     JCG  1  Q1   // We're not at any end. If '0' is scanned, move along
                  // (on bond site 2),
                  // otherwise a '1' is scanned, jump to Q1
```

```
                    // (on bond site 3)
     [A0]           // Insert code for action A0
     FIN qA0q0      // Go to appropriate fanin before q0 (on bond site 2)

Q1:  [A1]           // Insert code for action A1
     FIN qA1q1      // Go to appropriate fanin before q1 (on bond site 2)

QLE: [AL]           // Insert code for AL
     FIN qELALqL    // Go to appropriate fanin before qL (on bond site 2)

QRE: R[AR]          // Insert code for AR (with the R[ ]-function)
     FIN ERARqR     // Go to appropriate fanin before qR (on bond site 2)

                    // Code for q end
```

Code for q_{halt}:

```
Generate i-1 FIN  // Assume program port 2 is "next" operation always
                  // Each FIN is labeled as noted below
                  // The last FIN is bound (on its bond site 2) to
                  // the blob labeled 'Qh' below.

Qh: EXT
```

The JCG instructions test the data blob B_i to see which of the four possible kinds of transitions should be applied. Codes [A0], [A1], [AL], R[AR] simulate the effect of the transition, and the FIN after each in effect does a "go to" to the blob code for the Turing machine's next state. (This is made trickier by the fan-in restrictions, see Section B.3 below.)

Two auxiliary functions We use two auxiliary functions to generate code:

$$\texttt{[]} : \{L, R, W0, W1\} \longrightarrow \text{blobcode}$$

and

$$\texttt{R[]} : \{L, W0, W1\} \longrightarrow \text{blobcode}$$

Function [] is used for code generation on arbitrary tape cells, and R[] for code generation when the Turing machine head is on the right end marker where some housekeeping chores must be performed due to tape extension.

Code generation for instructions not affecting the right end of the tape

[W0]

```
SCG 0 1           // Set tape cell content to 0
```

[W1]

```
SCG 1 1         // Set tape cell content to 1
```

$[L]$

```
CHD  1          // Set ADB to previous blob (move tape left)
```

[R]

```
CHD  2          // Set ADB to next blob (move tape right)
```

Code generation for instructions that can extend the tape

R[W0]

```
 SCG  0 3       // Current blob is no longer at right tape end
 INS  2 1       // Insert new blob at bond port 2 on ADB
                // (new tape cell). New blob is bound at site 1.
 CHD  2         // Change ADB to new blob      (move head right)
 SCG  1 3       // New blob is at the right end of the tape
 CHD  1         // Change ADB to original blob (move head left)
 SCG  0 1       // Write a '0' in the tape cell (as per W0).
```

R[W1]

```
 SCG  0 3       // Current blob is no longer at right tape end
 INS  2 1       // Insert new blob at bond port 2 on ADB
                // (new tape cell). New blob is bound at site 1
 CHD  2         // Change ADB to new blob      (move head right)
 SCG  1 3       // New blob is right tape end
 CHD  1         // Change ADB to original blob (move head left)
 SCG  1 1       // Write a '1' in the tape cell (as per W1)
```

R[L]

```
R[L] = [L]      // Move to the left
        // TM does not move right at right tape end.
```

Control flow in the generated blob code A technical problem in code generation. We now explain the meaning of the somewhat cryptical comments such as "`Go to appropriate fanin before q1`" in Section B.3, and notations such as `qA0q0`.

The problem: while a pointer-oriented language allows an unbounded number of pointers into the same memory cell, this is not true for the blob structures (the reason is that a bond is intended to model a chemical connection between two molecules). This is a “fan-in” restriction on program (and data) syntax.

A consequence: blob program code may not contain more than one control transfer to a given instruction, unless this is done by a bond site different from the usual “predecessor” site 1. The purpose of the instruction `FIN` is to allow *two* entry points: one as usual by bond site 1, and a second by bond site 3.

The initial `FIN` code generated of Section B.3. This concerns the entry points into blob code for a Turing state q. Let i be the number of directed edges to q in the state graph (i.e., the number of “go to’s” to q).

If $i \leq 1$, we generate no fanin blobs.

Otherwise, we generate $i - 1$ fanin blobs before the code generated for q; these handle the i transitions to q. The blobs bound to the fanin nodes occur in the code generated for other states (perhaps from q to itself). For each transition $\delta(q', b) \rightarrow (A, q)$, a blob in the code generated for q' is bound to a single fanin blob for q. The fanin blob generated above, before the generated code for state q, is labeled by `q’bAq`.

C Dimensionality limitations

Limitation to three dimensions. The physical world imposes a dimensionality requirement we have not yet addressed: data and program code cannot be packed with a density greater than that allowed by three-dimensional Euclidean space. The idea of a biologically plausible computing model that must work in 3-space provokes several interesting questions.

Realisability in 3-space: In the blob model, following a chain of k bonds from the active data blob (at any time in a computation) should give access to at most $O(k^3)$ blobs. This is not guaranteed by the blob model as presented above; indeed, a blob program could build a complete 3-ary tree of depth k and containing 3^k blobs at distance k. This structure could not be represented in 3-space with our restrictions, and still have the intended semantic structure: that any two blobs linked by a bond should be adjacent in the biological “soup”.

On dimensional limits in other computation models. The usual Turing machine has a fixed number of 1-dimensional tapes (though k-dimensional versions exist, for fixed k). Cellular automata as in [29,8,32] have a fixed 2-dimensional architecture. Dimensionality questions are not relevant to Minsky-style machines with a fixed number of registers, e.g., the two-counter machine.

Machines that allow computed addresses and indirection, e.g., the RAM, RASP, etc., have no dimensionality limitations at all, just as in the “raw” blob model: traversing a chain of k bonds from one memory can give access to a number of cells exponential in k (or higher if indexing is allowed).

3D complexity classes? The well-known and well-developed Turing-based computational complexity theory starts by restricted programs’ running time and/or space. An possible analogy would be to limit the dimensionality of the data structures that a program may build during a computation.

Pursuing the analogy, the much-studied complexity class PTIME is quite large, indeed so large that dimensionality makes no difference: on any traditional model where data dimensionality makes sense, it would be an easy exercise to show that PTIME = PTIME3D. What if instead we study the class LINTIME of problems solvable in linear time (as a function of input size)? Alas, this smaller, realistically motivated class is not very robust for Turing machines, as small differences in Turing models can give different versions of LINTIME (Sections 18, 19, 25.6 in [19]). It seems likely though that the LINTIME class for blob machines is considerably more robust.

Conjecture: LINTIME3D $\subsetneq$ LINTIME on the blob model.

Another interesting question: does self-interpretation cause a need for higher dimensionality? We conjecture that this is not so for any one fixed interpreted program; but that diagonalisation constructions can force the necessary dimensionality to increase. This appears to be an excellent direction for future work.

Approaches to Supercompilation

Simon Peyton Jones

simonpj@microsoft.com
Microsoft Research Ltd,
Cambridge, England

Abstract. My student Max Bolingbroke and I have been studying supercompilation, with a view to making GHC into a supercompiler. In this talk I'll describe various approaches that we have explored, and focus on the one we are pursuing right now. A well-known practical challenge in supercompilation is over-specialisation and code explosion. I will present some ideas we have developed to tackle this problem.

Preliminary Report on a Self-Applicable Online Partial Evaluator for Flowchart

Robert Glück [⋆]

DIKU, Dept. of Computer Science, University of Copenhagen,
DK-2100 Copenhagen, Denmark

Abstract. This is a preliminary report on a self-applicable online partial evaluator for a flowchart language with recursive calls. Self-application of the partial evaluator yields generating extensions that are as efficient as those reported in the literature for offline partial evaluation. This result is remarkable because partial evaluation folklore has indicated that online partial evaluation techniques unavoidably lead to overgeneralized generating extensions. The purpose of this paper is not to argue which line of partial evaluation is better, but to show how the problem can be solved by recursive polyvariant specialization. The online partial evaluator, its techniques and implementation, are presented in a complete way. Full self-application according to the Futamura projections is demonstrated.

1 Introduction

This paper reports on the design and implementation of a self-applicable online partial evaluator for a flowchart language with recursive calls. The partial evaluator does not require partial evaluation techniques that are stronger than those already known, but another organization of the algorithm. This result is remarkable because partial evaluation folklore has indicated that online techniques unavoidably lead to overgeneralized generating extensions [13, Ch. 7]. Offline partial evaluation was invented specifically to solve the problem of self-application [14]. The purpose of this investigation is not to argue which line of partial evaluation is better, but to show how the problem can be solved. Self-application of the online partial evaluator converts interpreters into compilers and produces a self-generating compiler generator, all of which are as efficient as those known from the literature on offline partial evaluation (*e.g.*, [10,13,14,17]).

The offline partial evaluator `mix` for a flowchart language described by Gomard and Jones [10] is well suited as the basis for the online partial evaluator because their partial evaluator does not follow the binding-time annotations of a subject program, but bases its decisions whether to interpret or residualize flowchart commands on a division of the program variables into static and dynamic ones, which was precomputed by a monovariant binding-time analysis. Another important advantage is that partial evaluation for flowchart languages

⋆ Part of this work was performed while the author was visiting the National Institute of Informatics (NII), Tokyo.

$p ::= \texttt{((}x^*\texttt{) (}l\texttt{) (}b^+\texttt{))}$	(program)
$b ::= \texttt{(}l\ a^*\ j\texttt{)}$	(basic block)
$a ::= \texttt{(}x\ \texttt{:=}\ e\texttt{)}$	(assignment)
$\quad \mid\ \texttt{(}x\ \texttt{:= call}\ l\texttt{)}$	(call)
$j ::= \texttt{(goto}\ l\texttt{)}$	(unconditional jump)
$\quad \mid\ \texttt{(if}\ e\ l\ l\texttt{)}$	(conditional jump)
$\quad \mid\ \texttt{(return}\ e\texttt{)}$	(return)
$e ::= \texttt{(}o\ u^*\texttt{)}$	(simple expression)
$o ::= \texttt{hd} \mid \texttt{tl} \mid \texttt{cons} \mid \texttt{+} \mid \texttt{-} \mid \texttt{=} \mid \texttt{<} \mid \ldots$	(primitive operator)
$u ::= x \mid \texttt{'}v$	(operator argument)
$x \in \mathit{Name} \quad v \in \mathit{Value} \quad l \in \mathit{Label}$	

Fig. 1. Scheme representation of Flowchart programs with recursive calls.

Call

$$\frac{\sigma \vdash_{block} \Gamma(l) \Rightarrow (\langle \mathsf{halt}, v\rangle, \sigma')}{\sigma \vdash_{assign} x\ \texttt{:= call}\ l\ \Rightarrow\ \sigma[x \mapsto v]}$$

Fig. 2. Inference rule extending Hatcliff's operational semantics of `FCL` [11, Fig. 6]

has been very well documented (*e.g.*, [2–4, 10, 11, 13]), which should make our results easily accessible and comparable.

Turning `mix` into an online partial evaluator required two modifications: (1) the division of the program variables is maintained as an updatable set of variable names at specialization time, and (2) the partial evaluator is rewritten to perform *recursive polyvariant specialization* [7] instead of the usual iterative version with an accumulating parameter (pending list). This required an extension of the flowchart language with a simple recursion mechanism. Full self-application according to the Futamura projections is demonstrated by converting an interpreter for Turing-machines [10] into a compiler, a universal parser for regular languages [2] into a parser generator, and the partial evaluator itself into a compiler generator. Self-application of the online partial evaluator can also generate generating extensions that are more optimizing than those produced by the original `mix`. The generating extension of the Ackermann function can specialize and precompute the function at program generation time, thereby producing fully optimized residual programs.

Throughout this paper, we assume that readers are familiar with the basics of partial evaluation, *e.g.*, as presented by Jones *et al.* [13, Part II]. Use is made as much as possible of existing partial evaluation techniques for flowchart languages [10,11] to focus the attention on essential differences, instead of irrelevant details.

```
((m n) (ack)
 ((ack  (if (= m 0) done next))
  (next (if (= n 0) ack0 ack1))
  (done (return (+ n 1)))
  (ack0 (n := 1)
        (goto ack2))
  (ack1 (n := (- n 1))
        (n := (call ack m n))
        (goto ack2))
  (ack2 (m := (- m 1))
        (n := (call ack m n))
        (return n)) ))
```

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } n=0 \\ A(m-1,A(m,n-1)) & \text{otherwise} \end{cases}$$

Fig. 3. Ackermann program and its function

2 A Simple Imperative Language with Recursive Calls

Flowchart is a simple imperative language [10, 11] with variables, assignments, and jumps (Fig. 1). A program consists of a sequence of labeled basic blocks. As is customary, the set of values and labels is that of the Lisp S-expressions. The operational semantics of the language is identical to the one that has been formalized and published by Hatcliff [11], except that we add a simple command for calling blocks:

x `:= call` l

The command executes the block labeled l in a copy of the current store σ and assigns the return value to the variable x in the original store σ. (In an actual implementation this only requires copying the variables that are *live* at the entry of block l.) The command allows recursive calls, has no side-effects, and requires only one additional inference rule in Hatcliff's operational semantics [11, Fig. 6]. The inference rule in Fig. 2 is parameterized with respect to Γ, a partial function that maps labels to blocks in a program, and updates the store σ at x with the return value v, which is marked by halt. Due to its simplicity, the command is easy to interpret and specialize. For the sake of brevity, we refer to the extended language as Flowchart. The syntax can easily be generalized to calls with argument expressions.

An implementation of the Ackermann function using the call command is shown in Fig. 3. For the sake of readability, we annotate each call with the variables that are live at the called block and write integer constants without quotes. The program is defined recursively, takes the non-negative integers `m` and `n` as input, and starts program execution at block `ack`. The double recursion on both `m` and `n` cannot be expressed in terms of primitive recursion.

For the sake of conciseness we shall write many of the programs in pseudocode using constructs such as while ... do and pattern matching case ... of, which are to be regarded as structured ways of writing Flowchart programs. Fig. 4 shows

```
procedure evalpp :: pp × vs × program → value
  block := lookup(pp,program);
  while block is not empty do
  begin command := first-command(block); block := rest(block);
     case command of
     x := rhs:      case rhs of
                    call pp: vs := vs[x ↦ call evalpp(pp,vs,program)];
                    e:       vs := vs[x ↦ evalexp(e,vs)];
     if e pp' pp'': if evalexp(e,vs) = true
                    then block := lookup(pp', program)
                    else  block := lookup(pp'',program);
     goto pp:       block := lookup(pp,program);
     return e:      value := evalexp(e,vs);
  end;
  return value
```

Fig. 4. A self-interpreter for Flowchart written in pseudocode

the *self-interpreter* evalpp for Flowchart written in pseudocode. It inputs a label pp, a store vs, and a program program, and returns the value of the program. Fragments of the interpreted program are written in *italics* for clarity. The self-interpreter makes liberal use of primitive operators such as lookup for finding a block pp in a program and evalexp for evaluating an expression e in a store vs. Note the simplicity of interpreting a call by recursively calling the self-interpreter.

3 Online Partial Evaluation Techniques

A partial evaluator for Flowchart takes values for the static parameters of a program and tries to precompute as many commands as possible. Program code is generated for commands that cannot be precomputed (they are residualized). A specialization strategy is said to be *online* if the values computed at the time of specialization can affect the choice of action taken; otherwise a strategy is said to be *offline* [13, Ch. 4.4.7]. The main advantage of an online strategy is that it can exploit static information that becomes available at specialization time, while an offline strategy bases its decisions on the results of a *binding-time analysis* (BTA) performed before specialization.

We now explain the specialization principles behind the online partial evaluator for Flowchart and in which way it differs from the offline partial evaluator.

3.1 Generating Code for Commands

Fig. 5 shows the specialization of an assignment x := e by an *offline* partial evaluator. The binding time of variable x determines whether the assignment is interpreted or residualized. If x is dynamic, e is reduced to e' by constant folding using the static values in the store vs and an assignment x := e' is generated. If x is static, e is evaluated in vs and the value of x in vs is updated. The congruence

Command	*Done at specialization time*	*Generated code*
$x := e$ (if x is dynamic)	$e' :=$ reduce(e,vs,division)	$x := e'$
$x :=$ *call pp* (if x is dynamic)		$x :=$ *call (pp*,vs*)*
$x := e$ (if x is static)	vs := vs[$x \mapsto$ evalexp(e,vs)]	
$x :=$ *call pp* (if x is static)	vs := vs[$x \mapsto$ **call** evalpp(*pp*,vs,program)]	

Fig. 5. Offline code generation for assignments

Command	*Done at specialization time*	*Generated code*
$x := e$ (if e is dynamic)	$e' :=$ reduce(e,vs,division); division := division $\setminus \{x\}$	$x := e'$
$x :=$ *call pp* (if a var live at *pp* is dynamic)	division := division $\setminus \{x\}$	$x :=$ *call (pp*,vs*)*

$x := e$
(if e is static)
	vs := vs[$x \mapsto$ evalexp(e,vs)];
	division := division $\cup \{x\}$	

$x :=$ *call pp*
(if all vars live at *pp* are static)
	vs := vs[$x \mapsto$ **call** evalpp(*pp*,vs,program)];
	division := division $\cup \{x\}$	

Fig. 6. Online code generation for assignments

Command	*Done at specialization time*	*Generated code*
if e pp' pp" (if e is dynamic)	$e' :=$ reduce(e,vs,division)	*if e' (pp',vs) (pp",vs)*
if e pp' pp" (if e is static and value = true)	value := evalexp(e,vs)	*goto (pp'*,vs*)*
if e pp' pp" (if e is static and value = false)	value := evalexp(e,vs)	*goto (pp"*,vs*)*
goto pp		*goto (pp*,vs*)*
return e	$e' :=$ reduce(e,vs,division);	*return e'*

Fig. 7. Online and offline code generation for control flow (no transition compression)

of the binding times calculated by the BTA guarantees that e can be evaluated in vs whenever x is static [12]. In the case of `mix`, which uses a monovariant BTA, the same division is valid at all program points. The division can be represented by a set division that contains the names of all static variables. Variables that are not in this set are dynamic.

Likewise, x := call pp is interpreted or residualized depending on the binding time of x. If x is dynamic, the call is residualized as x := call (pp,vs) and later a residual block (pp,vs), that is block pp specialized with respect to vs, is generated. Otherwise, the call is static and interpreted by the self-interpreter evalpp (Fig. 4).

In an *online* partial evaluator the division is not known until specialization time. Thus, instead of using the binding time of x in an assignment x := e, the binding time of e determines whether the assignment is interpreted or residualized. The binding time of x is changed accordingly by adding it to or removing it from the division. Fig. 5 shows the partial evaluation of an assignment by an online partial evaluator. The binding times of variables are determined and propagated at specialization time and may affect the course of partial evaluation.

Similarly for x := call pp, except that the binding times of all variables that are *live* at pp determine whether the assignment is interpreted or residualized. If all live variables are static, the call is interpreted in vs and the return value is assigned to x. Otherwise, an assignment x := call (pp,vs) is generated. The division is updated according to the binding time of x.

In an online partial evaluator the binding time of a variable x on the left-hand side of an assignment depends on the binding time of the expression or the call on the right-hand side. Depending on the outcome, the division is updated at specialization time. The division is not predetermined and can change during partial evaluation.

Control flow commands Code generation without transition compression is shown in Fig. 7. The partial evaluation of control flow commands is the same in online and offline partial evaluation. However, as a consequence of the online treatment of assignments, the decision taken by an online partial evaluator, namely whether to residualize a conditional or to select one of the branches (pp', pp''), depends on the actual static values obtained from the assignments preceding the conditional. Even though the only visible difference between code generation in an online and offline partial evaluator is confined to the handling of assignments (Fig. 5 *vs.* Fig. 6), the process of partial evaluation proceeds quite differently.

Transition compression Residual programs produced by the code generation described above often contain chains of trivial transitions (blocks consisting only of goto commands), which makes them less readable. Transition compression can be performed during partial evaluation by continuing code generation directly with the commands in a block pp instead of generating goto pp (Fig. 7). Transition compression can also be performed after partial evaluation in a post-processing phase. The choice of the transition compression strategy does not affect the specialization strength or the self-applicability of the partial evaluators, only the size of the residual programs which they generate.

3.2 Two Simple Polyvariant Specialization Algorithms

Program point specialization incorporates the values of the static variables into a program point. In *polyvariant block specialization* [2] each block in a subject

```
procedure polyloop :: pp × vs → code
  pend := {(pp,vs)}; done := {}; code := {};
  while pend is not empty do
  begin
    pick (pp,vs) ∈ pend;
    code := code ∪ generate residual block for pp using the values in vs;
    done := done ∪ {(pp,vs)}; pend := (pend ∪ successors(pp,vs)) \ done
  end;
  return code
```

Fig. 8. A simple iterative specialization algorithm polyloop [10]

```
procedure polyrec :: pp × vs × code → code
  if no residual block labeled (pp,vs) exists in code then
  begin
    code := code ∪ generate residual block for pp using the values in vs;
    let {(pp_1,vs_1), ... , (pp_n,vs_n)} = successors(pp,vs);
    code := call polyrec(pp_1,vs_1,code);
    ...
    code := call polyrec(pp_n,vs_n,code)
  end;
  return code
```

Fig. 9. A simple recursive specialization algorithm polyrec [7]

program may be specialized with respect to several different static stores. A residual block labeled (pp,vs) is the concatenation of the code generated for the commands in block pp using the values in vs. Polyvariant block specialization can be implemented in two different ways. Traditionally, an iterative method is used that maintains a set of pending and done specialization tasks. We shall see that a recursive method enables self-application of the online partial evaluator.

1. The *iterative method* in Fig. 8 maintains two sets, pend and done, to keep track of the pairs (pp,vs) that are pending specialization and those for which a residual block was already generated. Block specialization is repeated by the while-loop until pend is empty. After generating a residual block, the set of successor blocks, successors(pp,ss) = $\{(\mathsf{pp}_1,\mathsf{vs}_1), ..., (\mathsf{pp}_n,\mathsf{vs}_n)\}$, that is all blocks that occur in conditionals and calls of the residual block (pp,vs), are added to pend as new specialization tasks, unless they are already in done. The residual blocks are collected in a set code and returned as the final result. Iterative block specialization is invoked by **call** polyloop $(\mathsf{pp}_0,\mathsf{vs}_0)$, where pp_0 is the initial label of the subject program and vs_0 is the initial static store.
2. The *recursive method* in Fig. 9 performs block specialization in a depth-first manner without maintaining a set pend. The successor blocks are specialized *immediately* after the specialization of a residual block is completed. The set code of generated residual blocks doubles as set done. A block specialization

(pp,vs) is only performed if the residual block for (pp,vs) does not yet exist in code. Set code is single-threaded through the recursive calls to avoid repeating the generation of the same residual block. Consider as an example the specialization of the two successors blocks (pp_1,vs) and (pp_2,vs) that occur in a residual conditional if e (pp_1,vs) (pp_2,vs):

```
code := call polyrec(pp₁,vs,code);
code := call polyrec(pp₂,vs,code)
```

Recursive block specialization is invoked by **call** polyrec (pp_0,vs_0,{}), where pp_0 is the initial label, vs_0 is the initial static store, and {} is the initially empty set of residual blocks.

While the iterative version makes use of a data structure (pend) to keep track of the specialization tasks, the recursive version relies on the call stack of the implementation language. The iterative version corresponds to a tail-recursive function where pend is an accumulating parameter. Functions with accumulating parameters are notorious for being difficult to specialize.

3.3 Specializing the Simple Specialization Algorithms

The challenge of self-application is the specialization of the partial evaluation algorithm by itself with respect to a known subject program and a known division, but unknown values for the static store.

Consider first the iterative version. The set pend contains pairs of the form (pp,vs) where pp is part of the subject program to which the partial evaluator is specialized and vs is unknown. As a result, pend becomes dynamic. In offline partial evaluation this problem was solved by precomputing the static set and using a binding-time improvement, called "The Trick" [13], because the static set contains finitely many components (pp,names-in-vs). The lookup of a block pp in the subject program is implemented such that the dynamic pp is compared to all possible values it can assume and specializes the block to all possible outcomes. This trick is necessary to avoid generating trivial generating extensions in which no specialization is performed. The set is precomputed by a BTA.

Now, consider the recursive version. The problem of losing the control information does not arise because there is no set pend in which the program points pp get dynamized. This depth-first specialization method enables the information to be propagated statically and accurately, even in the case of self-application. This is the method that we choose for the implementation of the self-applicable online partial evaluator. It has the additional advantage that it can be used regardless of whether code generation for commands is online or offline. It can thus be used in self-applicable online and offline partial evaluators.

4 An Algorithm for Online Partial Evaluation

We now present the complete algorithm for online partial evaluation based on the specialization techniques described above. It performs code generation with

```
procedure onmix :: program × division × vs → code
 pp   := initial-label(program);
 code := make-header(varlist(program,division),pp,vs);
 return call pepoly(pp,vs,division,program,code)

procedure pepoly :: pp × vs × division × program × code → code
 if ¬done(pp,vs,code)
 then begin
   code  := new-block(code,pp,vs);
   block := lookup(pp,program);
   while block is not empty do
   begin command := first-command(block); block := rest(block);
     case command of
     x := rhs:     if static(vars(rhs,program),division)
                   then begin
                          case rhs of
                            call pp: vs := vs[x ↦ call evalpp(pp,vs,program)];
                            e:       vs := vs[x ↦ evalexp(e,vs)];
                          division := division ∪ {x} end
                   else begin
                         case rhs of
                           call pp: code := call pepoly(pp,vs,division,program,code);
                                    code := add-to(code,make-asg(x,make-call(pp,vs)));
                           e:       code := add-to(code,make-asg(x,reduce(e,vs,division)));
                         division := division \ {x} end;
     if e pp' pp'': if static(e,division)
                    then if evalexp(e,vs) = true
                           then block := lookup(pp', program)
                           else  block := lookup(pp'',program)
                    else begin
                      code := call pepoly(pp', vs,division,program,code);
                      code := call pepoly(pp'',vs,division,program,code);
                      code := add-to(code,make-if(reduce(e,vs,division),(pp',vs),(pp'',vs)))
                   end;
     goto pp:      block := lookup(pp,program);
     return e:     code  := add-to(code,make-return(reduce(e,vs,division)));
   end (* while *)
 end; (* then *)
 return code
```

Fig. 10. The self-applicable algorithm for online partial evaluation of Flowchart

on-the-fly transition compression and recursive polyvariant block specialization. The algorithm is given in pseudocode in Fig. 10.

The input to the main procedure onmix is a subject program program, a division division of the parameters of the subject program, and a store vs containing the values of the static parameters. The residual program code is returned as out-

```
((n) (ack-0)
 ((ack-0  (if (= n 0) ack0-0 ack1-0)) (ack-1  (if (= n 0) ack0-1 ack1-1))
  (ack0-0 (return 3))  ; A(1,1) = 3   (ack0-1 (return 2))  ; A(0,1) = 2
  (ack1-0 (n := (- n 1))              (ack1-1 (n := (- n 1))
          (n := (call ack-0 n))               (n := (call ack-1 n))
          (n := (call ack-1 n))               (n := (call ack-2 n))
          (return n))                         (return n))
                                      (ack-2  (return (+ n 1))) ))
```

Fig. 11. Ackermann program specialized with respect to m=2

put. Some initializations are performed before invoking pepoly, which performs the actual specialization of the blocks in the subject program.

The procedure pepoly implements recursive polyvariant block specialization. Block pp is specialized with respect to vs and division if no residual block exists in code, which is tested by the primitive operator done. If the residual block already exists, code is returned unchanged. Otherwise, the block is fetched from the program by lookup and the header of the new block is added to code. The list of commands is then specialized starting with the first command until it is empty. The while-loop contains a case dispatch which generates code for the commands as described above. The code is generated with transition compression as can be seen in the case of specializing goto pp. For example, see the case of goto pp: instead of generating a residual jump goto (pp,vs), specialization continues at pp. The primitive operation vars(rhs,program) in the assignment case returns e, if rhs = e, or an expression with the variables that are live at pp, if rhs = call pp.

The program in Fig. 11 is an example of a residual program produced by the online partial evaluation algorithm. It was obtained by specializing the Ackermann program with respect to m = 2. No post-optimization was performed on the residual program. Note that m is static throughout the entire Ackermann program (Fig. 3), while variable n, which is initially dynamic, becomes static in block ack0 due to assignment n := 1 and, consequently, the call to ack in ack2 is static and can be precomputed at specialization time. The result are the residual blocks ack0-0 and ack0-1 which return a constant. The same call to ack is partially static if it is reached from ack1 where n is dynamic. The offline partial evaluator mix [10] cannot perform this specialization (this would require a polyvariant expansion of the original Ackermann program based on a polyvariant binding-time analysis before applying mix). A post-optimization of the residual program could replace (call ack-2 n) in ack1-1 by expression (+ n 1).

The online partial evaluator described in this paper has the specialization strength of an offline partial evaluator with a polyvariant binding-time analysis and monovariant return values. This functional equivalence is not surprising because an offline partial evaluator with a maximally polyvariant binding-time analysis can be as accurate as an online partial evaluator [3].

4.1 Live Static Variables

The specialization of blocks with respect to dead static variables can cause considerable code duplication in the residual programs [10]. This is even more critical in an online partial evaluator because the division and the static store can grow and shrink during partial evaluation. It is therefore essential to remove all dead static variables from division and vs each time a new block is looked up in the subject program. At the beginning of onmix before pepoly is called, the set of live variables is determined for each block entry in a subject program and a parameter containing this information is added to pepoly. For readability this parameter and the operations restricting division and vs to the live variables after each lookup were omitted in Fig. 10. However, these cleaning-up operations are crucial for reducing the size of the generated residual program.

4.2 Self-Application

The classification of the three parameters of the main procedure onmix: program and division are static and vs is dynamic. Operations that depend only on program and division can be static, while all other operations that may depend on vs are dynamic. In particular, the parameters pp, division and program remain static and only vs and code are dynamic. The recursive method of polyvariant block specialization keeps this essential information static (pp, division, program), providing the key to successful self-application. Assignments that depend only on static variables are fully evaluated when the partial evaluator us specialized with respect to a subject program and do not occur in the generating extensions produced by self-application. As an example, the important tests static are always static when the online partial evaluator is specialized. They will thus never occur in the generating extensions. Also, a change of the transition compression strategy does not affect the binding-time separation.

5 Specializing the Online Partial Evaluation Algorithm

A classic example is the specialization of a partial evaluator with respect to an interpreter for Turing machines, which yields a compiler from Turing-machine programs to the residual language of the partial evaluator, here Flowchart. We used the same Turing-machine program and the same Turing-machine interpreter[1] written in Flowchart as in publication [10, Fig. 3 and Fig. 4].

The *first Futamura projection* translates the Turing-machine program p into a Flowchart program tar by specializing Turing-machine interpreter int with respect to p by the online partial evaluator onmix:

$$\mathtt{tar} = [\![\mathtt{onmix}]\!]\,\mathtt{(int,p)}. \tag{1}$$

[1] A generalization operator was inserted to change the classification of the variable representing the left-hand side of the tape from static to dynamic at `Left := (GEN '())`.

The program `tar` is identical to the one produced by `fcl-mix` [10, Fig. 5] modulo minor syntactic differences between the flowchart languages.

The *second Futamura projection* yields a compiler `comp` by self-application of `onmix`:

$$\texttt{comp} = [\![\texttt{onmix}]\!]\,\texttt{(onmix, int)}. \qquad (2)$$

The compiler `comp` translates Turing-machine programs into Flowchart. The compiler is as efficient as the one presented in [10, App. II]. Compilation is done recursively instead of iteratively due to the recursive polyvariant specialization used in the specialized `onmix`. This is also an example how the structure of the generated compilers can be influenced by specializing different partial evaluators.

The *third Futamura projection* yields a compiler generator `cogen` by double self-application:

$$\texttt{cogen} = [\![\texttt{onmix}]\!]\,\texttt{(onmix, onmix)}. \qquad (3)$$

The compiler generator is as efficient as the one reported for `mix` [10], except that `cogen` performs recursive polyvariant specialization and produces generating extensions that also perform recursive polyvariant specialization. A compiler generator produced by the third Futamura projection must be *self-generating*, which is a necessary condition for its correctness [6], and so is `cogen`, which produces a textually identical copy of itself when applied to `onmix`:

$$\texttt{cogen} = [\![\texttt{cogen}]\!]\,\texttt{onmix}. \qquad (4)$$

Applying `cogen` to the Ackermann program `ack` yields a generating extension `ackgen`, which produces residual programs of `ack` given a value for `m`, such as the one shown in Fig. 11.

$$\texttt{ackgen} = [\![\texttt{cogen}]\!]\,\texttt{ack}. \qquad (5)$$

Application of `cogen` to Bulyonkov's universal parser for regular languages [2] yields a parser generator `parsegen` that is comparable to the one reported in [7, Fig. 5], if we disregard the fact that the one in this paper is produced by self-application of `onmix` (or by `cogen`) and implemented in Flowchart, while the one produced by quasi-self-application [7] is implemented in Scheme and arity raised by the postprocessor of Unmix [17].

5.1 Overview of Performance

Tables 1 and 2 show some of the preliminary running times for a version of the online specialization algorithm `onmix`. Program `p` and interpreter `int` are the Turing-program and the Turing interpreter [10]. The data for `ack` is $\texttt{m} = 2$ and $\texttt{n} = 3$. The running times are measured AMD Athlon cpu milliseconds using Dr. Scheme version 4.1.3 under Windows XP Home Edition 2002 and include garbage collection, if any. The running times are comparable to those reported in the literature, albeit *all* Turing-related ratios are slightly smaller compared to the results [10, Tab. 2], which were reported for a different hardware/software.

Run		Time	Ratio
`out`	$=$ ⟦`int`⟧ `(p, d)`	78	
	$=$ ⟦`tar`⟧ `d`	16	4.9
`tar`	$=$ ⟦`onmix`⟧ `(int, p)`	172	
	$=$ ⟦`comp`⟧ `p`	47	3.7
`comp`	$=$ ⟦`onmix`⟧ `(onmix, int)`	938	
	$=$ ⟦`cogen`⟧ `int`	547	1.7
`cogen`	$=$ ⟦`onmix`⟧ `(onmix, onmix)`	3016	
	$=$ ⟦`cogen`⟧ `onmix`	907	3.3

Table 1. Turing interpreter

Run		Time	Ratio
`out`	$=$ ⟦`ack`⟧ `(m, n)`	20	
	$=$ ⟦`ackm`⟧ `n`	4	5
`ackm`	$=$ ⟦`onmix`⟧ `(ack, m)`	168	
	$=$ ⟦`ackgen`⟧ `m`	24	7
`ackgen`	$=$ ⟦`onmix`⟧ `(onmix, ack)`	668	
	$=$ ⟦`cogen`⟧ `ack`	496	1.3

Table 2. Ackermann program

6 Related Work

Conventional wisdom holds that only offline partial evaluators using a binding-time analysis can be specialized into efficient generating extensions (*e.g.*, [1] and [13, Ch. 7]). Offline partial evaluation was invented specifically to solve this problem. Mix was the first efficiently self-applicable partial evaluator [14].

The self-applicable partial evaluator presented in this paper makes use of recursive polyvariant specialization [7] to ensure that the information needed for specialization of blocks is not prematurely lost (dynamized) at program generator generation time. An implementation of recursive polyvariant specialization by the author in 1993 is part of the Unmix distribution, an offline partial evaluator for a first-order subset of Scheme [17].

A *higher-order* pending list was used by a breadth-first inverse interpreter to allow good specialization by the offline partial evaluator Similix [8, p. 15], but requires a partial evaluator powerful enough to specialize higher-order values. The self-application of an online partial evaluator for the λ-calculus without polyvariant specialization was reported, but the compilers were of the "overly general" kind [15]. A compromise strategy to self-application of online partial evaluators is a hybrid 'mixline' approach to partial evaluation that distinguishes between static, dynamic, and unknown binding times [13, Ch. 7.2.3] and [19,20]. A notable exception on the self-application of online specializers is V-Mix [5] and [9]. A weaker online specializer was specialized by a stronger one [18].

7 Conclusions and Future Work

We showed that not only offline partial evaluators, but also online partial evaluators can yield generating extensions by self-application that are as efficient as those reported in the literature for offline partial evaluation. It is noteworthy that this did not require partial evaluation techniques that are stronger than those already known today, only a restructuring of the partial evaluator. Although the design of the algorithm is based on a number of existing techniques, their combination in a new and non-trivial way produced this synergetic effect.

Full self-application according to the Futamura projections was demonstrated by implementing a non-trivial online partial evaluator for a flowchart language extended with a simple recursion mechanism. Self-application produced generating extensions whose structure is as "natural and understandable" as in the case of offline partial evaluation [16]. There was no loss of efficiency and no overgeneralization. Self-application of the online partial evaluator can also lead to generating extensions that are more optimizing than those produced by the offline partial evaluators for the flowchart language, such as the generating extension of the Ackermann function. The algorithm for online partial evaluation, the design, techniques and demonstration, were presented in a complete and transparent way. Several attempts have been made previously, including work by the author. We believe that the online partial evaluator for the flowchart language presented in this paper provides the clearest solution to date. It is believed that the techniques presented here can be carried over to other recursive programming languages. It is hoped that this investigation into self-application can be a basis for novel partial evaluators and stronger generating extensions.

Acknowledgements It is a great pleasure to thank Akihiko Takano for providing excellent working conditions to the author at the National Institute of Informatics (NII), Tokyo. Thanks to Kenji Moriyama for help with the program measurements and to Neil Jones and the anonymous reviewers for insightful comments.

References

1. A. Bondorf, N. D. Jones, T. Æ. Mogensen, P. Sestoft. Binding time analysis and the taming of self-application. Research note, DIKU, Dept. of Computer Science, University of Copenhagen, 1988.
2. M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21(5):473–484, 1984.
3. N. H. Christensen, R. Glück. Offline partial evaluation can be as accurate as online partial evaluation. *ACM TOPLAS*, 26(1):191–220, 2004.
4. S. Debois. Imperative-program transformation by instrumented-interpreter specialization. *Higher-Order and Symbolic Computation*, 21(1-2):37–58, 2008.
5. R. Glück. Towards multiple self-application. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 309–320. ACM Press, 1991.

6. R. Glück. Is there a fourth Futamura projection? In *Partial Evaluation and Program Manipulation. Proceedings*, 51–60. ACM Press, 2009.
7. R. Glück. An experiment with the fourth Futamura projection. In A. Pnueli, I. Virbitskaite, A. Voronkov (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 5947, 135–150. Springer-Verlag, 2010.
8. R. Glück, Y. Kawada, T. Hashimoto. Transforming interpreters into inverse interpreters by partial evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings*, 10–19. ACM Press, 2003.
9. R. Glück, V. F. Turchin. Application of metasystem transition to function inversion and transformation. In *Proceedings of the Int. Symposium on Symbolic and Algebraic Computation (ISSAC'90)*, 286–287. ACM Press, 1990.
10. C. K. Gomard, N. D. Jones. Compiler generation by partial evaluation: a case study. *Structured Programming*, 12(3):123–144, 1991.
11. J. Hatcliff. An introduction to online and offline partial evaluation using a simple flowchart language. In J. Hatcliff, T. Æ. Mogensen, P. Thiemann (eds.), *Partial Evaluation. Practice and Theory*, LNCS 1706, 20–82. Springer-Verlag, 1999.
12. N. D. Jones. Automatic program specialization: a re-examination from basic principles. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 225–282. North-Holland, 1988.
13. N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
14. N. D. Jones, P. Sestoft, H. Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In J.-P. Jouannaud (ed.), *Rewriting Techniques and Applications*, LNCS 202, 124–140. Springer-Verlag, 1985.
15. T. Æ. Mogensen. Self-applicable online partial evaluation of the pure lambda calculus. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 39–44. ACM Press, 1995.
16. S. A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 445–463. North-Holland, 1988.
17. S. A. Romanenko. The specializer Unmix, 1990. Program and documentation available from `ftp://ftp.diku.dk/pub/diku/dists/jones-book/Romanenko/`.
18. E. Ruf, D. Weise. On the specialization of online program specializers. *Journal of Functional Programming*, 3(3):251–281, 1993.
19. M. Sperber. Self-applicable online partial evaluation. In O. Danvy, R. Glück, P. Thiemann (eds.), *Partial Evaluation. Proceedings*, LNCS 1110, 465–480. Springer-Verlag, 1996.
20. E. Sumii, N. Kobayashi. Online-and-offline partial evaluation: a mixed approach. In *Partial Evaluation and Semantics-Based Program Manipulation. Proceedings*, 12–21. ACM Press, 2000.

A Appendix: Ackermann Generation Extension

Fig. 12 shows the complete generating extension of the Ackermann program in Fig. 3 produced by self-application of the online partial evaluator `onmix`. Given a value for `m`, the generating extension produces a residual program such as the one shown in Fig. 11. No arity raising was performed, so the input to the generating extension is a list `initial-vs` that contains a single value (`m`). The generating

```
((initial-vs) (0-0)
 ((0-0  (vs    := (initstore '(m) initial-vs))
        (vs    := (procrustes vs '(m)))
        (code  := (make-header '(n) 'ack vs))
        (code  := (call 1-0 vs code))
        (return code))

  (1-0  (if (done 'ack vs code) 2-0 3-0))
  (2-0  (return code))
  (3-0  (code  := (new-block code 'ack vs))
        (if (evalexp '(= m 0) vs) 16-0 17-0))
  (16-0 (return (add-to code (list 'return (reduce '(+ n 1) vs '())))))
  (17-0 (vs1   := (procrustes vs '(m)))
        (vs2   := (procrustes vs '(m)))
        (cond  := (list 'if (reduce '(= n 0) vs '(m)) (cons 'ack0 vs1) (cons 'ack1 vs2)))
        (vs    := vs1)
        (code  := (call 1-1 vs code))
        (vs    := vs2)
        (code  := (call 1-2 vs code))
        (return (add-to code cond)))

  (1-1  (if (done 'ack0 vs code) 2-0 3-1))
  (3-1  (code  := (new-block code 'ack0 vs))
        (vs    := (assign 'n (evalexp '1 vs) vs))
        (vs    := (assign 'm (evalexp '(- m 1) vs) vs))
        (value := (call 50-0 vs))
        (vs    := (assign 'n value vs))
        (return (add-to code (list 'return (reduce 'n vs '(m n))))))

  (1-2  (if (done 'ack1 vs code) 2-0 3-2))
  (3-2  (code  := (new-block code 'ack1 vs))
        (code  := (add-to code (list 'n ':= (reduce '(- n 1) vs '(m)))))
        (vs1   := vs)
        (vs    := (procrustes vs '(m)))
        (code  := (call 1-0 vs code))
        (code  := (add-to code (list 'n ':= (make-call 'ack vs '(n)))))
        (vs    := vs1)
        (vs    := (assign 'm (evalexp '(- m 1) vs) vs))
        (vs1   := vs)
        (vs    := (procrustes vs '(m)))
        (code  := (call 1-0 vs code))
        (code  := (add-to code (list 'n ':= (make-call 'ack vs '(n)))))
        (vs    := vs1)
        (return (add-to code (list 'return (reduce 'n vs '(m))))))

  (50-0 (if (evalexp '(= m 0) vs) 57-0 58-0))
  (57-0 (return (evalexp '(+ n 1) vs)))
  (58-0 (if (evalexp '(= n 0) vs) 57-1 58-1))
  (57-1 (vs    := (assign 'n (evalexp '1 vs) vs))
        (vs    := (assign 'm (evalexp '(- m 1) vs) vs))
        (value := (call 50-0 vs))
        (vs    := (assign 'n value vs))
        (return (evalexp 'n vs)))
  (58-1 (vs    := (assign 'n (evalexp '(- n 1) vs) vs))
        (value := (call 50-0 vs))
        (vs    := (assign 'n value vs))
        (vs    := (assign 'm (evalexp '(- m 1) vs) vs))
        (value := (call 50-0 vs))
        (vs    := (assign 'n value vs))
        (return (evalexp 'n vs))) ))
```

Fig. 12. Generating extension of the Ackermann program (`m` *static*, `n` *dynamic*)

extension inherits several primitive operations from the actual implementation of onmix. The store vs is updated by primitive operation assign and the primitive operation procrustes limits a store vs to the bindings of the variables listed as its second argument. Code is generated and added to the residual program by the primitive operations make-header, make-call, add-to, and new-block. The primitive operations reduce and evalexp are identical to the ones in Sect. 3.1.

The blocks from 1-0 to 17-0, from 1-1 to 3-1, and from 1-2 to 3-2 produce residual versions of the original blocks ack, ack0, and ack1, respectively. They are specialized versions of procedure pepoly (Fig. 10) implemented in Flowchart.

The blocks from 50-0 to 58-1 are a complete implementation of the Ackermann function albeit with interpretive overhead inherited from the self-interpreter evalpp (Fig. 4), which is part of onmix. The entry block 50-0 is called in block 3-1 to compute the value of the Ackermann function. It is instructive to compare this implementation to the original program (Fig. 3). Even though this version is slower than the original program, it allows the generating extension to precompute the Ackermann function when generating a residual program.

Transition compression has duplicated some commands (*e.g.*, the commands of the original block ack2 are the last four commands of the blocks 57-1 and 58-1).

A Graph-Based Definition of Distillation

G.W. Hamilton and G. Mendel-Gleason

School of Computing and Lero@DCU
Dublin City University
Ireland
e-mail: {*hamilton,ggleason*}*@computing.dcu.ie*

Abstract. In this paper, we give a graph-based definition of the distillation transformation algorithm. This definition is made within a similar framework to the positive supercompilation algorithm, thus allowing for a more in-depth comparison of the two algorithms. We find that the main distinguishing characteristic between the two algorithms is that in positive supercompilation, generalization and folding are performed with respect to expressions, while in distillation they are performed with respect to graphs. We also find that while only linear improvements in performance are possible using positive supercompilation, super-linear improvements are possible using distillation. This is because computationally expensive terms can only be extracted from within loops when generalizing graphs rather than expressions.

1 Introduction

Supercompilation is a program transformation technique for functional languages which can be used for program specialization and for the removal of intermediate data structures. Supercompilation was originally devised by Turchin in what was then the USSR in the late 1960s, but did not become widely known to the outside world until a couple of decades later. One reason for this delay was that the work was originally published in Russian in journals which were not accessible to the outside world; it was eventually published in mainstream journals much later [1,2]. Another possible reason why supercompilation did not become more widely known much earlier is that it was originally formulated in the language Refal, which is rather unconventional in its use of a complex pattern matching algorithm. This meant that Refal programs were hard to understand, and describing transformations making use of this complex pattern matching algorithm made the descriptions quite inaccessible. This problem was overcome by the development of *positive supercompilation* [3,4], which is defined over a more familiar functional language. The positive supercompilation algorithm was further extended by the first author to give the *distillation* algorithm [5,6].

In this paper we give a graph-based definition of distillation which we believe gives the algorithm a more solid theoretical foundation. This definition is made within a similar framework to the positive supercompilation algorithm, thus allowing a more detailed comparison between the two algorithms to be made.

There are two reasons why we do this comparison with positive supercompilation rather than any other formulation of supercompilation. Firstly, positive supercompilation is defined on a more familiar functional language similar to that for which distillation is defined, thus facilitating a more direct comparison. Secondly, the original supercompilation algorithm is less clearly defined and has many variants, thus making comparison difficult. We find that the main distinguishing characteristic between the two algorithms is that in positive supercompilation, generalization and folding are performed with respect to expressions, while in distillation, they are performed with respect to graphs. We find that super-linear improvements in performance are possible using distillation, but not using positive supercompilation, because computationally expensive terms can only be extracted from within loops when generalizing graphs rather than expressions.

The remainder of this paper is structured as follows. In Section 2 we define the higher-order functional language on which the described transformations are performed. In Section 3 we define the positive supercompilation algorithm. In Section 4 we define the distillation algorithm by using graphs to determine when generalization and folding should be performed. In Section 5 we show how programs can be extracted from the graphs generated by positive supercompilation and distillation, and Section 6 concludes.

2 Language

In this section, we describe the higher-order functional language which will be used throughout this paper. The syntax of this language is given in Fig. 1.

$prog$	$::= e_0$ **where** $f_1 = e_1 \ldots f_k = e_k$	Program
e	$::= v$	Variable
	$\mid c\ e_1 \ldots e_k$	Constructor
	$\mid f$	Function Call
	$\mid \lambda v.e$	λ-Abstraction
	$\mid e_0\ e_1$	Application
	$\mid$ **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	Case Expression
p	$::= c\ v_1 \ldots v_k$	Pattern

Fig. 1. Language Syntax

Programs in the language consist of an expression to evaluate and a set of function definitions. The intended operational semantics of the language is normal order reduction. It is assumed that erroneous terms such as $(c\ e_1 \ldots e_k)\ e$ and **case** $(\lambda v.e)$ **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$ cannot occur. The variables in the patterns of **case** expressions and the arguments of λ-abstractions are *bound*; all other variables are *free*. We use $fv(e)$ and $bv(e)$ to denote the free and bound variables respectively of expression e. We write $e \equiv e'$ if e and e' differ only in the names of bound variables. We require that each function has exactly one definition and that all variables within a definition are bound. We define a function *unfold* which replaces a function name with its definition.

Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. We allow the usual notation [] for *Nil*, $x : xs$ for $Cons\ x\ xs$ and $[e_1, \ldots, e_k]$ for $Cons\ e_1 \ldots (Cons\ e_k\ Nil)$.

Within the expression **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$, e_0 is called the *selector*, and $e_1 \ldots e_k$ are called the *branches*. The patterns in **case** expressions may not be nested. No variables may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive.

We use the notation $[e'_1/e_1, \ldots, e'_n/e_n]$ to denote a *replacement*, which represents the simultaneous replacement of the expressions $e_1, \ldots, e_n$ by the corresponding expressions $e'_1, \ldots, e'_n$, respectively. We say that a replacement is a *substitution* if all of the expressions $e_1, \ldots, e_n$ are variables and define a predicate *is-sub* to determine whether a given replacement is a substitution. We say that an expression e is an *instance* of another expression e' iff there is a substitution θ s.t. $e \equiv e'\ \theta$.

Example 1. An example program for reversing the list xs is shown in Fig. 2.

```
nrev xs
where
nrev = λxs.case xs of
              []        ⇒ []
            | x' : xs'  ⇒ app (nrev xs') [x']
app  = λxs.λys.case xs of
                 []        ⇒ ys
               | x' : xs'  ⇒ x' : (app xs' ys)
```

Fig. 2. Example Program for List Reversal

3 Positive Supercompilation

In this section, we define the positive supercompilation algorithm; this is largely based on the definition given in [4], but has been adapted to define positive supercompilation within a similar framework to distillation. Within our formulation, positive supercompilation consists of three phases; *driving* (denoted by $\mathcal{D}_\mathcal{S}$), *process graph* construction (denoted by $\mathcal{G}_\mathcal{S}$) and *folding* (denoted by $\mathcal{F}_\mathcal{S}$). The positive supercompilation $\mathcal{S}$ of an expression e is therefore defined as: $\mathcal{S}[\![e]\!] = \mathcal{F}_\mathcal{S}[\![\mathcal{G}_\mathcal{S}[\![\mathcal{D}_\mathcal{S}[\![e]\!]]\!]]\!]$

3.1 Driving

At the heart of the positive supercompilation algorithm are a number of *driving* rules which reduce a term (possibly containing free variables) using normal-order reduction to produce a *process tree*. We define the rules for driving by identifying the next reducible expression (*redex*) within some *context*. An expression which cannot be broken down into a redex and a context is called an *observable*. These are defined as follows.

Definition 1 (Redexes, Contexts and Observables). Redexes, contexts and observables are defined as shown in Fig. 3, where *red* ranges over redexes, *con* ranges over contexts and *obs* ranges over observables (the expression $con\langle e\rangle$ denotes the result of replacing the 'hole' $\langle\rangle$ in *con* by e).

$$
\begin{array}{lll}
red & ::= & f \\
 & | & (\lambda v.e_0)\ e_1 \\
 & | & \mathbf{case}\ (v\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k' \\
 & | & \mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k' \\
 & & \\
con & ::= & \langle\rangle \\
 & | & con\ e \\
 & | & \mathbf{case}\ con\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k \\
 & & \\
obs & ::= & v\ e_1 \ldots e_n \\
 & | & c\ e_1 \ldots e_n \\
 & | & \lambda v.e
\end{array}
$$

Fig. 3. Syntax of Redexes, Contexts and Observables

Lemma 1 (Unique Decomposition Property). For every expression e, either e is an observable or there is a unique context *con* and redex e' s.t. $e = con\langle e'\rangle$. □

Definition 2 (Process Trees). A *process tree* is a directed tree where each node is labelled with an expression, and all edges leaving a node are ordered. One node is chosen as the *root*, which is labelled with the original expression to be transformed. We use the notation $e \rightarrow t_1, \ldots, t_n$ to represent the tree with root labelled e and n children which are the subtrees $t_1, \ldots, t_n$ respectively. Within a process tree t, for any node α, $t(\alpha)$ denotes the label of α, $anc(t, \alpha)$ denotes the set of ancestors of α in t, $t\{\alpha := t'\}$ denotes the tree obtained by replacing the subtree with root α in t by the tree t' and $root(t)$ denotes the label at the root of t.

Definition 3 (Driving). The core set of transformation rules for positive supercompilation are the driving rules shown in Fig. 4, which define the map $\mathcal{D}_\mathcal{S}$

from expressions to process trees. The rules simply perform normal order reduction, with information propagation within **case** expressions giving the assumed outcome of the test. Note that the driving rules are mutually exclusive and exhaustive by the unique decomposition property.

$$\begin{array}{ll}
\mathcal{D}_{\mathcal{S}}[\![v\ e_1 \ldots e_n]\!] & = v\ e_1 \ldots e_n \rightarrow \mathcal{D}_{\mathcal{S}}[\![e_1]\!], \ldots, \mathcal{D}_{\mathcal{S}}[\![e_n]\!] \\
\mathcal{D}_{\mathcal{S}}[\![c\ e_1 \ldots e_n]\!] & = c\ e_1 \ldots e_n \rightarrow \mathcal{D}_{\mathcal{S}}[\![e_1]\!], \ldots, \mathcal{D}_{\mathcal{S}}[\![e_n]\!] \\
\mathcal{D}_{\mathcal{S}}[\![\lambda v.e]\!] & = \lambda v.e \rightarrow \mathcal{D}_{\mathcal{S}}[\![e]\!] \\
\mathcal{D}_{\mathcal{S}}[\![con\langle f\rangle]\!] & = con\langle f\rangle \rightarrow \mathcal{D}_{\mathcal{S}}[\![con\langle unfold\ f\rangle]\!] \\
\mathcal{D}_{\mathcal{S}}[\![con\langle(\lambda v.e_0)\ e_1\rangle]\!] & = con\langle(\lambda v.e_0)\ e_1\rangle \rightarrow \mathcal{D}_{\mathcal{S}}[\![con\langle e_0[e_1/v]\rangle]\!] \\
\multicolumn{2}{l}{\mathcal{D}_{\mathcal{S}}[\![con\langle \mathbf{case}\ (v\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k\rangle]\!]} \\
 & = con\langle \mathbf{case}\ (v\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k\rangle \rightarrow \\
 & \mathcal{D}_{\mathcal{S}}[\![v\ e_1 \ldots e_n]\!], \mathcal{D}_{\mathcal{S}}[\![e'_1[p_1/v\ e_1 \ldots e_n]]\!], \ldots, \mathcal{D}_{\mathcal{S}}[\![e'_k[p_k/v\ e_1 \ldots e_n]]\!] \\
\multicolumn{2}{l}{\mathcal{D}_{\mathcal{S}}[\![con\langle \mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k\rangle]\!]} \\
 & = con\langle \mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k\rangle \rightarrow \\
 & \mathcal{D}_{\mathcal{S}}[\![con\langle e_i[e_1/v_1, \ldots, e_n/v_n]\rangle]\!] \\
 & \text{where } p_i = c\ v_1 \ldots v_n
\end{array}$$

Fig. 4. Driving Rules

As process trees are potentially infinite data structures, they should be lazily evaluated.

Example 2. A portion of the process tree generated from the list reversal program in Fig. 2 is shown in Fig. 5.

3.2 Generalization

In positive supercompilation, generalization is performed when an expression is encountered which is an *embedding* of a previously encountered expression. The form of embedding which we use to inform this process is known as *homeomorphic embedding.* The homeomorphic embedding relation was derived from results by Higman [7] and Kruskal [8] and was defined within term rewriting systems [9] for detecting the possible divergence of the term rewriting process. Variants of this relation have been used to ensure termination within positive supercompilation [10], partial evaluation [11] and partial deduction [12,13]. It can be shown that the homeomorphic embedding relation $\lesssim_e$ is a *well-quasi-order*, which is defined as follows.

Definition 4 (Well-Quasi Order). A well-quasi order on a set S is a reflexive, transitive relation $\leq_S$ such that for any infinite sequence $s_1, s_2, \ldots$ of elements from S there are numbers i, j with $i < j$ and $s_i \leq_S s_j$.

This ensures that in any infinite sequence of expressions $e_0, e_1, \ldots$ there definitely exists some $i < j$ where $e_i \lesssim_e e_j$, so an embedding must eventually be encountered and transformation will not continue indefinitely.

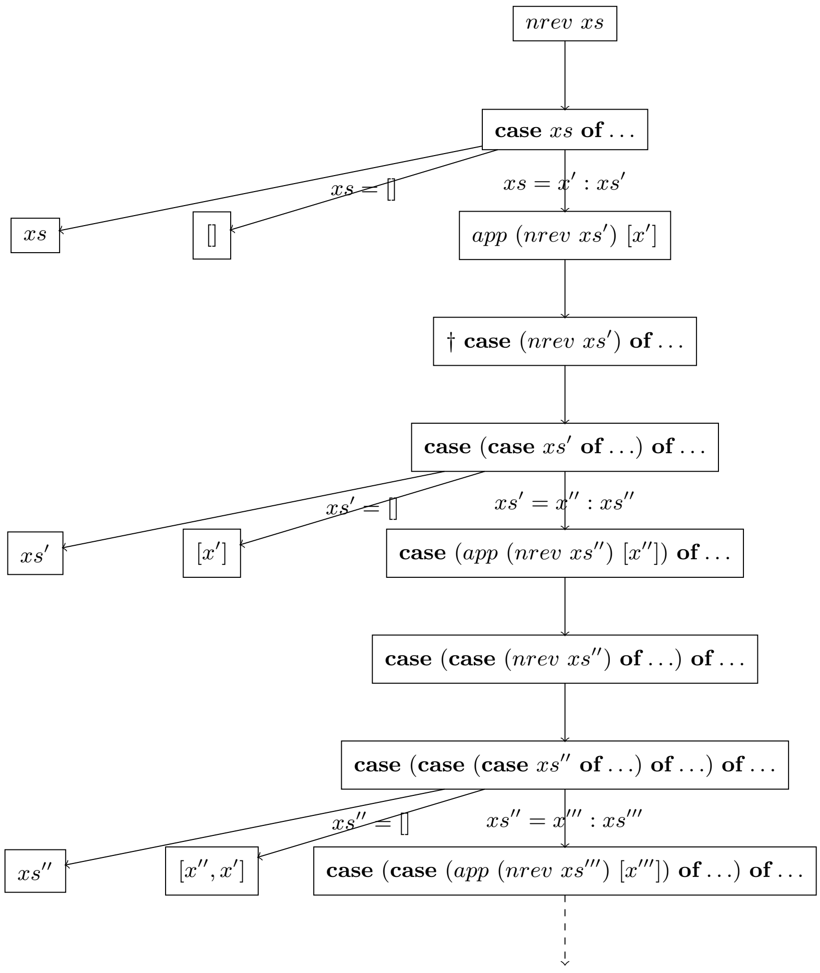

nrev xs
case xs of ...
xs = []
xs = x′ : xs′
xs
[]
app (nrev xs′) [x′]
† case (nrev xs′) of ...
case (case xs′ of ...) of ...
xs′ = []
xs′ = x″ : xs″
xs′
[x′]
case (app (nrev xs″) [x″]) of ...
case (case (nrev xs″) of ...) of ...
case (case (case xs″ of ...) of ...) of ...
xs″ = []
xs″ = x‴ : xs‴
xs″
[x″, x′]
case (case (app (nrev xs‴) [x‴]) of ...) of ...

Fig. 5. Portion of Process Tree Resulting From Driving *nrev xs*

Definition 5 (Homeomorphic Embedding of Expressions). To define the homeomorphic embedding relation on expressions $\lesssim_e$, we firstly define a relation $\trianglelefteq_e$ which requires that all of the free variables within the two expressions match up as follows:

$$\frac{e_1 \lhd_e e_2}{e_1 \trianglelefteq_e e_2} \qquad \frac{e_1 \bowtie_e e_2}{e_1 \trianglelefteq_e e_2} \qquad \frac{e \trianglelefteq_e (e'[v/v'])}{\lambda v.e \bowtie_e \lambda v'.e'}$$

$$\frac{\exists i \in \{1 \ldots n\}.e \trianglelefteq_e e_i}{e \lhd_e \phi(e_1, \ldots, e_n)} \qquad \frac{\forall i \in \{1 \ldots n\}.e_i \trianglelefteq_e e'_i}{\phi(e_1, \ldots, e_n) \bowtie_e \phi(e'_1, \ldots, e'_n)}$$

$$\frac{e_0 \trianglelefteq_e e'_0 \quad \forall i \in \{1 \ldots n\}.\exists \theta_i . p_i \equiv (p'_i\ \theta_i) \wedge e_i \trianglelefteq_e (e'_i\ \theta_i)}{(\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 | \ldots | p_n : e_n) \bowtie_e (\textbf{case}\ e'_0\ \textbf{of}\ p'_1 : e'_1 | \ldots | p'_n : e'_n)}$$

An expression is embedded within another by this relation if either *diving* (denoted by $\lhd_e$) or *coupling* (denoted by $\bowtie_e$) can be performed. Diving occurs when an expression is embedded in a sub-expression of another expression, and coupling occurs when two expressions have the same top-level functor and all the corresponding sub-expressions of the two expressions are embedded. This embedding relation is extended slightly to be able to handle constructs such as λ-abstractions and **case** expressions which may contain bound variables. In these instances, the bound variables within the two expressions must also match up. The homeomorphic embedding relation $\lesssim_e$ can now be defined as follows:

$$e_1 \lesssim_e e_2 \text{ iff } \exists \theta . \textit{is-sub}(\theta) \wedge e_1\ \theta \bowtie_e e_2$$

Thus, within this relation, the two expressions must be coupled, but there is no longer a requirement that all of the free variables within the two expressions match up.

Definition 6 (Generalization of Expressions). The generalization of two expressions e and e' (denoted by $e \sqcap_e e'$) is a triple (e_g, θ, θ') where θ and θ' are substitutions such that $e_g\theta \equiv e$ and $e_g\theta' \equiv e'$, as defined in term algebra [9][1]. This generalization is defined as follows:

$$e \sqcap_e e' = \begin{cases} (\phi(e^g_1, \ldots, e^g_n), \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i), & \text{if } e \lesssim_e e' \\ \quad \text{where } e = \phi(e_1, \ldots, e_n) & \\ \qquad\qquad e' = \phi(e'_1, \ldots, e'_n) & \\ \qquad\qquad (e^g_i, \theta_i, \theta'_i) = e_i \sqcap_e e'_i & \\ (v, [e/v], [e'/v]), & \text{otherwise} \end{cases}$$

Within these rules, if both expressions have the same functor at the outermost level, this is made the outermost functor of the resulting generalized expression, and the corresponding sub-expressions within the functor applications are then generalized. Otherwise, both expressions are replaced by the same variable. The rewrite rule $(e, \theta[e'/v_1, e'/v_2], \theta'[e''/v_1, e''/v_2]) \Rightarrow (e[v_2/v_1], \theta[e'/v_2], \theta[e''/v_2])$ is exhaustively applied to the triple resulting from generalization to minimize the substitutions by identifying common substitutions which were previously given different names.

To represent the result of generalization, we introduce a **let** construct of the form **let** $v_1 = e_1, \ldots, v_n = e_n$ **in** e_0 into our language. This represents the permanent extraction of the expressions $e_1, \ldots, e_n$, which will be transformed separately. The driving rule for this new construct is as follows:

$$\begin{array}{l} \mathcal{D}_\mathcal{S}[\![con\langle \textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_0 \rangle]\!] = \\ \quad con\langle \textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_0 \rangle \rightarrow \mathcal{D}_\mathcal{S}[\![e_1]\!], \ldots, \mathcal{D}_\mathcal{S}[\![e_n]\!], \mathcal{D}_\mathcal{S}[\![con\langle e_0 \rangle]\!] \end{array}$$

[1] Note that, in a higher-order setting, this is no longer a most specific generalization, as the most specific generalization of the terms $f\ (g\ x)$ and $f\ (h\ x)$ would be $(f\ (v\ x), [g/v], [h/v])$, whereas $f\ (g\ x) \sqcap_e f\ (h\ x) = (f\ v, [(g\ x)/v], [(h\ x)/v])$.

We now define an *abstract* operation on expressions which extracts the sub-terms resulting from generalization using **let** expressions.

Definition 7 (Abstract Operation).

$abstract_e(e, e') = \textbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \textbf{in}\ e_g$
where $e \sqcap_e e' = (e_g, [e_1/v_1, \ldots, e_n/v_n], \theta)$

3.3 Process Graph Construction

In our formulation of positive supercompilation, the potentially infinite process tree produced by driving is converted into a finite *process graph.*

Definition 8 (Process Graph). A process graph is a process tree which may in addition contain *replacement nodes.* A replacement node has the form $e \overset{\theta}{\dashrightarrow} \alpha$ where α is an ancestor node in the tree and θ is a replacement s.t. $t(\alpha)\ \theta \equiv e$.

Definition 9 (Process Graph Substitution). Substitution in a process graph is performed by applying the substitution pointwise to all the node labels within it as follows.

$$(e \rightarrow t_1, \ldots, t_n)\ \theta = e\ \theta \rightarrow t_1\ \theta, \ldots, t_n\ \theta$$

Definition 10 (Process Graph Equivalence). Two process graphs are equivalent if the following relation is satisfied.

$$con\langle e\rangle \rightarrow t_1, \ldots, t_n \equiv con'\langle e'\rangle \rightarrow t'_1, \ldots, t'_n, \text{ iff } e \lesssim_e e' \wedge \forall i \in \{1 \ldots n\}.t_i \equiv t'_i$$

$$e \overset{\theta}{\dashrightarrow} t \equiv e' \overset{\theta'}{\dashrightarrow} t', \text{ iff } t \equiv t'$$

Within this relation, there is therefore a requirement that the redexes within corresponding nodes are coupled.

Definition 11 (Process Graph Construction in Positive Supercompilation). The rules for the construction of a process graph from a process tree in positive supercompilation t are as follows.

$$\mathcal{G_S}[\![\beta = con\langle f\rangle \rightarrow t']\!] = \begin{cases} con\langle f\rangle \overset{[e'_i/e_i]}{\dashrightarrow} \alpha, & \text{if } \exists \alpha \in anc(t,\beta).t(\alpha) \lesssim_e t(\beta) \\ con\langle f\rangle \rightarrow \mathcal{G_S}[\![t']\!], & \text{otherwise} \end{cases}$$
$$\text{where}$$
$$t(\alpha) \sqcap_e t(\beta) = (e_g, [e_i/v_i], [e'_i/v_i])$$

$$\mathcal{G_S}[\![e \rightarrow t_1, \ldots, t_n]\!] = e \rightarrow \mathcal{G_S}[\![t_1]\!], \ldots, \mathcal{G_S}[\![t_n]\!]$$

A process graph is considered to be *folded* when all of the replacements within it are substitutions. This folding is performed as follows.

Definition 12 (Folding in Positive Supercompilation). The rules for folding a process graph t using positive supercompilation are as follows.

$$\mathcal{F}_{\mathcal{S}}[\![e \overset{\theta}{\dashrightarrow} \alpha]\!] \quad = \begin{cases} e \overset{\theta}{\dashrightarrow} \alpha, & \text{if } \textit{is-sub}(\theta) \\ t\{\alpha := \mathcal{S}[\![abstract_e(t(\alpha), e)]\!]\}, & \text{otherwise} \end{cases}$$

$$\mathcal{F}_{\mathcal{S}}[\![e \rightarrow t_1, \ldots, t_n]\!] = e \rightarrow \mathcal{F}_{\mathcal{S}}[\![t_1]\!], \ldots, \mathcal{F}_{\mathcal{S}}[\![t_n]\!]$$

Example 3. The process graph constructed from the process tree in Fig. 5 is shown in Fig. 6 where the replacement θ is equal to $[app\ (nrev\ xs'')\ [x'']/nrev\ xs']$.

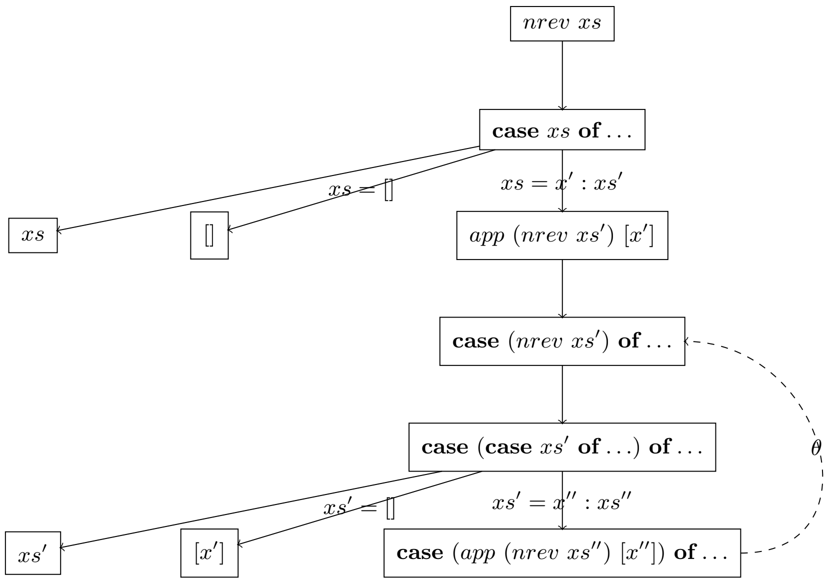


Fig. 6. Process Graph Constructed for *nrev xs*

The folded process graph constructed from the process graph in Fig. 6 is shown in Fig. 7.

4 Distillation

In this section, we define the distillation algorithm within a similar framework to that used to define positive supercompilation in the previous section. Distillation consists of two phases; driving (the same as for positive supercompilation) and folding (denoted by $\mathcal{F}_{\mathcal{D}}$). The distillation $\mathcal{D}$ of an expression e is therefore defined as: $\mathcal{D}[\![e]\!] = \mathcal{F}_{\mathcal{D}}[\![\mathcal{D}_{\mathcal{S}}[\![e]\!]]\!]$. Folding in distillation is performed with respect to process graphs. We therefore define what it means for one process graph to be an instance or a homeomorphic embedding of another.

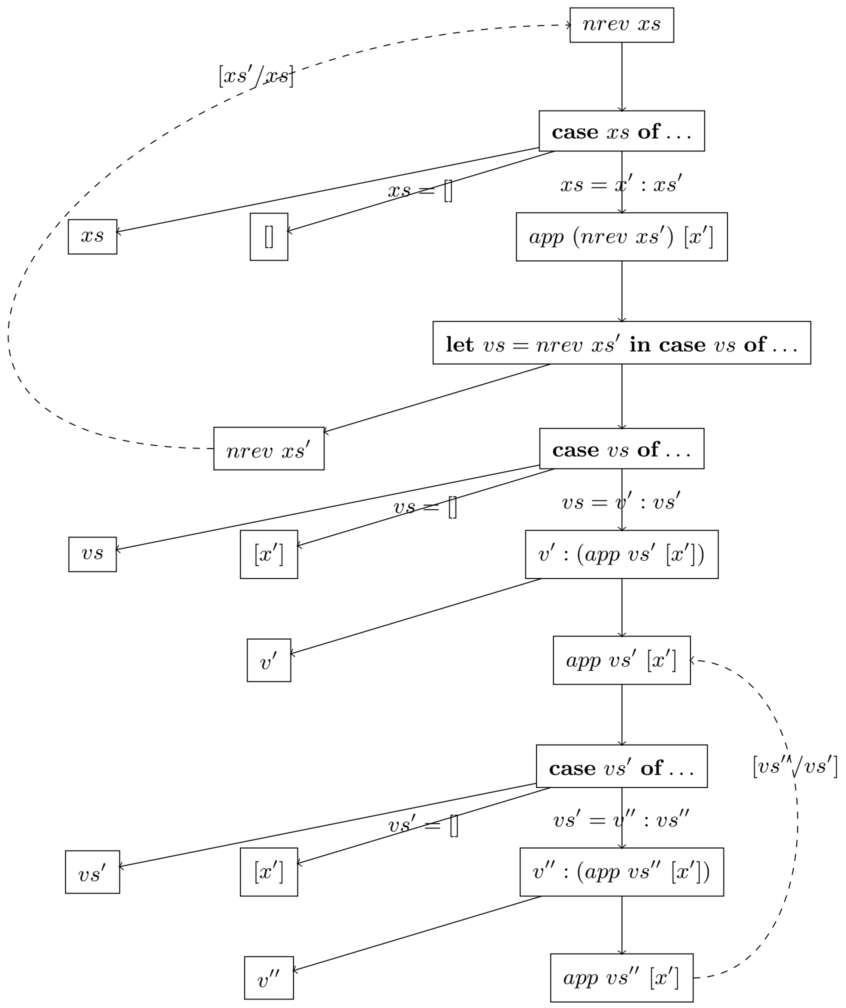


Fig. 7. Folded Process Graph for *nrev xs*

Definition 13 (Process Graph Instance). A process graph t' is an instance of another process graph t (denoted by $t \lessdot_\theta t'$) iff there is a substitution θ s.t. $t \equiv t'\ \theta$.

Definition 14 (Homeomorphic Embedding of Process Graphs). To define the homeomorphic embedding relation on process graphs $\lesssim_t$, we firstly define a relation $\trianglelefteq_t$ which requires that all the free variables in the two process graphs match up as follows:

$$\frac{t_1 \lhd_t t_2}{t_1 \trianglelefteq_t t_2} \qquad \frac{t_1 \bowtie_t t_2}{t_1 \trianglelefteq_t t_2} \qquad \frac{t \trianglelefteq_t (t'[v/v'])}{\lambda v.e \rightarrow t \bowtie_t \lambda v'.e' \rightarrow t'}$$

$$\frac{e \bowtie_e e' \quad \forall i \in \{1 \ldots n\}.t_i \trianglelefteq_t t'_i}{con\langle e\rangle\!\rightarrow t_1, \ldots, t_n \bowtie_t con'\langle e'\rangle\!\rightarrow t'_1, \ldots, t'_n} \qquad \frac{\exists i \in \{1 \ldots n\}.t \trianglelefteq_t t_i}{t \lhd_t e \rightarrow t_1, \ldots, t_n}$$

$$\frac{t \bowtie_t t'}{e \overset{\theta}{\dashrightarrow} t \bowtie_t e' \overset{\theta'}{\dashrightarrow} t'}$$

$$\frac{t_0 \trianglelefteq_t t'_0 \quad \forall i \in \{1...n\}.\exists \theta_i.p_i \equiv (p'_i\ \theta_i) \wedge t_i \trianglelefteq_t (t'_i\ \theta_i)}{(\mathbf{case}\ e_0\ \mathbf{of}\ p_1 : e_1|...|p_n : e_n) \rightarrow t_0, ..., t_n \bowtie_t (\mathbf{case}\ e'_0\ \mathbf{of}\ p'_1 : e'_1|...|p'_n : e'_n) \rightarrow t'_0, ..., t'_n}$$

A tree is embedded within another by this relation if either *diving* (denoted by $\lhd_t$) or *coupling* (denoted by $\bowtie_t$) can be performed. Diving occurs when a tree is embedded in a sub-tree of another tree, and coupling occurs when the redexes of the root expressions of two trees are coupled. As for the corresponding embedding relation on expressions, this embedding relation is extended slightly to be able to handle constructs such as λ-abstractions and **case** expressions which may contain bound variables. In these instances, the bound variables within the two process graphs must also match up. The homeomorphic embedding relation on process graphs $\lesssim_t$ can now be defined as follows:

$$t_1 \lesssim_t t_2 \text{ iff } \exists \theta.\textit{is-sub}(\theta) \wedge t_1\ \theta \bowtie_t t_2$$

Within this relation, there is no longer a requirement that all of the free variables within the two process graphs match up.

4.1 Generalization

Generalization is performed on two process trees if their corresponding process graphs are homeomorphically embedded as follows.

Definition 15 (Generalization of Process Trees). Generalization is performed on process trees using the $\sqcap_t$ operator which is defined as follows:

$$t \sqcap_t t' = \begin{cases} (e \rightarrow t^g_1, \ldots, t^g_n, \bigcup_{i=1}^n \theta_i, \bigcup_{i=1}^n \theta'_i), & \text{if } t \lesssim_t t' \\ \quad \text{where } t = e \rightarrow t_1, \ldots, t_n & \\ \qquad\qquad t' = e' \rightarrow t'_1, \ldots, t'_n & \\ \qquad\qquad (t^g_i, \theta_i, \theta'_i) = t_i \sqcap_t t'_i & \\ (\mathcal{D_S}[\![e^g]\!], \theta, \theta'), & \text{otherwise} \\ \quad \text{where } (e^g, \theta, \theta') = root(t) \sqcap_e root(t') & \end{cases}$$

Within these rules, if two trees are coupled then their corresponding sub-trees are generalized. Otherwise, the expressions in the corresponding root nodes are generalized. As the process trees being generalized are potentially infinite, this generalization should also be performed lazily. As is done for the generalization of expressions, the rewrite rule $(e, \theta[e'/v_1, e'/v_2], \theta'[e''/v_1, e''/v_2]) \Rightarrow (e[v_2/v_1], \theta[e'/v_2], \theta[e''/v_2])$ is also exhaustively applied to the triple resulting from generalization to minimize the substitutions by identifying common substitutions which were previously given different names. Note that the use of this rewrite rule is essential for the correctness of the distillation algorithm. We now define an *abstract* operation on process trees which extracts the sub-terms resulting from generalization using **let** expressions.

Definition 16 (Abstract Operation on Process Trees).
$abstract_t(t, t') = (\mathbf{let}\ v_1 = e_1, \ldots, v_n = e_n\ \mathbf{in}\ root(t)) \rightarrow \mathcal{D_S}[\![e_1]\!], ..., \mathcal{D_S}[\![e_n]\!], t_g$
where $t \sqcap_t t' = (t_g, [e_1/v_1, \ldots, e_n/v_n], \theta)$

4.2 Folding

In distillation, process graphs are used to determine when to perform folding and generalization. These process graphs are constructed slightly differently than those in positive supercompilation, with replacement nodes being added when an expression is encountered which is an embedding (rather than a coupling) of an ancestor expression. To facilitate this, a new relation $\lesssim'_e$ is defined as follows:

$$e_1 \lesssim'_e e_2 \text{ iff } \exists \theta . \textit{is-sub}(\theta) \wedge e_1\ \theta \trianglelefteq_e e_2$$

Definition 17 (Process Graph Construction in Distillation). The rules for the construction of a process graph from a process tree in distillation t are as follows.

$$\mathcal{G_D}[\![\beta = con\langle f\rangle \rightarrow t']\!] = \begin{cases} con\langle f\rangle \overset{[e'_i/e_i]}{\dashrightarrow} \alpha, & \text{if } \exists \alpha \in anc(t,\beta).t(\alpha) \lesssim'_e t(\beta) \\ con\langle f\rangle \rightarrow \mathcal{G_D}[\![t']\!], & \text{otherwise} \end{cases}$$
$$\text{where } t(\alpha) \sqcap_e t(\beta) = (e_g, [e_i/v_i], [e'_i/v_i])$$

$$\mathcal{G_D}[\![e \rightarrow t_1, \ldots, t_n]\!] \quad = e \rightarrow \mathcal{G_D}[\![t_1]\!], \ldots, \mathcal{G_D}[\![t_n]\!]$$

Definition 18 (Folding in Distillation). The rules for folding a process tree t using distillation are as follows.

$$\mathcal{F_D}[\![\beta = con\langle f\rangle \rightarrow t']\!] = \begin{cases} con\langle f\rangle \overset{\theta}{\dashrightarrow} \alpha, & \text{if } \exists \alpha \in anc(t,\beta).\mathcal{G_D}[\![\alpha]\!] \lessdot_\theta \mathcal{G_D}[\![\beta]\!] \\ t\{\alpha := \mathcal{F_D}[\![abstract_t(\alpha,\beta)]\!]\}, & \\ & \text{if } \exists \alpha \in anc(t,\beta).\mathcal{G_D}[\![\alpha]\!] \lesssim_t \mathcal{G_D}[\![\beta]\!] \\ con\langle f\rangle \rightarrow \mathcal{F_D}[\![t']\!], & \text{otherwise} \end{cases}$$

$$\mathcal{F_D}[\![e \rightarrow t_1, \ldots, t_n]\!] \quad = e \rightarrow \mathcal{F_D}[\![t_1]\!], \ldots, \mathcal{F_D}[\![t_n]\!]$$

Example 4. The process graph constructed from the root node of the process tree in Fig. 5 is shown in Fig. 8, where the replacement θ is $[app\ (nrev\ xs')\ [x']/nrev\ xs]$.

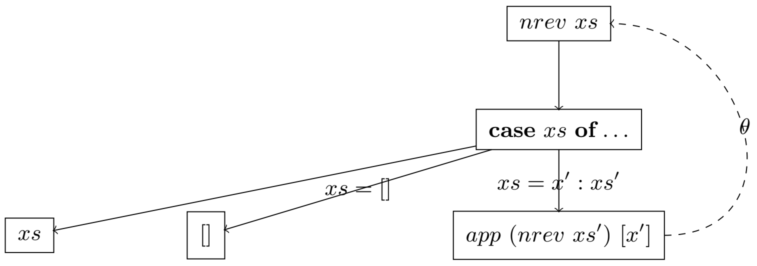


Fig. 8. Process Graph

Similarly, the process graph constructed from the node labelled † in the process tree in Fig. 5 is shown in Fig. 9, where the replacement θ' is equal to $[app\ (nrev\ xs'')\ [x'']/nrev\ xs']$.

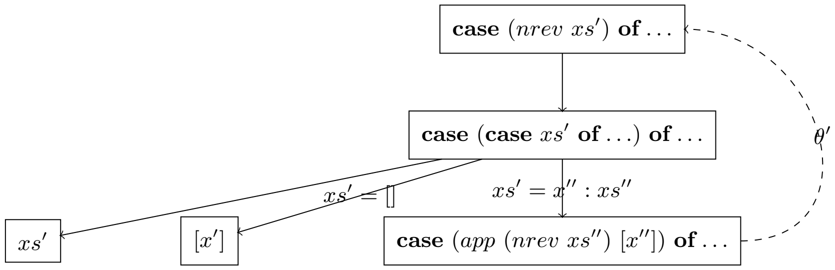

case (nrev xs′) of ...
case (case xs′ of ...) of ...
θ′
xs′ = []
xs′ = x″ : xs″
xs′
[x′]
case (app (nrev xs″) [x″]) of ...

Fig. 9. Process Graph

The process graph in Fig. 8 is embedded in the process graph in Fig. 9, so the corresponding process trees are generalized to produce the process tree shown in Fig. 10. The process graph constructed for the node labelled † is now an instance of the process graph constructed for the root node of this process tree, so folding is performed to produce the folded process graph shown in Fig. 11

5 Program Residualization

A residual program can be constructed from a folded process graph using the rules $\mathcal{C}$ as shown in Fig. 12.

Example 5. The program constructed from the folded process graph resulting from the positive supercompilation of *nrev xs* shown in Fig. 7 is as shown in Fig. 13. The program constructed from the folded process graph resulting from the distillation of *nrev xs* shown in Fig. 11 is as shown in Fig. 14. We can see that the distilled program is a super-linear improvement over the original, while the supercompiled program has produced no improvement.

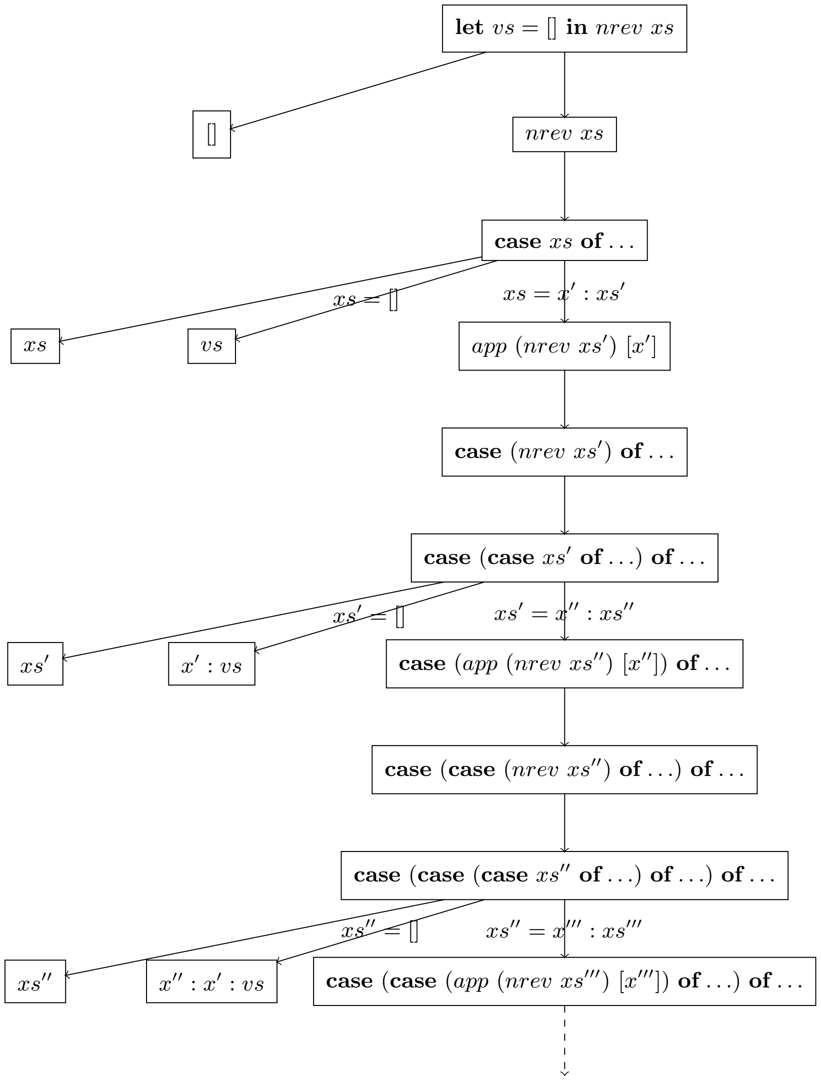

let $vs = []$ in $nrev\ xs$
$[]$
$nrev\ xs$
case xs of ...
$xs = []$
$xs = x' : xs'$
xs
vs
$app\ (nrev\ xs')\ [x']$
case $(nrev\ xs')$ of ...
case (case xs' of ...) of ...
$xs' = []$
$xs' = x'' : xs''$
xs'
$x' : vs$
case $(app\ (nrev\ xs'')\ [x''])$ of ...
case (case $(nrev\ xs'')$ of ...) of ...
case (case (case xs'' of ...) of ...) of ...
$xs'' = []$
$xs'' = x''' : xs'''$
xs''
$x'' : x' : vs$
case (case $(app\ (nrev\ xs''')\ [x'''])$ of ...) of ...

Fig. 10. Result of Generalizing *nrev xs*

6 Conclusion

We have presented a graph-based definition of the distillation transformation algorithm for higher-order functional languages. The definition is made within a similar framework to the positive supercompilation transformation algorithm, thus allowing for a more detailed comparison of the two algorithms. We have

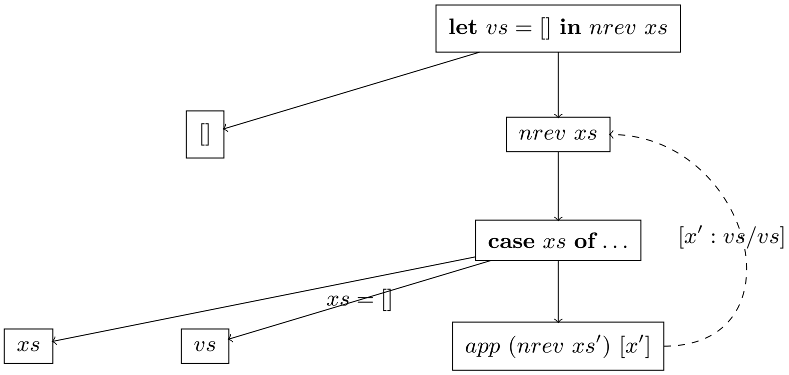


Fig. 11. Result of Folding *nrev xs*

$$
\begin{array}{ll}
\mathcal{C}[\![(v\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!]\ \phi & = v\ (\mathcal{C}[\![t_1]\!]\ \phi) \ldots (\mathcal{C}[\![t_n]\!]\ \phi) \\
\mathcal{C}[\![(c\ e_1 \ldots e_n) \rightarrow t_1, \ldots, t_n]\!]\ \phi & = c\ (\mathcal{C}[\![t_1]\!]\ \phi) \ldots (\mathcal{C}[\![t_n]\!]\ \phi) \\
\mathcal{C}[\![(\lambda v.e) \rightarrow t]\!]\ \phi & = \lambda v.(\mathcal{C}[\![t]\!]\ \phi) \\
\mathcal{C}[\![(con\langle f\rangle) \rightarrow t]\!]\ \phi & = f'\ v_1 \ldots v_n \\
& \quad \text{where} \\
& \quad f' = \lambda v_1 \ldots v_n.\mathcal{C}[\![t]\!]\ (\phi \cup \{f'\ v_1 \ldots v_n = con\langle f\rangle \rightarrow t\}) \\
& \quad \{v_1 \ldots v_n\} = fv(t) \\
\mathcal{C}[\![(con\langle f\rangle) \overset{\theta}{\dashrightarrow} t]\!]\ \phi & = (f\ v_1 \ldots v_n)\ \theta \\
& \quad \text{where} \\
& \quad (f'\ v_1 \ldots v_n = t) \in \phi \\
\multicolumn{2}{l}{\mathcal{C}[\![(con\langle \textbf{case}\ (v\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n\rangle) \rightarrow t_0, \ldots, t_n]\!]\ \phi} \\
& = \textbf{case}\ (\mathcal{C}[\![t_0]\!]\ \phi)\ \textbf{of}\ p_1 \Rightarrow \mathcal{C}[\![t_1]\!]\ \phi \mid \cdots \mid p_n \Rightarrow \mathcal{C}[\![t_n]\!]\ \phi \\
\multicolumn{2}{l}{\mathcal{C}[\![\textbf{let}\ v_1\ =\ t_1, \ldots, v_n\ =\ t_n\ \textbf{in}\ t]\!]\ \phi} \\
& = (\mathcal{C}[\![t]\!]\ \phi)[(\mathcal{C}[\![t_1]\!]\ \phi)/v_1, \ldots, (\mathcal{C}[\![t_n]\!]\ \phi)/v_n]
\end{array}
$$

Fig. 12. Rules For Constructing Residual Programs

found that the main distinguishing characteristic between the two algorithms is that in positive supercompilation, generalization and folding are performed with respect to expressions, while in distillation they are performed with respect to graphs. We have also found that while only linear improvements in performance are possible using positive supercompilation, super-linear improvements are possible using distillation. This is because computationally expensive terms can only be extracted from within loops when generalizing graphs rather than expressions. Of course, this extra power comes at a price. As generalization and folding are now performed on graphs rather than flat terms, there may be an exponential increase in the number of steps required to perform these operations in the worst case.

There are a number of possible directions for further work. It has already been shown how distillation can be used to verify safety properties of programs

```
$f\ xs$
**where**
$f\ = \lambda xs.$**case** $xs$ **of**
              $[]$          $\Rightarrow []$
            $\mid x' : xs' \Rightarrow$ **case** $(f\ xs')$ **of**
                                  $[]$            $\Rightarrow [x']$
                                $\mid x'' : xs'' \Rightarrow x'' : (f'\ xs''\ x')$
$f' = \lambda xs.\lambda y.$**case** $xs$ **of**
                  $[]$          $\Rightarrow [y]$
                $\mid x' : xs' \Rightarrow x' : (f'\ xs'\ y)$
```

Fig. 13. Result of Applying Positive Supercompilation to *nrev xs*

```
$f\ xs\ []$
**where**
$f = \lambda xs.\lambda vs.$**case** $xs$ **of**
                    $[]$          $\Rightarrow vs$
                  $\mid x' : xs' \Rightarrow f\ xs'\ (x' : vs)$
```

Fig. 14. Result of Applying Distilling to *nrev xs*

[14]; work is now in progress by the second author to show how it can also be used to verify liveness properties. Work is also in progress in incorporating the distillation algorithm into the Haskell programming language, so this will allow a more detailed evaluation of the utility of the distillation algorithm to be made. Distillation is being added to the York Haskell Compiler [15] in a manner similar to the addition of positive supercompilation to the same compiler in Supero [16]. Further work is also required in proving the termination and correctness of the distillation algorithm. Finally, it has been found that the output produced by the distillation algorithm is in a form which is very amenable to automatic parallelization. Work is also in progress to incorporate this automatic parallelization into the York Haskell Compiler.

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie), and by the School of Computing, Dublin City University.

References

1. Turchin, V.: Program Transformation by Supercompilation. Lecture Notes in Computer Science **217** (1985) 257–281
2. Turchin, V.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems **8**(3) (July 1986) 90–121

3. Sørensen, M.: Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen (1994) DIKU-rapport 94/17.
4. Sørensen, M., Glück, R., Jones, N.: A Positive Supercompiler. Journal of Functional Programming **6**(6) (1996) 811–838
5. Hamilton, G.W.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. (2007) 61–70
6. Hamilton, G.W.: Extracting the Essence of Distillation. In: Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics (PSI '09). Volume 5947 of Lecture Notes in Computer Science. (2009) 151–164
7. Higman, G.: Ordering by Divisibility in Abstract Algebras. Proceedings of the London Mathemtical Society **2** (1952) 326–336
8. Kruskal, J.: Well-Quasi Ordering, the Tree Theorem, and Vazsonyi's Conjecture. Transactions of the American Mathematical Society **95** (1960) 210–225
9. Dershowitz, N., Jouannaud, J.P.: Rewrite Systems. In van Leeuwen, J., ed.: Handbook of Theoretical Computer Science. Elsevier, MIT Press (1990) 243–320
10. Sørensen, M., Glück, R.: An Algorithm of Generalization in Positive Supercompilation. Lecture Notes in Computer Science **787** (1994) 335–351
11. Marlet, R.: Vers une Formalisation de l'Évaluation Partielle. PhD thesis, Université de Nice - Sophia Antipolis (1994)
12. Bol, R.: Loop Checking in Partial Deduction. Journal of Logic Programming **16**(1–2) (1993) 25–46
13. Leuschel, M.: On the Power of Homeomorphic Embedding for Online Termination. In: Proceedings of the International Static Analysis Symposium. (1998) 230–245
14. Hamilton, G.W.: Distilling Programs for Verification. Electronic Notes in Theoretical Computer Science **190**(4) (2007) 17–32
15. Mitchell, N.: Yhc Manual (wiki)
16. Mitchell, N., Runciman, C.: A Supercompiler for Core Haskell. Lecture Notes in Computer Science **5083** (2008) 147–164

Strengthening Supercompilation For Call-By-Value Languages

Peter A. Jonsson and Johan Nordlander

{pj, nordland}@csee.ltu.se
Luleå University of Technology

Abstract. A termination preserving supercompiler for a call-by-value language sometimes fails to remove intermediate structures that a supercompiler for a call-by-name language would remove. This discrepancy in power stems from the fact that many function bodies are either non-linear in use of an important variable or often start with a pattern match on their first argument and are therefore not strict in all their arguments. As a consequence, intermediate structures are left in the output program, making it slower. We present a revised supercompilation algorithm for a call-by-value language that propagates let-bindings into case-branches and uses termination analysis to remove dead code. This allows the algorithm to remove all intermediate structures for common examples where previous algorithms for call-by-value languages had to leave the intermediate structures in place.

1 Introduction

Intermediate lists in functional programs allows the programmer to write clear and concise programs, but carry a run time cost since list cells need to be both allocated and garbage collected. Much research has been conducted on automatic program transformations that remove these intermediate structures, both for lazy and strict languages [1,2,3,4].

A common pattern that appears both in input programs and during supercompilation is a let-expression where the body is a case-expression: $\textbf{let}\, x = e \,\textbf{in}\, \textbf{case}\, e' \,\textbf{of}\, \{p_i \rightarrow e_i\}$. A supercompiler for a strict language is only allowed to substitute e for x if we know that x is strict in the case-expression, and for pragmatic and proof technical reasons x must also be linear in the case-expression. As expected, it is quite easy to define functions that do not fulfill both of these requirements, or functions that are complex enough to fool the analyses used by the supercompiler. A standard example of such a function is *zip*.

If the supercompiler instead propagates let-expressions into the branches of case-expressions it simplifies the job for the analyses since they no longer need to account for different behaviours in different branches. Not only does this modification increase the precision of the analyses, but it also allows our improved supercompiler to remove more constructions that cause memory allocations. The propagation of let-expressions is orthogonal to the amount of information propagated, so it works for both positive [2] and perfect supercompilation [5]. We illustrate the increased strength through the following example:

$zip\ (map\ f_1\ xs)\ (map\ f_2\ ys)$

Its generalization to tree-like structures is also of interest:

$zipT\ (mapT\ f_3\ t_1)\ (mapT\ f_4\ t_2)$

These examples allow us to position supercompilation for a strict language relative to other well-known program transformations that perform program specialization and remove intermediate structures:

Shortcut deforestation [6] removes one of the intermediate lists in the first example, but does not remove arbitrary algebraic data types.
Stream fusion [7] removes both the intermediate lists in the first example, but does not remove arbitrary algebraic data types without manual extensions.
Positive supercompilation [2] for a strict language removes the first intermediate structure in both examples, and for a lazy language it removes both lists and both trees.

This paper presents one more step towards allowing the programmer to write clear and concise code in strict languages while getting good performance. The contributions of our work are:

- We provide a stronger algorithm for positive supercompilation in a strict and pure functional language (Section 4).
- We extend the supercompiler with a termination test that enables some unused let-bindings to be removed even though they are not fully evaluated. This feature is particularly beneficial in conjunction with the first contribution, since pushing bindings into case branches tend to result in many seemingly redundant let-expressions (Section 5).
- We prove the soundness of the algorithm in Section 6.

We start out with a step by step example where our improved supercompiler removes both intermediate lists for *zip* in Section 2 to give the reader an intuitive feel for how the algorithm behaves. Our language of study is defined in Section 3 followed by the technical contributions. We end with a discussion of the performance of the algorithm in Section 7.

2 Examples

This section gives a walk-through of the transformation of *zip* for readers who are already familiar with positive supercompilation for call-by-value [4]. For readers who are not at all familiar with these techniques there are more examples of step by step transformations in the work of Wadler [1] and Sørensen, Glück and Jones [2].

Our first example is transformation of the standard function *zip*, which takes two lists as parameters: $zip\ (map\ f\ xs')\ (map\ g\ ys')$. The standard definitions of *map* and *zip* are:

$$
\begin{array}{ll}
map = \lambda f\ xs.\ \textbf{case}\ xs\ \textbf{of} & \\
\qquad [] \rightarrow ys & \\
\qquad (x : xs) \rightarrow f\ x : map\ f\ xs & \\
zip = \lambda xs\ ys.\ \textbf{case}\ xs\ \textbf{of} & \\
\qquad [] \rightarrow [] & \\
\qquad (x' : xs') \rightarrow \textbf{case}\ ys\ \textbf{of} & \\
\qquad\qquad [] \rightarrow [] & \\
\qquad\qquad (y' : ys') \rightarrow (x', y') : zip\ xs'\ ys' &
\end{array}
$$

We start our transformation by allocating a new fresh function name (h_0) to this expression, inlining the body of *zip*, substituting $map\ f\ xs'$ into the body of *zip*, and putting $map\ g\ ys'$ into a let-expression to preserve termination properties of the program:

$$
\begin{array}{l}
\textbf{let}\ ys = map\ g\ ys' \\
\textbf{in case}\ map\ f\ xs'\ \textbf{of} \\
\quad [] \rightarrow [] \\
\quad (x' : xs') \rightarrow \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad [] \rightarrow [] \\
\qquad\qquad (y' : ys') \rightarrow (x', y') : zip\ xs'\ ys'
\end{array}
$$

The key difference between this algorithm and our previous work is that it transforms the case expression without touching the let-expression. After inlining the body of *map* in the head of the case-expression and substituting the arguments into the body the result becomes:

$$
\begin{array}{l}
\textbf{let}\ ys = map\ g\ ys' \\
\textbf{in case}\ (\ \textbf{case}\ xs'\ \textbf{of} \\
\qquad\qquad [] \rightarrow [] \\
\qquad\qquad (z : zs) \rightarrow f\ z : map\ f\ zs)\ \textbf{of} \\
\quad [] \rightarrow [] \\
\quad (x' : xs') \rightarrow \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad [] \rightarrow [] \\
\qquad\qquad (y' : ys') \rightarrow (x', y') : zip\ xs'\ ys'
\end{array}
$$

Notice how the let-expression is still untouched by the transformation – this is essential for the power of the transformation. We duplicate the let-expression and the outer case in each of the inner case's branches, using the expression in the branches as the head of the outer case-expression:

```
case xs' of
 [] → let ys = map g ys'
      in case [] of
           [] → []
           (x' : xs') → case ys of
                          [] → []
                          (y' : ys') → (x', y') : zip xs' ys'
 (z : zs) → let ys = map g ys'
            in case f z : map f zs of of
                 [] → []
                 (x' : xs') → case ys of
                                [] → []
                                (y' : ys') → (x', y') : zip xs' ys'
```

The case-expression in the first branch of the outermost case reduces to the empty list, but the let-expression must remain or we might introduce accidental termination in the program. The second branch is more interesting: we have a known constructor in the head of the case-expression so we can perform the reduction:

```
case xs' of
 [] → let ys = map g ys' in []
 (z : zs) → let ys = map g ys'
            in let x' = f z, xs' = map f zs
            in case ys of
                 [] → []
                 (y' : ys') → (x', y') : zip xs' ys'
```

The first branch can either be left as is, or one can transform the let-expression to get a new function that is isomorphic to *map* and a call to it. This is an orthogonal problem to removing multiple intermediate structures however, and we will not treat it further in this example. In Section 5 we show how to automatically remove superfluous let-expressions such as this through termination analysis. The reduction of the case-expression in the second branch reveals that the second branch is strict in *ys*, so *ys* will be evaluated, and the termination behavior will be the same even after performing the substitution. After performing the substitution we have:

```
case xs' of
 [] → let ys = map g ys' in []
 (z : zs) → let x' = f z, xs' = map f zs
            in case map g ys' of
                 [] → []
                 (y' : ys') → (x', y') : zip xs' ys'
```

We repeat inlining the body of *map* in the head of the inner case-expression and substituting the arguments into the body which gives:

```
case $xs'$ of
 $[] \to$ let $ys = map\ g\ ys'$ in $[]$
 $(z : zs) \to$ let $x' = f\ z,\ xs' = map\ f\ zs$
             in case ( case $ys'$ of
                          $[] \to []$
                          $(z' : zs') \to g\ z' : map\ g\ zs')$ of
                 $[] \to []$
                 $(y' : ys') \to (x', y') : zip\ xs'\ ys'$
```

Once again we move the let-expression and the middle case into the branches of the innermost case:

```
case $xs'$ of
 $[] \to$ let $ys = map\ g\ ys'$ in $[]$
 $(z : zs) \to$ case $ys'$ of
             $[] \to$ let $x' = f\ z,\ xs' = map\ f\ zs$
                  in case $[]$ of
                      $[] \to []$
                      $(y' : ys') \to (x', y') : zip\ xs'\ ys'$
             $(z' : zs') \to$ let $x' = f\ z,\ xs' = map\ f\ zs$
                        in case $g\ z' : map\ g\ zs'$ of
                            $[] \to []$
                            $(y' : ys') \to (x', y') : zip\ xs'\ ys'$
```

The first branch reduces to the empty list but we have to preserve the let-expression for termination purposes. Transforming the first branch is not going to reveal anything interesting, so we leave that branch as is, but of course the algorithm transforms that branch as well. The second branch is more interesting since it has a known constructor in the head of a case-expression, so we perform the reduction:

```
case $xs'$ of
 $[] \to$ let $ys = map\ g\ ys'$ in $[]$
 $(z : zs) \to$ case $ys'$ of
             $[] \to$ let $x' = f\ z,\ xs' = map\ f\ zs$ in $[]$
             $(z' : zs') \to$ let $x' = f\ z,\ xs' = map\ f\ zs$
                         in $(x', g\ z') : zip\ xs'\ (map\ g\ zs')$
```

After the reduction it is clear that both *x'* and *xs'* are really strict, so it is safe to substitute them:

```
case $xs'$ of
 $[] \to$ let $ys = map\ g\ ys'$ in $[]$
 $(z : zs) \to$ case $ys'$ of
             $[] \to$ let $x' = f\ z,\ xs' = map\ f\ zs$ in $[]$
             $(z' : zs') \to (f\ z, g\ z') : zip\ (map\ f\ zs)\ (map\ g\ zs')$
```

We notice a familiar expression in $zip\ (map\ f\ zs)\ (map\ g\ zs')$, which is a renaming of what we started with, and fold here. This gives a new function h_0 and a call to that function as a final result:

```
letrec $h_0 = \lambda f\ xs'\ g\ ys'.$
   case $xs'$ of
   $[] \rightarrow$ let $ys = map\ g\ ys'$ in $[]$
   $(z : zs) \rightarrow$ case $ys'$ of
                 $[] \rightarrow$ let $x' = f\ z,\ xs' = map\ f\ zs$ in $[]$
                 $(z' : zs') \rightarrow (f\ z, g\ z') : h_0\ f\ zs\ g\ zs'$
in $h_0\ f\ xs'\ g\ ys'$
```

The new function h_0 does not pass any intermediate lists for the common case when both *xs* and *ys* are non-empty. If one of them is empty, it is necessary to run *map* on the remaining part of the other list.

In the introduction we claimed that we can fuse both intermediate lists and both intermediate trees when zipping a list or a tree. The second example requires some new definitions of *map* and *zip* over a simple tree datatype:

```
$data\ \ Tree\ a = Node\ (Tree\ a)\ a\ (Tree\ a) \mid Empty$

$mapT = \lambda f\ xs.$ case $xs$ of
                 $Empty \rightarrow Empty$
                 $Node\ l\ a\ r \rightarrow Node\ (mapT\ f\ l)\ (f\ a)\ (mapT\ f\ r)$
$zipT\ \ = \lambda xs\ ys.$
        case $xs$ of
          $Empty \rightarrow Empty$
          $Node\ l\ a\ r \rightarrow$
              case $ys$ of
                $Empty \rightarrow Empty$
                $Node\ l'\ a'\ r' \rightarrow Node\ (zipT\ l\ l')\ (a,\ a')\ (zipT\ r\ r')$
```

We transform the expression $zipT\ (mapT\ f\ xs)\ (mapT\ g\ ys)$, which applies f to the first tree, g to the second tree and create a final tree whose nodes consists of pairs of the data from the two intermediate trees. We start our transformation by allocating a new fresh function name (h_1) and repeat many of the transformation steps that we just saw for the list case. The end result is:

```
letrec $h_1 = \lambda f\ xs\ g\ ys.$
  case $xs$ of
   $Empty \rightarrow$ let $ys' = mapT\ g\ ys$ in $Empty$
   $Node\ l\ a\ r \rightarrow$
     case $ys$ of
      $Empty \rightarrow$ let $l_1 = mapT\ f\ l,\ a_1 = f\ a$
                    $r_1 = mapT\ f\ r$ in $Empty$
      $Node\ l'\ a'\ r' \rightarrow Node\ (h_1\ f\ l\ g\ l')\ (f\ a,\ g\ a')\ (h_1\ f\ r\ g\ r')$
in $h_1\ f\ xs\ g\ ys$
```

The same result as in the list case: the new function h_1 does not pass any intermediate trees for the common case: when both *xs* and *ys* are non-empty. If one of them is empty, it is necessary to run *mapT* on the remaining part of the other tree. This example also highlights the need to discard unused let-bindings.

The third example of how the new algorithm improves the strength of supercompilation for call-by-value languages is non-linear occurrences of variables, such as in

$$\textbf{let}\, x \;=\; e \;\textbf{in}\, \textit{fst}\;(x,\; x)$$

Our previous algorithm would separately transform e and *fst (x, x)* which would result in $\textbf{let}\, x \;=\; e' \;\textbf{in}\, x$, where it is obvious that x is linear. Our improved algorithm instead inlines *fst* without touching e:

$$\textbf{let}\, x \;=\; e \;\textbf{in}\,\textbf{case}\,(x,\; x)\;\textbf{of}\,\{\;(x,\; y) \;\rightarrow\; x\;\}$$

The algorithm continues to transform the case-expression giving a let-expression that is linear in x: $\textbf{let}\, x \;=\; e \;\textbf{in}\, x$. This expression can be transformed to e and the supercompiler can continue to transform e, having eliminated the entire let-expression in the initial program.

3 Language

Our language of study is a strict, higher-order, functional language with let-bindings and case-expressions. Its syntax for expressions and values is shown in Figure 1.

Expressions

$$\begin{array}{ll} e,\, f ::= x \mid g \mid k\,\overline{e} \mid f\,e \mid \lambda x.e \mid \textbf{case}\, e\, \textbf{of}\, \{k_i\,\overline{x}_i \rightarrow e_i\} \mid \textbf{let}\, x = e\, \textbf{in}\, f \\ \quad\;\; \mid \;\textbf{letrec}\, g = v\, \textbf{in}\, e \end{array}$$

Values

$$v \quad ::= \lambda x.e \mid k\,\overline{v}$$

Fig. 1. The language

Let X be an enumerable set of variables ranged over by x and K a set of constructor symbols ranged over by k. Let g range over an enumerable set of defined names and let $\mathcal{G}$ be a given set of recursive definitions of the form (g, v).

We abbreviate a list of expressions $e_1 \ldots e_n$ as $\overline{e}$, and a list of variables $x_1 \ldots x_n$ as $\overline{x}$. We denote the set of ordered free variables of an expression e by $\mathit{fv}(e)$, and the function names by $\mathit{fn}(e)$.

A program is an expression with no free variables and all function names defined in $\mathcal{G}$. The intended operational semantics is given in Figure 2, where $[\overline{e}/\overline{x}]e'$ is the capture-free substitution of expressions $\overline{e}$ for variables $\overline{x}$ in e'.

A reduction context $\mathcal{E}$ is a term containing a single hole $[\,]$, which indicates the next expression to be reduced. The expression $\mathcal{E}\langle e\rangle$ is the term obtained by

Reduction contexts

$$\mathcal{E} ::= [\,] \mid \mathcal{E}\,e \mid (\lambda x.e)\,\mathcal{E} \mid k\,\overline{\mathcal{E}} \mid \textbf{case}\ \mathcal{E}\ \textbf{of}\ \{p_i \to e_i\} \mid \textbf{let}\,x = \mathcal{E}\ \textbf{in}\,e$$

Evaluation relation

$$\begin{array}{lll}
\mathcal{E}\langle g\rangle & \mapsto \mathcal{E}\langle v\rangle & \text{(Global)}\\
 & \quad \text{if } (g,v) \in \mathcal{G} & \\
\mathcal{E}\langle(\lambda x.e)\,v\rangle & \mapsto \mathcal{E}\langle[v/x]e\rangle & \text{(App)}\\
\mathcal{E}\langle\textbf{let}\,x = v\,\textbf{in}\,e\rangle & \mapsto \mathcal{E}\langle[v/x]e\rangle & \text{(Let)}\\
\mathcal{E}\langle\textbf{letrec}\,g = v\,\textbf{in}\,f\rangle & \mapsto \mathcal{E}\langle[\textbf{letrec}\,g = v\,\textbf{in}\,v/g]f\rangle & \text{(Letrec)}\\
\mathcal{E}\langle\textbf{case}\,k_j\,\overline{v}\,\textbf{of}\ \{k_i\,\overline{x}_i \to e_i\}\rangle & \mapsto \mathcal{E}\langle[\overline{v}/\overline{x}_j]e_j\rangle & \text{(KCase)}
\end{array}$$

Fig. 2. Reduction semantics

replacing the hole in $\mathcal{E}$ with e. $\overline{\mathcal{E}}$ denotes a list of terms with just a single hole, evaluated from left to right.

If a free variable appears no more than once in a term, that term is said to be *linear* with respect to that variable. Like Wadler [1], we extend the definition slightly for linear case expressions: no variable may appear in both the scrutinee and a branch, although a variable may appear in more than one branch.

4 Positive Supercompilation

This section presents an algorithm for positive supercompilation for a higher-order call-by-value language, which removes more intermediate structures than previous work [4].

Our supercompiler is defined as a set of rewrite rules that pattern-match on expressions. This algorithm is called the *driving* algorithm, and is defined in Figure 3. Three additional parameters appear as subscripts to the rewrite rules: a memoization list ρ, a driving context $\mathcal{R}$, and an ordered set $\mathcal{B}$ of expressions bound to variables ($x_1 = e_1$, $x_2 = e_2, \ldots$). We use the short-hand notation **let** $\mathcal{B}$ **in** e for **let** $x_1 = e_1$ **in let** $x_2 = e_2$ **in** .. **in** e. The memoization list holds information about expressions already traversed and is explained more in detail in Section 4.1. An important detail is that our driving algorithm immediately performs the program extraction instead of producing a process tree. The driving context $\mathcal{R}$ is an evaluation context for a call-by-name language:

$$\mathcal{R} ::= [\,] \mid \mathcal{R}\,e \mid \textbf{case}\,\mathcal{R}\ \textbf{of}\ \{p_i \to e_i\}$$

An expression e is strict with regards to a variable x if it eventually evaluates x; in other words, if $e \mapsto \ldots \mapsto \mathcal{E}\langle x\rangle$. Such information is not decidable in general, although call-by-value semantics allows for reasonably tight approximations. One such approximation is given in Figure 4, where the strict variables of an expression e are defined as all free variables of e except those that only

$$
\begin{array}{lll}
\mathcal{D}[\![x]\!]_{[],\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\,\mathcal{D}[\![\mathcal{B}]\!]_{[],\emptyset,\mathcal{G},\rho}\,\textbf{in}\,x & \text{(R1)}\\
\mathcal{D}[\![g]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}_{app}(g)_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & \text{(R2)}\\
\mathcal{D}[\![k\,\overline{e}]\!]_{[],\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\,\mathcal{D}[\![\mathcal{B}]\!]_{[],\emptyset,\mathcal{G},\rho}\,\textbf{in}\,k\,\mathcal{D}[\![\overline{e}]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R3)}\\
\mathcal{D}[\![x\,\overline{e}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\,\mathcal{D}[\![\mathcal{B}]\!]_{[],\emptyset,\mathcal{G},\rho}\,\textbf{in}\,\mathcal{R}\langle x\,\mathcal{D}[\![\overline{e}]\!]_{[],\emptyset,\mathcal{G},\rho}\rangle & \text{(R4)}\\
\mathcal{D}[\![\lambda\overline{x}.e]\!]_{[],\mathcal{B},\mathcal{G},\rho} & = (\lambda\overline{x}.\mathcal{D}[\![e]\!]_{[],\mathcal{B},\mathcal{G},\rho}) & \text{(R5)}\\
\mathcal{D}[\![(\lambda\overline{x}.f)\,\overline{e}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\,\overline{x}=\overline{e}\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & \text{(R6)}\\
\mathcal{D}[\![e\,e']\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![e]\!]_{\mathcal{R}\langle[]\,e'\rangle,\mathcal{B},\mathcal{G},\rho} & \text{(R7)}\\
\mathcal{D}[\![\textbf{let}\,x=v\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\,\mathcal{B}\,\textbf{in}\,\mathcal{R}\langle[v/x]f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R8)}\\
\mathcal{D}[\![\textbf{let}\,x=y\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\,\mathcal{B}\,\textbf{in}\,\mathcal{R}\langle[y/x]f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R9)}\\
\mathcal{D}[\![\textbf{let}\,x=e\,\textbf{in}\,f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\,\mathcal{B}\,\textbf{in}\,\mathcal{R}\langle[e/x]f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho},\ \text{if } x\in \mathit{strict}(f) & \text{(R10)}\\
 & \qquad\qquad\qquad\qquad\qquad\qquad \text{and } x\in \mathit{linear}(f) & \\
 & \quad \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{B}\oplus x=e,\mathcal{G},\rho},\ \text{otherwise} & \\
\mathcal{D}[\![\textbf{letrec}\,g=v\,\textbf{in}\,e]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \textbf{letrec}\,g=v\,\textbf{in}\,e',\ \text{if } g\in \mathit{fn}(e') & \text{(R11)}\\
 & \quad e',\ \text{otherwise} & \\
 & \quad \text{where } e' = \mathcal{D}[\![\textbf{let}\,\mathcal{B}\,\textbf{in}\,\mathcal{R}\langle e\rangle]\!]_{[],\emptyset,\mathcal{G}',\rho} & \\
 & \qquad\qquad \mathcal{G}' = \mathcal{G}\cup(g,v) & \\
\mathcal{D}[\![\textbf{case}\,x\ \textbf{of}\,\{k_i\,\overline{x}_i\to e_i\}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\,\mathcal{D}[\![\mathcal{B}|x]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R12)}\\
 & \quad \textbf{in}\,\textbf{case}\,x\ \textbf{of}\,\{ & \\
 & \qquad k_i\,\overline{x}_i\to \mathcal{D}[\![[k_i\,\overline{x}_i/x]\textbf{let}\,\mathcal{B}\backslash x\,\textbf{in}\,\mathcal{R}\langle e_i\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \\
 & \qquad \} & \\
\mathcal{D}[\![\textbf{case}\,k_j\,\overline{e}\,\textbf{of}\,\{k_i\,\overline{x}_i\to e_i\}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![\textbf{let}\,\mathcal{B}\,\textbf{in}\,\mathcal{R}\langle\textbf{let}\,\overline{x}_j=\overline{e}\,\textbf{in}\,e_j\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R13)}\\
\mathcal{D}[\![\textbf{case}\,x\,\overline{e}\,\textbf{of}\,\{k_i\,\overline{x}_i\to e_i\}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \textbf{let}\,\mathcal{D}[\![\mathcal{B}|(\mathit{fv}(\overline{e})\cup\{x\})]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{(R14)}\\
 & \quad \textbf{in}\,\textbf{case}\,x\,\mathcal{D}[\![\overline{e}]\!]_{[],\emptyset,\mathcal{G},\rho}\,\textbf{of}\,\{ & \\
 & \qquad k_i\,\overline{x}_i\to \mathcal{D}[\![\textbf{let}\,\mathcal{B}\backslash(\mathit{fv}(\overline{e})\cup\{x\})\,\textbf{in}\,\mathcal{R}\langle e_i\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \\
 & \qquad \} & \\
\mathcal{D}[\![\textbf{case}\,e\ \textbf{of}\,\{k_i\,\overline{x}_i\to e_i\}]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} & = \mathcal{D}[\![e]\!]_{\mathcal{R}\langle\textbf{case}\,[]\,\textbf{of}\,\{k_i\,\overline{x}_i\to e_i\}\rangle,\mathcal{B},\mathcal{G},\rho} & \text{(R15)}
\end{array}
$$

Fig. 3. Driving algorithm

appear under a lambda or not inside all branches of a case. Our experience is that this approximation is sufficient in practice.

The rules of the driving algorithm are ordered; i.e., all rules must be tried in the order they appear. Rule R7 and rule R15 are the default fallback cases which extend the given driving context $\mathcal{R}$ and zoom in on the next expression to drive. Notice how rule R8 recursively applies the driving algorithm to the entire new term $\textbf{let}\,\mathcal{B}\,\textbf{in}\,\mathcal{R}\langle[v/x]f\rangle$, forcing a re-traversal of the new term with the hope of further reductions.

Some expressions should be handled differently depending on their context. If a constructor application appears in an empty context, there is not much we can do except to drive the argument expressions (rule R3). On the other hand, if the application occurs at the head of a case expression, we may choose a branch based on the constructor and leave the arguments unevaluated in the hope of finding fold opportunities further down the syntax tree (rule R13).

Rule R12 and rule R14 uses some new notation: $\mathcal{B}|x$ is the smallest prefix of $\mathcal{B}$ that is necessary to define x and $\mathcal{B}\backslash x$ is the largest suffix not necessary to

$$\begin{array}{ll}
\mathit{strict}(x) & = \{x\} \\
\mathit{strict}(g) & = \emptyset \\
\mathit{strict}(k\,\overline{e}) & = \mathit{strict}(\overline{e}) \\
\mathit{strict}(\lambda x.e) & = \emptyset \\
\mathit{strict}(f\;e) & = \mathit{strict}(f) \cup \mathit{strict}(e) \\
\mathit{strict}(\mathbf{let}\,x = e\,\mathbf{in}\,f) & = \mathit{strict}(e) \cup (\mathit{strict}(f) \backslash \{x\}) \\
\mathit{strict}(\mathbf{letrec}\,g = v\,\mathbf{in}\,f) & = \mathit{strict}(f) \\
\mathit{strict}(\mathbf{case}\,e\,\mathbf{of}\,\{k_i\,\overline{x}_i \rightarrow e_i\}) & = \mathit{strict}(e) \cup (\bigcap(\mathit{strict}(e_i) \backslash \overline{x}_i))
\end{array}$$

Fig. 4. The strict variables of an expression

define x. Rule R10 uses $\oplus$ which we define as:

$$\begin{aligned}
(\mathcal{B}, y = e) \oplus x = e' &\overset{\text{def}}{=} (\mathcal{B} \oplus x = e'), y = e \text{ if y} \notin \mathit{fv}(e') \\
&\overset{\text{def}}{=} (\mathcal{B}, y = e, x = e') \text{ otherwise}
\end{aligned}$$

The key idea in this improved supercompilation algorithm is to float let-expressions into the branches of case-expressions. We accomplish this by adding the bound expressions from let-expressions to our binding set $\mathcal{B}$ in rule R10. We make sure that we do not change the order between definition and usage of variables in rule R8 by extracting the necessary bindings outside of the case-expression, and the remaining independent bindings are brought into all the branches along with the surrounding context $\mathcal{R}$.

The algorithm is allowed to move let-expressions into case-branches since that transformation only changes the evaluation order, and non-termination is the only effect present in our language.

4.1 Application Rule

Our extension does not require any major changes to the folding mechanism that supercompilers use to ensure termination. Since our goal is not to study termination properties of supercompilers we present a simplified version of the folding mechanism which does not guarantee termination, but guarantees that if the algorithm terminates the output is correct. The standard techniques [8,9,4] for ensuring termination can be used with our extension.

In the driving algorithm rule R2 refer to $\mathcal{D}_{app}(\,)$, defined in Figure 5. $\mathcal{D}_{app}(\,)$ can be inlined in the definition of the driving algorithm, it is merely given a separate name for improved clarity of the presentation. Figure 5 contains some new notation: we use σ for a variable to variable substitution and $=$ for syntactical equivalence of expressions.

The driving algorithm keeps a record of previously encountered applications in the memoization list ρ; whenever it detects an expression that is equivalent (up to renaming of variables) to a previous expression, the algorithm creates a new recursive function h_n for some n. Whenever an expression from the memoization list is encountered, a call to h_n is inserted.

$$\begin{array}{lll}
\mathcal{D}_{app}(g)_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} = h\,\overline{x} & \text{if } \exists (h,t) \in \rho \,.\, \sigma t = \mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle g\rangle & (1) \\
\quad \text{where } \overline{x} = \sigma(fv(t)) & & \\
\mathcal{D}_{app}(g)_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} = \mathbf{letrec}\,h = \lambda\overline{x}.e'\,\mathbf{in}\,h\,\overline{x} & \text{if } h \in fn(e') & (2a) \\
\qquad\qquad\qquad\qquad e' & \text{otherwise} & (2b) \\
\quad \text{where } (g,v) \in \mathcal{G}, & & \\
\qquad e' = \mathcal{D}[\![\mathcal{R}\langle v\rangle]\!]_{[],\mathcal{B},\mathcal{G},\rho'}, & & \\
\qquad \overline{x} = fv(\mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle g\rangle), & & \\
\qquad \rho' = \rho \cup (h, \mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle g\rangle) \text{ and} & & \\
\qquad h \text{ fresh} & &
\end{array}$$

Fig. 5. Driving of applications

5 Removing Unnecessary Traversals

The first example in Section 2 showed that there might be let-expressions in case-branches where the computed results are never used in the branch. This gives worse runtime performance than necessary since more intermediate results have to be computed, and also increases the compilation time since there are more expressions to transform. The only reason to have these let-expressions is to preserve the termination properties of the input program.

We could remove these superfluous let-expressions if we knew that they were terminating, something that would save both transformation time and execution time. It is clear that termination is undecidable in general, but our experience is that the functions that appear in practice are often recursive over the input structure. Functions with this property are quite well suited for termination analysis, for example the size-change principle [10,11].

Given a function *terminates*(e) that returns true if the expression e terminates, we can augment the let-rule (R10) to incorporate the termination information and discard such expressions, shown in Figure 6. This allows the supercompiler to discard unused expressions, i.e dead code, which saves both transformation time and runtime.

$$\begin{array}{lll}
\mathcal{D}[\![\mathbf{let}\,x = e\,\mathbf{in}\,f]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho} = & \mathcal{D}[\![\mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{if } terminates(\mathrm{e}) \text{ and } x \notin fv(f) \\
& \mathcal{D}[\![\mathbf{let}\,\mathcal{B}\,\mathbf{in}\,\mathcal{R}\langle [e/x]f\rangle]\!]_{[],\emptyset,\mathcal{G},\rho} & \text{if } x \in strict(f) \text{ and } x \in linear(f) \\
& \mathcal{D}[\![\mathcal{R}\langle f\rangle]\!]_{[],\mathcal{B}\oplus x=e,\mathcal{G},\rho} & \text{otherwise}
\end{array}$$

Fig. 6. Extended Let-rule (R10)

Since we leave the choice of termination analysis open, it is hard to discuss scalability in general. The size-change principle has been used with good results in partial evaluation of large logic programs [12] and there are also polynomial time algorithms for approximating termination [13].

6 Correctness

We need a class of expressions w that is essentially the union of the expressions on weak head normal form (whnf) and expressions where the redex contains a free variable, a:

$$a ::= x \mid a\,\overline{e}$$
$$w ::= \lambda x.e \mid k\,\overline{e} \mid a$$

Lemma 1 (Totality). *Let $\mathcal{R}\langle e\rangle$ be an expression such that*

- *$\mathcal{R}\langle e\rangle$ is well-typed*
- *if $\mathcal{R} \neq [\,]$ then $e \neq w$*

then $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho}$ is matched by a unique rule in Figure 3.

Proof. Follows the structure of the proof by Jonsson and Nordlander [14].

To prove that the algorithm does not alter the semantics we use the improvement theory [15]. We define the standard notions of operational approximation and equivalence and introduce a general context C which has zero or more holes in the place of some subexpressions.

Definition 1 (Operational Approximation and Equivalence).

- *e operationally approximates e', $e \sqsubseteq\!\!\!\!\sim e'$, if for all contexts C such that C[e] and C[e'] are closed, if evaluation of C[e] terminates then so does evaluation of C[e'].*
- *e is operationally equivalent to e', $e \cong e'$, if $e \sqsubseteq\!\!\!\!\sim e'$ and $e' \sqsubseteq\!\!\!\!\sim e$*

We use Sands's definitions for improvement and strong improvement:

Definition 2 (Improvement, Strong Improvement).

- *e is improved by e', $e \trianglerighteq e'$, if for all contexts C such that C[e], C[e'] are closed, if computation of C[e] terminates using n calls to named functions, then computation of C[e'] also terminates, and uses no more than n calls to named functions.*
- *e is strongly improved by e', $e \trianglerighteq_s e'$, iff $e \trianglerighteq e'$ and $e \cong e'$.*

which allows us to state the final theorem:

Theorem 1 (Total Correctness). *Let $\mathcal{R}\langle e\rangle$ be an expression, and ρ an environment such that*

- *the range of ρ contains only closed expressions, and*
- *$fv(\mathcal{R}\langle e\rangle) \cap dom(\rho) = \emptyset$, and*
- *if $\mathcal{R} \neq [\,]$ then $e \neq w$*
- *the supercompiler $\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho}$ terminates*

then $\mathcal{R}\langle e\rangle \trianglerighteq_s \rho(\mathcal{D}[\![e]\!]_{\mathcal{R},\mathcal{B},\mathcal{G},\rho})$.

Proof. Similar to the total correctness proof by Jonsson and Nordlander [14].

Recall the simplified algorithm we have presented preserves the semantics only if it terminates; however, termination of the supercompiler can be recovered using a similar $\mathcal{D}_{app}(\,)$ as we did in our previous work [4].

7 Performance and Limitations

There are two aspects of performance that are interesting to the end user: how long the optimization takes; and how much faster the optimized program is.

The work on supercompiling Haskell by Mitchell and Runciman [9] shows that some problems remain for supercompiling large Haskell programs. These problems are mainly related to speed, both of the compiler and of the transformed program. When they profiled their supercompiler they found that the majority of the time was spent in the homeomorphic embedding test, the test which is used to ensure termination.

Our preliminary measurements show the same thing: a large proportion of the time spent on supercompiling a program is spent testing for non-termination of the supercompiler. This paper presents a stronger supercompiler at the cost of larger expressions to test for the homeomorphic embedding. We estimate that our current work ends up somewhere between Supero and our previous work with respect to transformation time since we are testing smaller expressions than Supero, at the expense of runtime performance.

The complexity of the homeomorphic embedding has been investigated separately by Narendran and Stillman [16] and they give an algorithm that takes two terms e and f and decides if there is a risk of non-termination in time $O(size(e) \times size(f))$.

For the second dimension: it is well known that programs with many intermediate lists have worse performance than their corresponding listless versions [6]. We have shown that the output from our supercompiler does not contain intermediate structures by manual transformations in Section 2. It is reasonable to conclude that these programs would perform better in a microbenchmark. We leave the question of performance of large real world programs open.

A limitation of our work is that there are still examples that our algorithm does not give the desired output for. Given $\textbf{let}\, x = (\lambda y.y)\, 1 \,\textbf{in}\, (x,\, x)$ a human can see that the result after transformation should be $(1,\, 1)$, but our supercompiler will produce $\textbf{let}\, x = 1 \,\textbf{in}\, (x,\, x)$. Mitchell [17][Sec 4.2.2] has a strategy to handle this, but we have not been able to incorporate his solution without severely increasing the amount of testing for non-termination done with the homeomorphic embedding. The reason we can not transform $(\lambda y.y)\, 1$ in isolation and then substitute the result is that the result might contain freshly generated function names, which might cause the supercompiler to loop.

8 Related Work

8.1 Deforestation

Deforestation is a slightly weaker transformation than supercompilation [18]. Deforestation algorithms for call-by-name languages can remove all intermediate structures from the examples we outlined in Section 1.

Deforestation was pioneered by Wadler [1] for a first order language more than fifteen years ago. The initial deforestation had support for higher order macros which are incapable of fully emulating higher order functions.

Marlow and Wadler [19] addressed the restriction to a first-order language when they presented a deforestation algorithm for a higher order language. This work was refined in Marlow's [20] dissertation, where he also related deforestation to the cut-elimination principle of logic. Chin [21] has also generalised Wadler's deforestation to higher-order functional programs by using syntactic properties to decide which terms can be fused.

Both Hamilton [22] and Marlow [20] have proven that their deforestation algorithms terminate. More recent work by Hamilton [23] extends deforestation to handle a wider range of functions, with an easy-to-recognise treeless form, giving more transparency for the programmer.

Alimarine and Smetsers [24] have improved the producer and consumer analyses in Chin's [21] algorithm by basing them on semantics rather than syntax. They show that their algorithm can remove much of the overhead introduced from generic programming [25].

8.2 Supercompilation

Except for our previous work [4], the work on supercompilation has been for call-by-name semantics. All call-by-name supercompilers succeed on the examples we outlined in Section 1, and are very close algorithmically to our current work.

Supercompilation [26,27,28,29] removes intermediate structures and achieves partial evaluation as well as some other optimisations. Scp4 [30] is the most well-known implementation from this line of work.

The *positive supercompiler* [2] is a variant which only propagates positive information, such as equalities. The propagation is done by unification and the work highlights how similar deforestation and positive supercompilation really are. We have previously investigated the theoretical foundations for positive supercompilation for strict languages [4]. Narrowing-driven partial evaluation [31,32] is the functional logic programming equivalent of positive supercompilation but formulated as a term rewriting system. They also deal with non-determinism from backtracking, which makes the algorithm more complicated.

Strengthening the information propagation mechanism to propagate not only positive but also negative information yields *perfect supercompilation* [5,33,34]. Negative information is the opposite of positive information – inequalities. These inequalities can for example be used to prune branches that we can be certain are not taken in case-expressions.

More recently, Mitchell and Runciman [9] have worked on supercompiling Haskell. Their algorithm is closely related to our supercompiler, but their work is limited to call-by-name. They report runtime reductions of up to 55% when their supercompiler is used in conjunction with GHC.

8.3 Short Cut Fusion

Short cut deforestation [35,6] takes a different approach to deforestation, sacrificing some generality by only working on lists.

The idea is that the constructors *Nil* and *Cons* can be replaced by a *foldr* consumer, and a special function *build* is used to allow the transformation to recognize the producer and enforce the type requirement. Lists using *build/foldr* can easily be removed with the *foldr/build* rule: $\mathit{foldr}\ f\ c\ (\mathit{build}\ g)\ =\ g\ f\ c$.

This forces the programmer or compiler writer to make sure list-traversing functions are written using *build* and *foldr*, thereby cluttering the code with information for the optimiser and making it harder to read and understand for humans.

Takano and Meijer [36] generalized short cut deforestation to work for any algebraic datatype through the acid rain theorem. Ghani and Johann [37] have also generalized the *foldr/build* rule to a *fold/superbuild* rule that can eliminate intermediate structures of inductive types without disturbing the contexts in which they are situated.

Launchbury and Sheard [38] worked on automatically transforming programs into suitable form for shortcut deforestation. Onoue et al. [39] showed an implementation of the acid rain theorem for Gofer where they could automatically transform recursive functions into a form suitable for shortcut fusion.

Type-inference can be used to transform the producer of lists into the abstracted form required by short cut deforestation, and this is exactly what Chitil [40] does. Given a type-inference algorithm which infers the most general type, Chitil is able to determine the list constructors that need to be replaced.

Takano and Meijer [36] noted that the foldr/build rule for short cut deforestation had a dual. This is the *destroy/unfoldr* rule used by Svenningsson [41] which has some interesting properties. The method can remove all argument lists from a function which consumes more than one list, addressing one of the main criticisms against the *foldr/build* rule. The technique can also remove intermediate lists from functions which consume their lists using accumulating parameters, a known problematic case that most techniques can not handle.

The method is simple, and can be implemented the same way as short cut deforestation. It still suffers from the drawback that the programmer or compiler writer has to make sure the list traversing functions are written using *destroy* and *unfoldr*.

In more recent work Coutts et al. [7] have extended these techniques to work on functions that handle nested lists, list comprehensions and filter-like functions.

9 Conclusions

We have presented an improved supercompilation algorithm for a higher-order call-by-value language. Our extension is orthogonal to the information propagation by the algorithm. Through examples we have shown that the algorithm can remove multiple intermediate structures, which previous algorithms could not, such as the *zip* examples in Section 2.

9.1 Future Work

We are currently working on improving the scalability of supercompilation for real programs. MLTon [42] has successfully performed whole program compilation of programs up to 100 000 lines, which suggests that any bottlenecks should occur in the supercompiler, not the other parts of the compiler.

An anonymous referee suggested performing on-demand termination analysis on already simplified terms. We are looking into this possibility. Another anonymous referee suggested a characterization, such as the treeless form by Wadler [1], of input terms that would guarantee termination of the supercompiler as specified in this paper.

Acknowledgements The authors would like to thank Germán Vidal for sharing his practical experience with the size-change principle and Duncan Coutts for answering our questions about stream fusion. We would also like to thank Viktor Leijon and the anonymous referees for providing useful comments that helped improve the presentation and contents.

References

1. Wadler, P.: Deforestation: transforming programs to eliminate trees. Theoretical Computer Science **73**(2) (June 1990) 231–248
2. Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming **6**(6) (1996) 811–838
3. Ohori, A., Sasano, I.: Lightweight fusion by fixed point promotion. In: POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2007) 143–154
4. Jonsson, P.A., Nordlander, J.: Positive supercompilation for a higher-order call-by-value language. In: POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. (2009)
5. Glück, R., Klimov, A.: Occam's razor in metacomputation: the notion of a perfect process tree. Lecture Notes in Computer Science **724** (1993) 112–123
6. Gill, A.J.: Cheap Deforestation for Non-strict Functional Languages. PhD thesis, Univ. of Glasgow (January 1996)
7. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: from lists to streams to nothing at all. In: ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM (2007) 315–326
8. Sørensen, M., Glück, R.: An algorithm of generalization in positive supercompilation. In Lloyd, J., ed.: International Logic Programming Symposium, Cambridge, MA: MIT Press (1995) 465–479
9. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In et al., O.C., ed.: Selected Papers from the Proceedings of IFL 2007. Volume 5083 of Lecture Notes in Computer Science., Springer-Verlag (2008) 147–164

10. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL'01. (2001) 81–92
11. Sereni, D.: Termination analysis and call graph construction for higher-order functional programs. In Hinze, R., Ramsey, N., eds.: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007, ACM (2007) 71–84
12. Leuschel, M., Vidal, G.: Fast Offline Partial Evaluation of Large Logic Programs. In: Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'08), Springer LNCS 5438 (2009) 119–134
13. Ben-Amram, A.M., Lee, C.S.: Program termination analysis in polynomial time. ACM Trans. Program. Lang. Syst **29**(1) (2007)
14. Jonsson, P.A., Nordlander, J.: Positive Supercompilation for a Higher Order Call-By-Value Language: Extended Proofs. Technical Report 2008:17, Department of Computer science and Electrical engineering, Luleå University of Technology (October 2008)
15. Sands, D.: From SOS rules to proof principles: An operational metatheory for functional languages. In: Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM Press (January 1997)
16. Narendran, P., Stillman, J.: On the Complexity of Homeomorphic Embeddings. Technical Report 87-8, Computer Science Department, State Univeristy of New York at Albany (March 1987)
17. Mitchell, N.: Transformation and Analysis of Functional Programs. PhD thesis, University of York (June 2008)
18. Sørensen, M., Glück, R., Jones, N.: Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Sannella, D., ed.: Programming Languages and Systems — ESOP'94. 5th European Symposium on Programming, Edinburgh, U.K., April 1994 (Lecture Notes in Computer Science, vol. 788), Berlin: Springer-Verlag (1994) 485–500
19. Marlow, S., Wadler, P.: Deforestation for higher-order functions. In Launchbury, J., Sansom, P.M., eds.: Functional Programming. Workshops in Computing, Springer (1992) 154–165
20. Marlow, S.D.: Deforestation for Higher-Order Functional Programs. PhD thesis, Department of Computing Science, University of Glasgow (April 27 1995)
21. Chin, W.N.: Safe fusion of functional expressions II: Further improvements. J. Funct. Program **4**(4) (1994) 515–555
22. Hamilton, G.W.: Higher order deforestation. In: PLILP '96: Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs, London, UK, Springer-Verlag (1996) 213–227
23. Hamilton, G.W.: Higher order deforestation. Fundam. Informaticae **69**(1-2) (2006) 39–61
24. Alimarine, A., Smetsers, S.: Improved fusion for optimizing generics. In Hermenegildo, M.V., Cabeza, D., eds.: Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10-11, 2005, Proceedings. Volume 3350 of Lecture Notes in Computer Science., Springer (2005) 203–218
25. Hinze, R.: Generic Programs and Proofs. Habilitationsschrift, Bonn University (2000)
26. Turchin, V.: A supercompiler system based on the language Refal. SIGPLAN Notices **14**(2) (February 1979) 46–54

27. Turchin, V.: Semantic definitions in Refal and automatic production of compilers. In Jones, N., ed.: Semantics-Directed Compiler Generation, Aarhus, Denmark (Lecture Notes in Computer Science, vol. 94). Berlin: Springer-Verlag (1980) 441–474
28. Turchin, V.: Program transformation by supercompilation. In Ganzinger, H., Jones, N., eds.: Programs as Data Objects, Copenhagen, Denmark, 1985 (Lecture Notes in Computer Science, vol. 217). Berlin: Springer-Verlag (1986) 257–281
29. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems **8**(3) (July 1986) 292–325
30. Nemytykh, A.P.: The supercompiler SCP4: General structure. In Broy, M., Zamulin, A.V., eds.: Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers. Volume 2890 of LNCS., Springer (2003) 162–170
31. Alpuente, M., Falaschi, M., Vidal, G.: Partial Evaluation of Functional Logic Programs. ACM Transactions on Programming Languages and Systems **20**(4) (1998) 768–844
32. Albert, E., Vidal, G.: The narrowing-driven approach to functional logic program specialization. New Generation Comput **20**(1) (2001) 3–26
33. Secher, J.P.: Perfect supercompilation. Technical Report DIKU-TR-99/1, Department of Computer Science (DIKU), University of Copenhagen (February 1999)
34. Secher, J., Sørensen, M.: On perfect supercompilation. In Bjørner, D., Broy, M., Zamulin, A., eds.: Proceedings of Perspectives of System Informatics. Volume 1755 of Lecture Notes in Computer Science., Springer-Verlag (2000) 113–127
35. Gill, A., Launchbury, J., Peyton Jones, S.: A short cut to deforestation. In: Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, 1993. (1993)
36. Takano, A., Meijer, E.: Shortcut deforestation in calculational form. In: FPCA. (1995) 306–313
37. Ghani, N., Johann, P.: Short cut fusion of recursive programs with computational effects. In Achten, P., Koopman, P., Morazán, M.T., eds.: Draft Proceedings of The Ninth Symposium on Trends in Functional Programming (TFP). Number ICIS–R08007 (2008)
38. Launchbury, J., Sheard, T.: Warm fusion: Deriving build-cata's from recursive definitions. In: FPCA. (1995) 314–323
39. Onoue, Y., Hu, Z., Iwasaki, H., Takeichi, M.: A calculational fusion system HYLO. In Bird, R.S., Meertens, L.G.L.T., eds.: Algorithmic Languages and Calculi, IFIP TC2 WG2.1 International Workshop on Algorithmic Languages and Calculi, 17-22 February 1997, Alsace, France. Volume 95 of IFIP Conference Proceedings., Chapman & Hall (1997) 76–106
40. Chitil, O.: Type-Inference Based Deforestation of Functional Programs. PhD thesis, RWTH Aachen (October 2000)
41. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: ICFP. (2002) 124–132
42. Weeks, S.: Whole-program compilation in MLton. In: ML '06: Proceedings of the 2006 workshop on ML, New York, NY, USA, ACM (2006) 1–1 http://mlton.org/pages/References/attachments/060916-mlton.pdf.

Towards Higher-Level Supercompilation*

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
ilya.klyuchnikov@gmail.com romansa@keldysh.ru

Abstract. We show that the power of supercompilation can be increased by constructing a hierarchy of supercompilers, in which a lower-level supercompiler is used by a higher-level one for proving *improvement lemmas*. The lemmas thus obtained are used to transform expressions labeling nodes in process trees, in order to avoid premature generalizations. Such kind of supercompilation, based on a combination of several metalevels, is called *higher-level* supercompilation (to differentiate it from *higher-order* supercompilation related to transforming higher-order functions). Higher-level supercompilation may be considered as an application of a more general principle of *metasystem transition*.

1 Introduction

The concept of *metasystem transition* was introduced by V.F.Turchin in his 1977 book *The Phenomenon of Science* [26]. In the context of computer science, Turchin gives the following (somewhat simplified) formulation of the main idea of metasystem transition [28]:

> Consider a system S of any kind. Suppose that there is a way to make some number of copies of it, possibly with variations. Suppose that these systems are united into a new system S' which has the systems of the S type as its subsystems, and includes also an additional mechanism which somehow examines, controls, modifies and reproduces the S-subsystems. Then we call S' a *metasystem* with respect to S, and the creation of S' a *metasystem transition*. As a result of consecutive metasystem transitions a multilevel hierarchy of control arises, which exhibits complicated forms of behavior.

Futamura projections [6] may serve as a good example of metasystem transition. Let p be a program, i an interpreter, and s a program specializer. Then $s(i, p)$ may be regarded as a compiled program (the "first projection"), $s(s, i)$ as a compiler (the "second projection") and $s(s, s)$ as a compiler generator (the "third

* Supported by Russian Foundation for Basic Research projects No. 08-07-00280-a and No. 09-01-00834-a.

projection"). (The second Futamura projection is also referred to by Ershov as "Turchin's theorem of double driving" [5].)

In the second projection the evaluation of $s(s, i)$ involves two copies of the specializer s, the second copy "examining and controlling" the first one. In the third projection, there are 3 copies of s, the third one "controlling" the second one "controlling" the first one. Moreover, as shown by Glück [7], there may be considered the "fourth" Futamura projection, corresponding to the next metasystem transition!

Futamura projections, however, are not the only possible way of exploiting the idea of metasystem transition by combining program transformers. In the present paper we consider another technique of constructing a multilevel hierarchy of control using *supercompilers* as its building blocks.

A supercompiler is a program transformer based on *supercompilation* [27], a program transformation technique bearing close relation to the fold/unfold method by Burstall and Darlington [4]. Unfortunately, "pure" supercompilation is known not to be very good at transforming non-linear (i.e. containing repeated variables) expressions and functions with accumulating parameters.

We argue, however, that the power of supercompilation can be increased by combining several copies of a "classic" supercompiler and making them control each other. Such kind of supercompilation, based on a combination of several metalevels will be called *higher-level* supercompilation (to differentiate it from *higher-order* supercompilation related to transforming higher-order functions).

The technique suggested in the paper is (conceptually) simple and modular, and is based on the use of *improvement lemmas* [20,21], which are automatically generated by lower-level supercompilers for a higher-level supercompiler.

2 Higher-Level Supercompilation

2.1 What is a "zero-level" supercompiler?

The descriptions of supercompilation given in the literature differ in some secondary details, irrelevant to the main idea of higher-level supercompilation. We follow the terminology and notation used by Sørensen and Glück [24,23,25].

All program transformation examples considered in the paper have been carried out by HOSC [13,14], a higher-order supercompiler whose general structure is shown in Fig. 1.

2.2 Accumulating parameter: "zero-level" supercompilation

Let us try to apply the supercompiler HOSC [13] to the program shown in Fig. 2. At the beginning, a few steps of driving produce the process tree shown in Fig. 3.

At this point the whistle signals that there is a node b embedding a previously encountered node a, but b is not an instance of a:

```
case double x Z of {Z → True; S y → odd y);}
   ⊴c case double n (S (S Z)) of {Z → True; S m → odd m;}
```

```
def scp(tree)
  b = unprocessed_leaf(tree)
  if b == null
    return makeProgram(tree)
  if trivial(b)
    return scp(drive(b, tree))
  a = ancestor(tree, b, renaming)
  if a != null
    return scp(fold(tree, a, b))
  a = ancestor(tree, b, instance)
  if a != null
    return scp(abstract(tree, b, a))
  a = ancestor(tree, b, whistle)
  if a == null
    return scp(drive(b, tree))
  return scp(abstract(tree, a, b))
```

- `unprocessed_leaf(tree)` returns an unprocessed leaf `b` in the process tree.
- `trivial(b)` checks whether the leaf `b` is "trivial". (A leaf is trivial if driving it does not result in unfolding a function call or applying a substitution to a variable.)
- `drive(b,tree)` performs a driving step for a node `b` and returns the modified tree.
- `ancestor(tree,b,renaming)` returns a node `a` such that `b` is a renaming of `a`.
- `ancestor(tree,b,instance)` returns a node `a` such that `b` is an instance of `a`.
- `ancestor(tree,b,whistle)` returns a node `a` such that `a` is homeomorphically embedded in `b` by coupling.
- `fold(t,a,b)` makes a cycle in the process tree from `b` to `a`.
- `abstract(tree,a,b)` generalizes `a` to `b`.

Fig. 1. "Zero-level" supercompilation algorithm

```
data Bool = True | False;
data Nat = Z | S Nat;

even (double x Z) where

even = λx → case x of { Z → True; S x1 → odd x1;};
odd = λx → case x of { Z → False; S x1 → even x1;};

double = λx y → case x of { Z → y; S x1 → double x1 (S (S y));};
```

Fig. 2. even (double x Z): source program

Hence, HOSC has to throw away the whole subtree under a and "generalize" a by replacing a with a new node a', such that a and b are instances of a'. Then the supercompilation continues to produce the residual program shown in Fig. 4.

This result is correct, but it is not especially exciting! In the goal expression `even (double x Z)` the inner function `double` multiplies its argument by 2, and the outer function `even` checks whether this number is even. Hence, the whole expression never returns `False`. But this can not be immediately seen from the residual program, the program text still containing `False`.

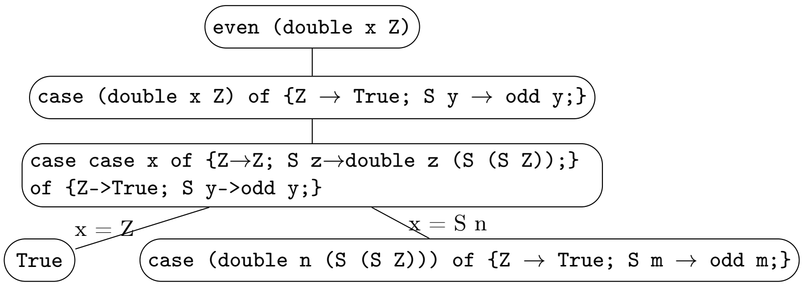

even (double x Z)
case (double x Z) of {Z → True; S y → odd y;}
case case x of {Z→Z; S z→double z (S (S Z));}
of {Z->True; S y->odd y;}
x = Z
x = S n
True
case (double n (S (S Z))) of {Z → True; S m → odd m;}

Fig. 3. even (double x Z): driving

```
letrec
  f=λw2 λp2→
    case w2 of {
      Z →
        letrec g=λr2→
          case r2 of {
            S r → case r of {Z → False; S z2 → g z2;};
            Z → True;}
        in g p2;
      S z → f z (S (S p2));
    }
in f x Z
```

Fig. 4. The result of "zero-level" supercompilation

2.3 Accumulating parameter: applying a lemma

As was pointed out by Burstall and Darlington [4], the power of a program transformation system can be increased by enabling it to use "laws" or "lemmas" (such as the associativity and commutativity of addition and multiplication). In terms of supercompilation it amounts to replacing a node in the process tree with an "equivalent" one.

Let us return to the tree in Fig. 3. The result of supercompilation was not good enough, because HOSC had to generalize a node, and a generalization resulted in a "loss of precision". But we can avoid generalization by making use of the following (mysterious) equivalence

```
case double n (S (S Z)) of {Z → True; S m → odd m;} ≅
    ≅ even (double n Z)
```

Since the node `even (double n Z)` is a renaming of an upper one, the supercompiler can now form a cycle to produce the program shown in Fig. 5. Note that this time there is no occurrences of `False`, hence the supercompiler succeeded in proving that `False` can never be returned by the program.

```
letrec f=λt→
  case t of {Z → True; S s → f s;}
in f x
```

Fig. 5. The result of applying a lemma

```
def scp(tree, n)
  b = unprocessed_leaf(tree)
  if b == null
    return makeProgram(tree)
  if trivial(b)
    return scp(drive(b, tree), n)
  a = ancestor(tree, b, renaming)
  if a != null
    return scp(fold(t, a, b), n)
  a = ancestor(tree, b, instance)
  if a != null
    return scp(abstract(tree, b, a))
  a = ancestor(tree, b, whistle)
  if a == null
    return scp(drive(b, tree), n)
  if n > 0
    e = findEqExpr(b.expr, n)
    if e != null
      return scp(replace(tree, b, e), n)
  return scp(abstract(tree, a, b))
```

```
def findEqExpr(e1, n)
  e = scp(e1, n-1)
  cands = candidates(e1, n)
  for cand <- cands
    if equivalent(
         scp(cand, n-1), e)
      return cand
  return null

def candidates(e1, n)
  ...



                     
  ...
```

Fig. 6. “Multi-level" supercompilation algorithm

Hence, there are good reasons to believe that lemmas are a good thing, but there appear two questions: (1) how to prove lemmas, and (2) how to find useful lemmas.

2.4 Proving lemmas by supercompilation

In [15] we have shown that the supercompiler HOSC [13] may be used for proving interesting equivalences of higher-order expressions. The technique is quite straightforward. Let e_1 and e_2 be expressions appearing in a program p. Let e'_1 and e'_2 be the results of supercompiling e_1 and e_2 with respect to p. Then, if e'_1 and e'_2 are the same (modulo alpha-renaming), then the original expressions e_1 and e_2 are equivalent (provided that the supercompiler strictly preserves the equivalence of programs). Since e_1 and e_2 may contain free variables, the use of a higher-order supercompiler enables us to prove equalities with universal quantification over functions and infinite data types by using a higher-order supercompiler.

```
def scp(tree, n)
  b = unprocessed_leaf(tree)
  if b == null
    return [makeProgram(tree)]
  if trivial(b)
    return scp(drive(b, tree), n)
  a = ancestor(tree, b, renaming)
  if a != null
    return scp(fold(t, a, b), n)
  a = ancestor(tree, b, instance)
  if a != null
    return scp(abstract(tree, b, a))
  a = ancestor(tree, b, whistle)
  if a == null
    return scp(drive(b, tree), n)
  progs = scp(abstract(tree, a, b))
  if n > 0
    for e <- findEqExpr(b.expr, n)
      progs ++= scp(replace(tree, b, e), n)
  return progs
```

```
def findEqExpr(e1, n)
  es = scp(e1, n-1)
  cands = candidates(e1, n)
  exps = []
  for cand <- cands
    if not_disjoint(
          scp(cand, n-1), es)
      exps = exps + cand
  return exps

def candidates(e1, n)
  ...




  ...
```

Fig. 7. “Multi-level" supercompilation algorithm: multiple residual programs

Thus the reasoning about operational equivalence ($\cong$) of programs can be reduced to a trivial check of the syntactic equality of supercompiled programs. Since all residual programs produced by HOSC are expressions (that may contain letrec-subexpressions), checking the equality of residual programs boils down to a syntactical comparison of expressions.

Generally speaking, the idea of proving equivalence by normalization is a well-known one, being a standard technique in such fields as computer algebra. The idea of using supercompilation for normalization is due to Lisitsa and Webster [17], who have successfully applied supercompilation for proving the equivalence of programs written in a first-order functional language, on condition that the programs deal with finite input data and are guaranteed to terminate. Later it has been found [15,13] that these restrictions can be lifted in cases where the supercompiler deals with programs in a lazy functional language and preserves the termination properties of programs.

2.5 Stacking supercompilers, jumping to higher-level supercompilation

As we have seen the power of supercompilation can be increased by using lemmas (i.e. replacing some expressions with equivalent ones). On the other hand, supercompilers can be used for checking the equality of expressions. Hence, recalling the principle of metasystem transition [26,28], we come to the following idea: let us construct a tower of supercompilers, the higher-level ones running the lower-level ones in order to obtain useful lemmas.

This can be done by adding to the function `scp(tree)` shown in Fig. 1 an additional parameter `n`, the "level" the supercompiler is invoked at. The modified supercompilation algorithm is shown in Fig. 6.

Note that for `n = 0` the algorithm degrades to the "classic" supercompilation. But in cases where `n > 0` the supercompiler calls the function `findEqExpr(b.expr,n)` passing to it the expression in the node `b` and the current level. The function tries to produce an expression equivalent to `b.expr` by generating a set of candidate expressions and selecting an expression equivalent to `b.expr`. The check for equivalence is performed by invoking the same supercompiler at the lower level `n - 1`.

2.6 Generating sets of residual programs

The algorithm in Fig. 6 assumes that there is a single result of supercompilation. However, there are certain points in the process of supercompilation, where the supercompiler has an opportunity to make a choice among several options, so that, given a source program, several (equivalent) residual programs may be generated.

This may be used for increasing the power of the supercompilation-based equality check. Suppose we have to check for equivalence two expressions e_1 and e_2. Then, instead of generating and comparing just two residual expressions e'_1 and e'_2, we can supercompile e_1 and e_2 to produce two sets of residual expressions and try to find a residual expression common to both sets (modulo alpha-renaming).

To implement this idea we need a version of a supercompilation algorithm producing a set of residual programs (see Fig. 7). This version, instead of choosing an arbitrary acceptable expression from the set of candidate expressions, returns the set of all acceptable expressions. Note that for any n the set of residual programs produced by the algorithm includes the result returned by the zero-level supercompiler.

2.7 A few open questions

Fig. 6 and Fig. 7 present the general idea of higher-level supercompilation, but there still remains a few open questions.

Correctness At the first glance, replacing an expression with an equivalent one looks as a "natural" and "safe-by-construction" operation. And yet, as shown by Sands [20,21], the unrestricted use of equivalences may lead to incorrect transformations that do not preserve the meaning of programs.

How to generate candidate expressions Supercompilation can be used for checking the equivalence of expressions, but it does not help us in finding candidate expressions that are worth being checked for equivalence.

3 Correctness = equivalence + improvement

3.1 Notation

It is clear that the meaning of expressions may depend on the context. Thus, to avoid making our notation unnecessarily cumbersome, when speaking about the equivalence of expressions, we will assume that the expressions are evaluated and supercompiled in the context of the same program.

We use $\mathcal{SC}[\![e]\!]$ to denote the expression produced by supercompiling the expression e by a "classic", "zero-level" supercompiler, and $e \equiv e'$ to denote the fact that e is the same as e' (modulo alpha-renaming).

3.2 Operational equivalence

Definition 1 (Operational approximation). *An expression e operationally approximates e', $e \underset{\sim}{\sqsubset} e'$, if for all contexts C such that $C[e]$, $C[e']$ are closed, if the evaluation of $C[e]$ terminates then so does the evaluation of $C[e']$.*

Definition 2 (Operational equivalence). *An expression e is operationally equivalent to e', $e \cong e'$ if $e \underset{\sim}{\sqsubset} e'$ and $e' \underset{\sim}{\sqsubset} e$.*

In the following we assume the supercompilers to preserve operational equivalence, i.e. that $e' = \mathcal{SC}[\![e]\!]$ implies $e' \cong \mathcal{SC}[\![e]\!]$ (which is true of the supercompiler HOSC [13]).

3.3 Improvement

The replacement of an expression e with an equivalent expression e', followed by a fold, may result in producing an incorrect residual program (some examples can be found in [21]).

Definition 3 (Improvement). *An expression e is improved by e', $e \underset{\sim}{\triangleright} e'$, if for all contexts C such that $C[e]$ and $C[e']$ are closed, if the computation of $C[e]$ terminates using n function calls, then the computation of $C[e']$ also terminates, and uses no more than n function calls.*

As has been shown by Sands [21], the replacement of an expression e_1 with an expression e_2 will not violate the correctness of transformation if the following conditions are met: $e_1 \cong e_2$ and $e_1 \underset{\sim}{\triangleright} e_2$.

Definition 4 (Improvement lemma). *A pair (e_1, e_2) is an improvement lemma if $e_1 \cong e_2$ and $e_1 \underset{\sim}{\triangleright} e_2$.*

3.4 Checking the improvement relation by supercompilation

Let e_1 and e_2 be expressions whose equivalence has been proven by supercompilation. Does it mean that one of the expressions is an improvement over the other one? Not at all!

Supercompiling the following two expressions with respect to the program shown in Fig. 10 proves them to be operationally equivalent:

```
or (even n) (odd n)
  ≅ case (even n) of {True → True; False → odd (S (S n));}
```

However, neither is an improvement over another one. Indeed, if $n =$ `Z`, the evaluation of the expressions involves 2 and 1 function calls, respectively. But, if $n =$ `S Z`, the evaluation involves 5 and 6 function calls. Therefore, this lemma is unsafe to be used in program transformation.

Fortunately, the check that $e_1 \mathrel{\underset{\sim}{\trianglerighteq}} e_2$ holds for two expressions e_1 and e_2, such that $e_1 \cong e_2$, can also be performed by supercompilation! And this can be done almost for free in the following way.

In order to check e_1 and e_2 for equivalence, we have to supercompile them to e_1' and e_2'. The check for equivalence succeeds if e_1' and e_2' are the same (modulo alpha-renaming). Therefore, e_1' and e_2' contain insufficient information to make any conclusions about e_2 being an improvement over e_1. However, the process trees produced by supercompiling e_1 and e_2 contain more information than the residual expressions.

Namely, let us examine the process trees and mark with a star (*) the edges corresponding to an unfolding (a function call). For example, supercompiling the expressions considered above produces the process trees shown in Fig. 12 and Fig. 14 (the subtrees for `odd x` are omitted for brevity). Now the starred edges provide some information that can be used for checking that an expression improves another one. But this information is not accessible from outside. But, in the case of HOSC [13], this information can be made visible by modifying the algorithm that converts process trees into residual expression.

The modified algorithm converts starred edges into annotations in residual expressions. When traversing a starred edge, the residual expression produced by traversing the (single) child node is annotated with a star (*). In this way the information about unfolds is recorded in residual expressions, so that a lower-level supercompiler can be used as a “black box”.

For example, Fig. 13 and Fig. 16 show the annotated programs produced from the process trees in Fig. 12 and Fig. 14.

Now, let $\mathrel{\underset{\sim}{\trianglerighteq}}^*$ denote a binary relation on expressions such that $e \mathrel{\underset{\sim}{\trianglerighteq}}^* e'$ iff (1) e and e' differ only in their annotations, and (2) e can be transformed into e' by erasing some stars in e. See Fig. 8 for a more formal definition, where ${}^n\phi$ denotes a functor ϕ prefixed with n stars. (It is curious to note that $\mathrel{\underset{\sim}{\trianglerighteq}}^*$ can be considered as a special case of homeomorphic embedding relation.)

Theorem 1. *Let $e_1' = \mathcal{SC}[\![e_1]\!]$ and $e_2' = \mathcal{SC}[\![e_1]\!]$. If $e_1' \equiv e_2'$ and $e_1' \mathrel{\underset{\sim}{\trianglerighteq}}^* e_2'$, then $e_1 \mathrel{\underset{\sim}{\trianglerighteq}} e_2$.*

$$\frac{m \geq n \qquad \forall i : e_i \mathrel{\underset{\sim}{\rhd}^*} e'_i}{{}^m\phi(e_1, \ldots, e_k) \mathrel{\underset{\sim}{\rhd}^*} {}^n\phi(e'_1, \ldots, e'_k)} \qquad \frac{\mathcal{SC}[\![e_1]\!] \cong \mathcal{SC}[\![e_2]\!] \qquad \mathcal{SC}[\![e_1]\!] \mathrel{\underset{\sim}{\rhd}^*} \mathcal{SC}[\![e_2]\!]}{e_1 \mathrel{\underset{\sim}{\rhd}} e_2}$$

Fig. 8. Estimation of improvement based on annotated supercompiled expressions

Proof. Since $e'_1 \equiv e'_2$, e'_1 is the same as e'_2 (modulo annotations and a bound variable renaming). Therefore, if we put the expressions in the same context C and try to evaluate $C[e'_1]$ and $C[e'_2]$ (disregarding the annotations), this will result in two sequences of reduction steps, differing only in the number of stars encountered during computation. Since $e'_1 \mathrel{\underset{\sim}{\rhd}^*} e'_2$, after any number of reduction steps, the number of stars encountered in the evaluation of $C[e'_2]$ cannot be greater that the number of stars encountered in the evaluation of $C[e'_1]$, and the stars correspond to unfoldings in the original expressions e_1 and e_2. So, by evaluating $C[e'_1]$ and $C[e'_2]$ and counting stars we can count the number of unfolds in the evaluation of $C[e_1]$ and $C[e_2]$. Hence, the number of unfolds in the evaluation of $C[e_2]$ is no more than in the evaluation of $C[e_1]$. Therefore $e_1 \mathrel{\underset{\sim}{\rhd}} e_2$.

Thus, by examining annotated supercompiled expressions, we can check the improvement relation for the original expressions. For example, consider the annotated supercompiled expressions for

```
or (even n) (odd n)
```

and

```
case (even n) of {True → True; False → odd (S (S n));}
```

shown in Fig. 13 and Fig. 16. Since the supercompiled expressions are not related by $\underset{\sim}{\rhd}^*$, we cannot make the conclusion that the original expressions are related by $\underset{\sim}{\rhd}$.

4 A proof-of-concept implementation

Although, the general idea of higher-level supercompilation is conceptually simple, there are a number of problems to be solved in a practical implementation.

- Which "zero-level" supercompiler to use as the basic for implementing higher-level supercompilation?
- How to guarantee the correctness of transformations?
- How to generate useful lemmas?
- How to ensure the termination of higher-level supercompilation?

To show the feasibility of higher-level supercompilation we have implemented a simple "proof-of-concept" higher-level supercompiler HLSC by modifying the supercompiler HOSC [13,15]. HOSC has been chosen because it (1) preserves the meaning of programs (including their termination properties), (2) is able to prove lemmas with universal quantification over functions and infinite data types, (3) generates residual programs in the form of expressions, which enables the program equivalence checking to be reduced to expression equivalence checking.

The correctness of the transformations is guaranteed, since HLSC uses lemmas that are improvement ones. Note, however, that the check for improvement is based on "zero-level" supercompilation, for which reason HLSC currently implements only a two-level hierarchy of supercompilers, rather than a multi-level one, consisting of the "top" and "bottom" supercompilers.

The least elaborated points are the search for useful lemmas and ensuring the termination of higher-level supercompilation.

$$\begin{array}{ll}
\mathcal{S}[\![v]\!] & = 1 \\
\mathcal{S}[\![c\ \overline{e_i}]\!] & = 1 + \sum_i \mathcal{S}[\![e_i]\!] \\
\mathcal{S}[\![\lambda v \to e]\!] & = 1 + \mathcal{S}[\![e]\!] \\
\mathcal{S}[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] & = 1 + \mathcal{S}[\![e_0]\!] + \sum_i \mathcal{S}[\![e_i]\!] \\
\mathcal{S}[\![e_1\ e_2]\!] & = \mathcal{S}[\![e_1]\!] + \mathcal{S}[\![e_2]\!]
\end{array}$$

Fig. 9. The size of expression

Presently the generation of candidate expressions is implemented in a rather crude and straightforward way. When the top supercompiler finds a node b containing an expression e and embedding a previously encountered node a, it generates and tries all expressions e' whose size (see Fig. 9) is less than the size of e. Then the bottom supercompiler is used to check whether (e, e') is an improvement lemma and the search for a lemma stops.

Ensuring the termination of the top supercompiler is an exciting problem that requires further investigation. The termination of zero-level supercompilation is achieved by the check for homeomorphic embedding and generalization [23,22,13]. However, the main idea of higher-level supercompilation (in the version presented in Fig. 6) consists in avoiding generalization. When an embedding is detected, the use of an improvement lemma enables the supercompiler to avoid generalization and continue to build the process tree. And this, potentially, may lead to non-termination.

From the practical point of view, though, the non-termination can be avoided by imposing some restrictions on the use of lemmas. A simple and straightforward solution is the following.

Suppose, the check for embedding finds that there an upper node a is embedded in a lower node b. Then b is replaced with b' with the aid of an improvement lemma, and supercompilation continues without generalization. But this fact is recorded in the node a, so that next time when a gets embedded in another node, no lemma will be used, and a will be generalized as in the case of zero-level supercompilation.

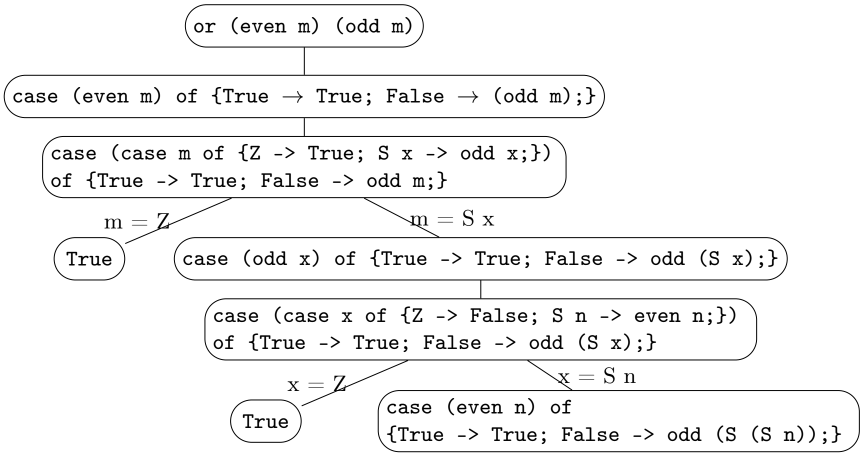
```
data Bool = True | False;
data Nat = Z | S Nat;

or (even m) (odd m) where

even = λx → case x of { Z → True; S x1 → odd x1;};
odd = λx → case x of { Z → False; S x1 → even x1;};

or = λx y → case x of { True → True; False → y;};
```

Fig. 10. or (even m) (odd m): the source program



or (even m) (odd m)
case (even m) of {True → True; False → (odd m);}
case (case m of {Z -> True; S x -> odd x;}) of {True -> True; False -> odd m;}
m = Z
m = S x
True
case (odd x) of {True -> True; False -> odd (S x);}
case (case x of {Z -> False; S n -> even n;}) of {True -> True; False -> odd (S x);}
x = Z
x = S n
True
case (even n) of {True -> True; False -> odd (S (S n));}

Fig. 11. *or* (*even m*) (*odd m*): the whistle blows

To some extent this idea is alike to "cross-fertilization" used by Boyer and Moore [2] in their theorem prover. They argue that the induction hypothesis should be used just once, and then thrown away. (And then comes the turn of generalization.)

5 Examples

5.1 Supercompiling a non-linear expression

Let us try to supercompile the program shown in Fig. 10. We know in advance that the expression `or (even m) (odd m)` can never return `False`, since a natural number m is either even or odd. But this cannot be readily seen from the program's text! After a few driving steps, we get the process tree shown in Fig. 11. At this point an embedding (by coupling) is detected:

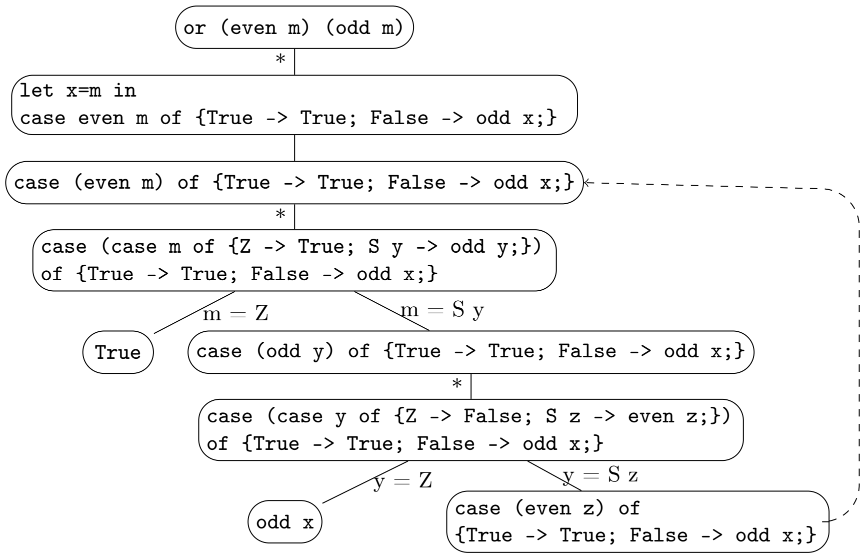

or (even m) (odd m)
*
let x=m in
case even m of {True -> True; False -> odd x;}
case (even m) of {True -> True; False -> odd x;}
*
case (case m of {Z -> True; S y -> odd y;})
of {True -> True; False -> odd x;}
m = Z
m = S y
True
case (odd y) of {True -> True; False -> odd x;}
*
case (case y of {Z -> False; S z -> even z;})
of {True -> True; False -> odd x;}
y = Z
y = S z
odd x
case (even z) of
{True -> True; False -> odd x;}

Fig. 12. *or* (*even m*) (*odd m*): after generalization

```
*(letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          letrec g = *(λw→
            case w of {
              Z → False;
              S t → *(case t of {Z → True; S z → g z;});})
           in g m;
        S x → f x;
      });
  })
in f m)
```

Fig. 13. or (even m) (odd m): the result of “zero-level” supercompilation

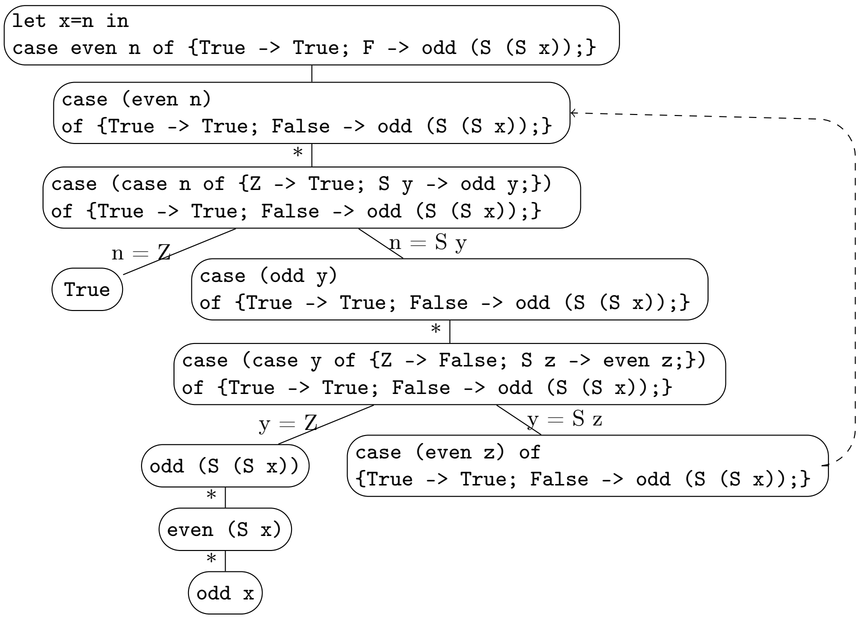

let x=n in
case even n of {True -> True; F -> odd (S (S x));}
case (even n)
of {True -> True; False -> odd (S (S x));}
*
case (case n of {Z -> True; S y -> odd y;})
of {True -> True; False -> odd (S (S x));}
n = Z
n = S y
True
case (odd y)
of {True -> True; False -> odd (S (S x));}
*
case (case y of {Z -> False; S z -> even z;})
of {True -> True; False -> odd (S (S x));}
y = Z
y = S z
odd (S (S x))
case (even z) of
{True -> True; False -> odd (S (S x));}
*
even (S x)
*
odd x

Fig. 14. *case even n of* $\{True \rightarrow True; False \rightarrow odd\ (S\ (S\ n)); \}$: annotated process tree

```
case (even m) of {True → True; False → (odd m);}
    ⊴c case (even n) of {True → True; False → (odd (S (S n)));}
```

But the second expression is not an instance of the first one, for which reason a folding cannot be performed. So the zero-level supercompiler HOSC would perform a generalization by replacing the first expression with the let-expression:

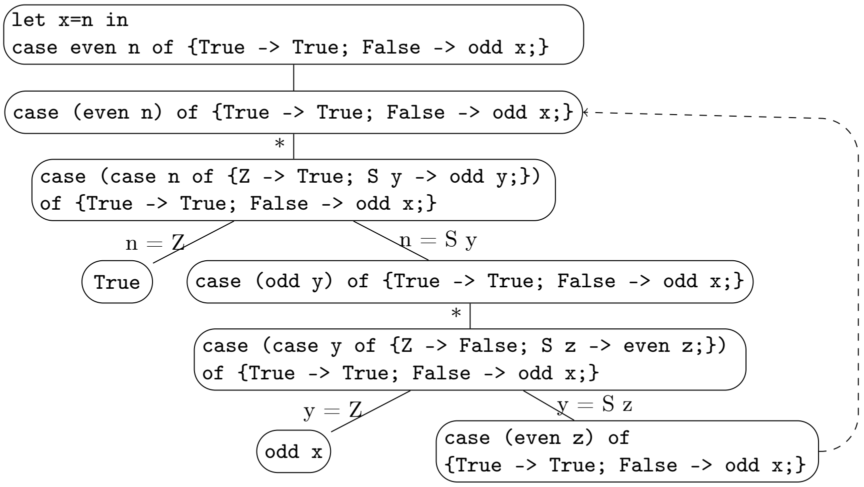
```
let x = m in case (even m) of {True → True; False → (odd x);}
```

Then it would continue by transforming the body of the let-expression, instead of the original expression, thereby "forgetting" that x and m have the same value. This loss of information would result in the residual program in Fig. 13, containing `False`, despite the fact that `False` can never be returned by the program.

The higher-level version of HOSC, however, tries to find and apply an improvement lemma. The first lemma it finds has the size 5:

```
case (even n) of {True → True; False → (odd (S (S n)));}
    ≅ or (even n) (odd n)
```

But this lemma is not an improvement one, and the higher-level supercompiler rejects it by supercompiling its left and right sides with the bottom supercompiler to produce annotated expressions in Fig. 16 and Fig. 13, respectively.



let x=n in
case even n of {True -> True; False -> odd x;}
case (even n) of {True -> True; False -> odd x;}
*
case (case n of {Z -> True; S y -> odd y;})
of {True -> True; False -> odd x;}
n = Z
n = S y
True
case (odd y) of {True -> True; False -> odd x;}
*
case (case y of {Z -> False; S z -> even z;})
of {True -> True; False -> odd x;}
y = Z
y = S z
odd x
case (even z) of
{True -> True; False -> odd x;}

Fig. 15. *case even n of* $\{True \rightarrow True; False \rightarrow odd\ n;\}$: annotated process tree

```
letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          **(letrec g = *(λw→
            case w of {
              Z → False;
              S t → *(case t of {Z → True; S z → g z;});})
           in g n);
        S x → f x;
      };)
  })
in f n
```

Fig. 16. *case even n of* $\{True \rightarrow True; False \rightarrow odd\ (S\ (S\ n));\}$: annotated residual program

```
letrec f=*(λv→
  case v of {
    Z → True;
    S p →
      *(case p of {
        Z →
          (letrec g = *(λw→
            case w of {
              Z → False;
              S t → *(case t of {Z → True; S z → g z;});})*
           in g n);
        S x → f x;
      };)
  })
in f n
```

Fig. 17. *case even n of* $\{True \rightarrow True; False \rightarrow$ *odd n*$;\}$: annotated residual program

```
letrec f=λw→
  case w of {
    Z → True;
    S x → case x of { Z → True; S z → f z;};
  }
in f m
```

Fig. 18. or (even m) (odd m): the result of higher-level supercompilation

However, there exist two improvement lemmas of size 6:

```
case (even n) of {True → True; False → odd (S (S n));}
    ⊳̰ case (even n) of {True → True; False → odd n;}

case (even n) of {True → True; False → odd (S (S n));}
    ⊳̰ case (odd n) of {True → odd n; False → True;}
```

The higher-level HOSC finds and applies the first one, thereby avoiding generalization and producing the program in Fig. 18. Now `False` does not appear in the program!

5.2 Accumulating parameter: using an improvement lemma

Let us reconsider the program with an accumulating parameter shown in Fig. 2. If we try to supercompile it, the whistle blows for the following expressions:

```
case double x Z of {Z → True; S y → odd y);}
    ⊴c case double n (S (S Z)) of {Z → True; S m → odd m;}
```

There are two improvement lemmas (of minimal size):

```
case double n (S (S Z)) of {Z → True; S m → odd m;}
       ⊵ case double n (S Z) of {Z → True; S m → even m;}
case double n (S (S Z)) of {Z → True; S m → odd m;}
       ⊵ case double n (S Z) of {Z → False; S m → even m;}
```

The higher-level HOSC finds and applies the first one and, after some driving, the whistle blows for the second time:

```
case double n (S Z) of {Z → True; S m → even m;}
    ⊴_c case double p (S (S (S Z))) of {Z → True; S m → even m;}
```

Again, there are two improvement lemmas (of minimal size):

```
case double p (S (S (S Z))) of {Z → True; S m → even m;}
        ⊵ case double p (S Z) of {Z → True; S m → even m;}
case double p (S (S (S Z))) of {Z → True; S m → even m;}
        ⊵ case double p (S Z) of {Z → False; S m → even m;}
```

The application of the first lemma enables a fold to be performed without generalization, so that the higher-level HOSC produces the program in Fig. 19.

```
case x of {
  Z → True;
  S y1 →
    letrec f=λt2→
      case t2 of {Z → True; S u2 → f u2;}
    in f y1;
}
```

Fig. 19. even (double x Z): the result of higher-level supercompilation

6 Discussion and conclusion

The main idea of higher-level supercompilation is based on the principle of *metasystem transition* [27,28].

Another approach to increasing the power of supercompilation based on metasystem transition is *distillation* [8,10,9].

In many cases distillation and higher-level supercompilation produce similar results, but, seemingly, an advantage of higher-level supercompilation is its conceptual simplicity and modularity: it can be implemented by a slight modification of a "classic" supercompiler, adding a (conceptually) trivial lemma generator, and making several copies of the same supercompiler to interact. Since the lemma generator uses the supercompiler as a "black box", its design does not depend on the subtle details of the supercompilation process.

Our current implementation of higher-level supercompilation is a proof-of-concept one, is rather “naive”, and can be improved in a variety of ways.

First, the higher-level supercompilation algorithm shown in Fig. 6 tries to apply a lemma only to the whole embedding (lower) expression. But lemmas could be applied in a more refined way.

– An (instance of an) improvement lemma could be applied to a subexpression of the embedding expression.
– To avoid generalization, an (instance of an) improvement lemma could also be applied to a (sub)expression of the embedded (upper) expression.

Second, the search for lemmas is implemented in a straightforward way: no attempt is made to take into account the structure of the embedding (lower) and embedded (upper) expressions. However, there are a few techniques developed in the field of inductive theorem proving (like difference matching [1], rippling [3] and divergence critic [29]) that could be used in implementing a more refined lemma generator.

Higher-level supercompilation does not depend on minor implementation details of the supercompiler it is based upon. However, some properties of the supercompiler do matter. First of all, the check whether a pair of expressions forms an improvement lemma [20,21] relies on the supercompiler preserving termination properties of programs [15,13]. This requirement is not met by all supercompilers. For example, the supercompiler SCP4 [16] dealing with programs in Refal, a strict first-order functional language, may extend the domain of a transformed function, for which reason the equivalence of expressions can be proven by supercompilation only for total expressions operating on finite data structures [17].

During supercompilation, termination properties are easier to preserve for a lazy functional language, than for a strict one. Nevertheless, Jonsson [11] succeeded in developing a termination-preserving supercompilation technique for a higher-order call-by-value language. Therefore, higher-level supercompilation is certainly applicable to higher-order strict languages.

Since any residual program produced by HOSC is a self-contained expression, the check for equality and improvement amounts to a trivial comparison of expressions. In the case of a supercompiler like Supero [19,18], residual programs may have less trivial structure, therefore, comparing them for syntactic isomorphism may be more intricate that in case of HOSC.

In principle, higher-level supercompilation should be implementable also on the basis of a supercompiler for an imperative or object oriented language, such as the Java supercompiler by Klimov [12], but there remains a number of technical problems to be investigated.

Acknowledgements We would like to express our gratitude for Andrei V. Klimov and Yuri A. Klimov for their comments and fruitful discussions. Special thanks are due to Sergei M. Abramov for his encouragement and support for the project.

References

1. D. A. Basin and T. Walsh. Difference matching. In *CADE-11: Proceedings of the 11th International Conference on Automated Deduction*, pages 295–309, London, UK, 1992. Springer-Verlag.
2. R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *J. ACM*, 22(1):129–144, 1975.
3. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: a heuristic for guiding inductive proofs. *Artif. Intell.*, 62(2):185–253, 1993.
4. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM (JACM)*, 24(1):44–67, 1977.
5. A. P. Ershov. On the essence of compilation. In E. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.
6. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
7. R. Glück. Is there a fourth Futamura projection? In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 51–60, New York, NY, USA, 2009. ACM.
8. G. W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70. ACM Press New York, NY, USA, 2007.
9. G. W. Hamilton. Extracting the essence of distillation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 151–164, 2010.
10. G.W. Hamilton and M.H. Kabir. Constructing programs from metasystem transition proofs. In *Proceedings of the First International Workshop on Metacomputation in Russia*, pages 9–26, Pereslavl-Zalessky, Russia, July 2008. Ailamazyan University of Pereslavl.
11. P. A. Jonsson. Positive supercompilation for a higher-order call-by-value language. Master's thesis, Luleå University of Technology, 2008.
12. And. V. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 185–192, 2010.
13. I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, 2009.
14. I. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. Preprint 21, Keldysh Institute of Applied Mathematics, 2010.
15. I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.
16. A. P. Lisitsa and A. P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Program. Comput. Softw.*, 33(1):14–23, 2007.
17. A.P. Lisitsa and M. Webster. Supercompilation for equivalence testing in metamorphic computer viruses detection. In *Proceedings of the First International Workshop on Metacomputation in Russia*, pages 113–118. Ailamazyan University of Pereslavl, 2008.
18. N. Mitchell. *Transformation and Analysis of Functional Programs.* PhD thesis, University of York, 2008.
19. N. Mitchell and C. Runciman. A supercompiler for core Haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *LNCS*, pages 147–164, Berlin, Heidelberg, 2008. Springer-Verlag.

20. D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1-2):193–233, 1996.
21. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.
22. M. H. Sørensen. Convergence of program transformers in the metric space of trees. *Sci. Comput. Program.*, 37(1-3):163–205, 2000.
23. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Logic Programming: The 1995 International Symposium*, pages 465–479, 1995.
24. M. H. Sørensen, R. Glück, and N. D. Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 485–500, London, UK, 1994. Springer-Verlag.
25. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
26. V. F. Turchin. *The phenomen of science. A cybernetic approach to human evolution.* Columbia University Press, New York, 1977.
27. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.
28. V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In *Partial Evaluation*, volume 1110 of *LNCS*, pages 481–509. Springer, 1996.
29. T. Walsh. A divergence critic for inductive proof. *J. Artif. Int. Res.*, 4(1):209–235, 1996.

A Simple Supercompiler Formally Verified in Coq

Dimitur Krustev

IGE+XAO Balkan, Bulgaria
dkrustev@ige-xao.com

Abstract. We study an approach for verifying the correctness of a simplified supercompiler in Coq. While existing supercompilers are not very big in size, they combine many different program transformations in intricate ways, so checking the correctness of their implementation poses challenges. The presented method relies on two important technical features to achieve a compact and modular formalization: first, a very limited object language; second, decomposing the supercompilation process into many sub-transformations, whose correctness can be checked independently. In particular, we give separate correctness proofs for two key parts of driving – normalization and positive information propagation – in the context of a non-Turing-complete expression sub-language. Though our supercompiler is currently limited, its formal correctness proof can give guidance for verifying more realistic implementations.

1 Introduction

Supercompilation [21,18] typically combines a small set of local source transformations with (function) unfolding/folding and generalization in an intricate way. Some general methods have been developed for verifying its correctness – both in the sense of semantics preservation [15] and concerning termination on all inputs [16]. Nonetheless, in view of recent advances in tools for formal computer-verified reasoning, it appears interesting to develop techniques for formal proofs of supercompiler correctness. If one is to pursue such a task, there are two options. The first is to follow as closely as possible the definition of a working supercompiler and to develop a proof of its correctness that can be verified automatically by an existing proof checker. This approach has already been shown to work on systems as complex as compilers for real-world programming languages [12] and operating system kernels [10], so it should be feasible, but probably a lot of work. Alternatively, we can start with simplifying the task as much as possible, so that formal proofs become much easier. This approach is useful even if one is not interested in computer-checked proofs, as it can shed new light on the interaction of the various ingredients of supercompilation.

To simplify the definition of supercompilation, we combine two methods. Firstly, we use a toy programming language operating on an equally simple value domain. Then we take a step forward by isolating as much as possible the different ingredients of supercompilation, with independent proofs of correctness for each one, which are then combined in a modular way in a proof of correctness for the whole supercompiler. Traditionally supercompilation is presented as the combination of several processes – driving, "whistle", folding, generalization (see [18] for a good introduction of positive supercompilation). In the actual definition of the complete supercompiler algorithm this conceptual decomposition is typically blurred, and it appears non-trivial to exploit it directly in verification. We achieve a more significant separation of phases, with an individual proof of correctness for each of them:

- Driving (minus unfolding) is defined in the context of a non-Turing-complete language of simple expressions (whose denotational semantics is defined in Sect. 2), and is decomposed into two separate transformations:
  - Normalization (equivalent to simple deforestation [22] minus unfolding, Sect. 2.1);
  - Positive information propagation (Sect. 2.3). An interesting technical detail here is, that we use a variable-free language and a simple form of explicit substitutions [1] for propagating information (Sect. 2.2);
- We use a small imperative language, which embeds the expression sublanguage discussed above and whose programs contain a single while loop. We define its semantics in a big-step operational style (Sect. 3). Unfolding and folding are correspondingly replaced by a basic form of loop unrolling. The proof of correctness is cleanly split in two – correctness of one-step unrolling (Sect. 3.1), and correctness of repeated unrollings (Sect. 3.4);
- The treatment of the "whistle" brings no novelties beside the fact that the proof of termination is completely separated from the proof of (partial) correctness (Sect. 3.2). This is one of the few places where we "cheat" a bit – we take Kruskal's tree theorem as an assumption, as its formal proof is a big topic of its own;
- We ignore generalization for the time being, as it does not appear essential for our current definition of loop unrolling.

While the resulting supercompiler is too limited to be practically useful, it can still achieve interesting results on select small examples (Sect. 3.3). To give a glimpse of the separation of concerns achieved, here is a small example of a normalization (*normConv*) of a simple expression, contrasted with normalization plus positive information propagation (*norm*):

```
Eval compute in (let e := (IfNil Hd (Nil # Nil) (IfNil Hd Nil Tl))
  in (ntrm2trm (normConv e), ntrm2trm (norm e))).
=(IfNil Hd (Nil # Nil) (IfNil Hd Nil Tl), IfNil Hd (Nil # Nil) Tl)
```

Notice the removal of the redundant test in the second expression.

1.1 Notation

The text of this article is produced from a literate Coq script using the coqdoc tool [19]. All data type and function definitions, as well as the statements of all lemmas/theorems, are given directly in Coq syntax. We use almost none of the more advanced or specific features of this proof assistant, so while our readers should be familiar with functional programming and first-order logic, they do not need prior experience with Coq. Coq contains a total functional programming sublanguage, similar in many respects to languages like Haskell and OCaml (modulo totality requirements). It permits well-founded inductive[1] data type definitions (keyword `Inductive` ...), non-recursive global definitions (`Definition`), structurally recursive global (`Fixpoint`) and local (*fix*) definitions, pattern matching (`match` ... `with` | ... ⇒ ... | ... `end`), lambda functions (`fun` ... ⇒ ...). Coq also embeds (a form of) intuitionistic logic[2], with the usual logical quantifiers and connectives ($\forall$, $\exists$, $\to$, *False*, *True*, $\neg$, $\wedge$, $\vee$, $\leftrightarrow$, $=$). Type information is specified as (x: T) for an object x having a type T. Computable types usually have sort `Set`, while logical propositions live in sort `Prop`. There is a computable type *bool*: `Set` with a number of operations on it, which should not be confused with the non-executable logical propositions and connectives living in `Prop`. We use other standard library types – natural numbers (*nat* with constructors O and S, and standard arithmetic operations), and lists (*list X* with constructors *nil* and ::, and standard list operations like *length* and ++ (append)).

We believe, that the formulation of definitions and lemma statements is more valuable in understanding this work than the detailed proofs themselves. Proofs can be quite lengthy and are usually expressed using a special tactic language – which can be difficult to follow outside of Coq. The complete proofs can always be checked inside the original Coq source. Therefore, we have omitted them here and only give brief informal hints for some of the more complicated ones. Most of the lengthier proofs are broken up into a series of lemmas, each one building on the previous ones, and culminating in a final theorem with a typically trivial proof. We do give the statements of all such lemmas, not only the main theorems. Furthermore, proofs of individual lemmas usually proceed straightforwardly by induction, and can be automated to a great extent using suitable heuristics for automatic proof search [4].

2 Simple Expression Language

We start with an extremely simple domain of values - binary trees (or, equivalently, Lisp-like S-expressions) with a single atom, *VNil*. As some of our built-in functions will be partial, we also include a second dedicated atom, *VBottom*, used to make all built-in functions total.

```
Inductive Val: Set :=
  | VNil: Val | VCons: Val → Val → Val | VBottom: Val.
```

[1] Coinductive definitions are also possible, but we do not use them here
[2] Classical reasoning is also possible in Coq, but not required here

The use of an untyped language is motivated by a hope for greater overall simplicity, although a move to a typed setting would certainly bring some benefits. Notice that the domain is lifted, as *VCons* is not strict w.r.t. *VBottom*: *VCons VBottom VBottom* ≠ *VBottom*.

The expressions of our simple language are built of primitives for constructing (*Nil*, *Cons*) and deconstructing (*Sel*) binary trees, function composition and identity (*Cmp*, *Id*), and conditional expressions testing for null values. It is convenient to have also a bottom-building primitive, *Bottom*, but there is no way of testing for bottom:

```
Inductive Selector: Set := | HD | TL.

Inductive Trm: Set :=
 | Nil: Trm | Cons: Trm → Trm → Trm | Sel: Selector → Trm
 | Id: Trm | Cmp: Trm → Trm → Trm
 | IfNil: Trm → Trm → Trm → Trm | Bottom.
```

We can use Coq's `Notation` mechanism to add a small amount of syntax sugar (note that lower levels correspond to higher precedence).

```
Infix "$" := Cmp (at level 60, right associativity).
Notation Hd := (Sel HD). Notation Tl := (Sel TL).
Infix "#" := Cons (at level 62, right associativity).
```

A few things are notable in the choice of language. It is variable-free, all expressions denoting functions of type *Val* → *Val*. It is the presence of pair constructor and selectors, as well as function composition, as primitives, that gives this language the ability to encode substitutions and to do away with variables. As the language is not Turing-complete, it is straightforward to give its semantics as a total function, *evalT*:

```
Definition evalSel (sel: Selector) (v: Val) : Val :=
 match v with
 | VCons v1 v2 ⇒ match sel with | HD ⇒ v1 | TL ⇒ v2 end
 | _ ⇒ VBottom
 end.

Definition evalSels (sels: list Selector) (v: Val) : Val :=
 fold_left (fun v sel ⇒ evalSel sel v) sels v.

Fixpoint evalT (t: Trm) (v: Val) {struct t} : Val :=
 match t with
 | Nil ⇒ VNil
 | Cons t1 t2 ⇒ VCons (evalT t1 v) (evalT t2 v)
 | Sel sel ⇒ evalSel sel v
 | Id ⇒ v
 | Cmp t1 t2 ⇒ evalT t1 (evalT t2 v)
 | IfNil t1 t2 t3 ⇒ match evalT t1 v with
```

```
    | VNil ⇒ evalT t2 v | VCons _ _ ⇒ evalT t3 v | VBottom ⇒ VBottom
    end
  | Bottom ⇒ VBottom
  end.
```

2.1 Normalization of Simple Expressions

The first step in our series of transformations will be to perform some standard normalizing simplifications to expressions. As the resulting expressions will always have a specific shape, we can define a special type for normal-form expressions:

```
Inductive NTrm: Set :=
  | NNil: NTrm | NCons: NTrm → NTrm → NTrm
  | NSelCmp: list Selector → NTrm
  | NIfNil: list Selector → NTrm → NTrm → NTrm
  | NBottom: NTrm.
```

The important difference is, that in normal forms function composition can only be applied to pair selectors, and that tests in conditional expressions are only of this special form of selector compositions. Notice that the selectors appear in reverse order in lists, and such lists of selectors can be directly interpreted as positions in the binary trees of values. Of course, normal forms can be injected back into the set of full-blown expressions:

```
Definition sels2trm (sels: list Selector): Trm := fold_left (fun t sel ⇒
  match t with | Id ⇒ Sel sel | _ ⇒ Cmp (Sel sel) t end) sels Id.
```

```
Fixpoint ntrm2trm (nt: NTrm) {struct nt} :Trm :=
  match nt with
  | NNil ⇒ Nil
  | NCons nt1 nt2 ⇒ Cons (ntrm2trm nt1) (ntrm2trm nt2)
  | NSelCmp sels ⇒ sels2trm sels
  | NIfNil sels nt1 nt2 ⇒ IfNil (sels2trm sels) (ntrm2trm nt1) (ntrm2trm nt2)
  | NBottom ⇒ Bottom
  end.
```

Using this injection, we can define a specialized evaluation function for normal terms by re-using the main evaluation function.

```
Definition evalNT (nt: NTrm) (v: Val) : Val := evalT (ntrm2trm nt) v.
```

Next we establish some basic properties involving *evalSels*, that will be useful in subsequent proofs.

```
Lemma evalT_sels2trm: ∀ sels: list Selector, ∀ v: Val,
  evalT (sels2trm sels) v = evalSels sels v.
```

```
Lemma evalSelsVBottom: ∀ sels: list Selector, evalSels sels VBottom = VBottom.
Lemma evalSelsAppend: ∀ sels1 sels2: list Selector, ∀ v: Val,
  evalSels (sels1 ++ sels2) v = evalSels sels2 (evalSels sels1 v).
```

The main normalization function, *normConv*, uses a number of auxiliary operations on normal-form expressions, dealing mostly with special cases of function composition. We list along the way some lemmas establishing characteristic properties of the functions defined. The simplest cases cover composition of a selector or a list of selectors with an expression.

```
Fixpoint normSelNCmp (sel: Selector) (nt: NTrm) {struct nt}: NTrm :=
  match nt with
  | NNil ⇒ NBottom
  | NCons nt1 nt2 ⇒ match sel with | HD ⇒ nt1 | TL ⇒ nt2 end
  | NSelCmp sels ⇒ NSelCmp (sels ++ (sel::nil))
  | NIfNil sels nt1 nt2 ⇒ NIfNil sels
      (normSelNCmp sel nt1) (normSelNCmp sel nt2)
  | NBottom ⇒ NBottom
  end.

Lemma normSelNCmpPreservesEval: ∀ (sel: Selector) (nt: NTrm) (v: Val),
  evalNT (normSelNCmp sel nt) v = evalSel sel (evalNT nt v).

Definition normSelsNCmp (sels: list Selector) (nt: NTrm) : NTrm :=
  fold_left (fun nt sel ⇒ normSelNCmp sel nt) sels nt.

Lemma normSelsNCmpPreservesEvalT: ∀ sels: list Selector, ∀ nt: NTrm,
  ∀ v: Val, evalT (ntrm2trm (normSelsNCmp sels nt)) v
  = evalSels sels (evalT (ntrm2trm nt) v).

Lemma normSelsNCmpPreservesEval: ∀ sels: list Selector, ∀ nt: NTrm, ∀ v: Val,
  evalNT (normSelsNCmp sels nt) v = evalSels sels (evalNT nt v).

Lemma normSelsNCmp_NSelCmp: ∀ (sels1 sels2: list Selector),
  normSelsNCmp sels1 (NSelCmp sels2) = NSelCmp (sels2 ++ sels1).
```

We also consider composition of selectors to the right of a normal-form expression *nt*.

```
Fixpoint normNCmpSels (nt: NTrm) (sels: list Selector) {struct nt}
  : NTrm := match nt with
  | NNil ⇒ NNil
  | NCons nt1 nt2 ⇒
      NCons (normNCmpSels nt1 sels) (normNCmpSels nt2 sels)
  | NSelCmp sels2 ⇒ NSelCmp (sels ++ sels2)
  | NIfNil sels2 nt1 nt2 ⇒ NIfNil (sels ++ sels2)
      (normNCmpSels nt1 sels) (normNCmpSels nt2 sels)
  | NBottom ⇒ NBottom
  end.
```

```
Lemma normNCmpSelsPreservesEval: ∀ sels: list Selector, ∀ nt: NTrm, ∀ v: Val,
  evalNT (normNCmpSels nt sels) v = evalNT nt (evalSels sels v).
```

```
Lemma normNCmpSels_app: ∀ (sels1 sels2: list Selector) (nt: NTrm),
  normNCmpSels nt (sels1 ++ sels2)
  = normNCmpSels (normNCmpSels nt sels2) sels1.
```

Next, we deal with building conditional expressions in normal form. If the normal form of the condition is a value-constructing primitive, we can statically reduce the whole if-expression. The other interesting case is when the condition is itself another if-expression – in this case we switch the order of the tests and duplicate the original outer *NIfNil* inside the branches of the new outer *NIfNil*.

```
Fixpoint normNIf (nt1 nt2 nt3: NTrm) {struct nt1} : NTrm :=
  match nt1 with
  | NNil ⇒ nt2
  | NCons _ _ ⇒ nt3
  | NSelCmp sels ⇒ NIfNil sels nt2 nt3
  | NIfNil sels nt1_1 nt1_2 ⇒ NIfNil sels
      (normNIf nt1_1 nt2 nt3) (normNIf nt1_2 nt2 nt3)
  | NBottom ⇒ NBottom
  end.
```

```
Lemma normNIfPreservesEvalT: ∀ nt1 nt2 nt3: NTrm, ∀ v: Val,
  evalT (ntrm2trm (normNIf nt1 nt2 nt3)) v
  = match evalT (ntrm2trm nt1) v with
    | VNil ⇒ evalT (ntrm2trm nt2) v
    | VCons _ _ ⇒ evalT (ntrm2trm nt3) v
    | VBottom ⇒ VBottom
    end.
```

```
Lemma normNIfPreservesEval: ∀ nt1 nt2 nt3: NTrm, ∀ v: Val,
  evalNT (normNIf nt1 nt2 nt3) v = match evalNT nt1 v with
    | VNil ⇒ evalNT nt2 v
    | VCons _ _ ⇒ evalNT nt3 v
    | VBottom ⇒ VBottom
    end.
```

The sequence of operations on normal-form expressions culminates in a function *normNCmp*, which permits to form the composition of two normal-form expressions without having function composition as primitive. The most interesting cases here involve the composition of *NIfNil* and *NSelCmp*/*NIfNil*:

```
Definition normNCmp : NTrm → NTrm → NTrm :=
  fix normNCmp_nt1 (nt1: NTrm): NTrm → NTrm :=
  fix normNCmp_nt2 (nt2: NTrm): NTrm :=
  match nt1 with
  | NNil ⇒ NNil
  | NCons nt1_1 nt1_2 ⇒
      NCons (normNCmp_nt1 nt1_1 nt2) (normNCmp_nt1 nt1_2 nt2)
  | NSelCmp sels ⇒ normSelsNCmp sels nt2
  | NIfNil sels nt1_1 nt1_2 ⇒ match nt2 with
    | NSelCmp sels2 ⇒ NIfNil (sels2 ++ sels)
        (normNCmpSels nt1_1 sels2) (normNCmpSels nt1_2 sels2)
    | NIfNil sels2 nt2_1 nt2_2 ⇒ NIfNil sels2
        (normNCmp_nt2 nt2_1) (normNCmp_nt2 nt2_2)
    | _ ⇒ normNIf (normSelsNCmp sels nt2)
        (normNCmp_nt1 nt1_1 nt2) (normNCmp_nt1 nt1_2 nt2)
    end
  | NBottom ⇒ NBottom
  end.
```

We can easily establish that the composition of 2 if-expressions can be replaced by pushing the first if-expression inside the branches of the second:

```
Lemma normNCmpIfIf: ∀ sels1 sels2: list Selector,
  ∀ nt1_1 nt1_2 nt2_1 nt2_2: NTrm, let nt1 := NIfNil sels1 nt1_1 nt1_2 in
  normNCmp nt1 (NIfNil sels2 nt2_1 nt2_2)
  = NIfNil sels2 (normNCmp nt1 nt2_1) (normNCmp nt1 nt2_2).
```

We also establish that *normNCmp* satisfies the defining property of function composition; this is the key lemma on which correctness of normalization relies:

```
Lemma normNCmpPreservesEval: ∀ nt1 nt2: NTrm, ∀ v: Val,
  evalNT (normNCmp nt1 nt2) v = evalNT nt1 (evalNT nt2 v).
```

The last lemma is a bit tricky to prove: as *normNCmp* is defined using nested lexicographic recursion, we must use nested induction in the proof and apply rewritings using the previously proved lemmas.

Finally, the stage is set for the conversion of arbitrary expressions into normal form:

```
Fixpoint normConv (t: Trm) {struct t} :NTrm :=
  match t with
  | Nil ⇒ NNil
  | Cons t1 t2 ⇒ NCons (normConv t1) (normConv t2)
  | Sel sel ⇒ NSelCmp (sel::nil)
  | Id ⇒ NSelCmp nil
  | Cmp t1 t2 ⇒ normNCmp (normConv t1) (normConv t2)
  | IfNil t1 t2 t3 ⇒ normNIf (normConv t1) (normConv t2) (normConv t3)
  | Bottom ⇒ NBottom
  end.
```

With all this in place, the main theorem establishing the correctness of normalization can be proved by straightforward induction on the expression structure:

```
Theorem normConvPreservesEval: ∀ (t: Trm) (v: Val),
  evalNT (normConv t) v = evalT t v.
```

We can see on an example, that *normConv* not only brings expressions into normal form, but also achieves some optimizations like deforestation:

```
Eval compute in (ntrm2trm (normConv ((IfNil Hd ((Tl $ Tl) # (Hd $ Tl))
Tl) $ (Nil # Id)))).
```

```
  = Tl # Hd : Trm
```

As we have seen in the introduction, however, normalization by itself does not eliminate redundant tests.

2.2 Emulating Substitutions

Before we tackle positive information propagation, we need to make a small detour and show how substitutions can be emulated inside our language, giving a simple form of explicit substitutions [1]. Let's first note that we can replace a set of values, denoted by variables, with a list structure built from pairs (e.g. [8]). Variables in this case can be replaced by positions in the list structure, represented by lists of pair selectors. For example, the expression *IfNil x1 x2 x3*, has three free variables. We can pack their values into a list – *x1 # x2 # x3* – and replace their references inside the expression with the corresponding list positions: *IfNil Hd (Hd $ Tl) (Tl $ Tl)*, as the following clearly holds: (*IfNil Hd* (*Hd $ Tl*) (*Tl $ Tl*)) $ (*x1 # x2 # x3*) = *IfNil x1 x2 x3*. We can define an operation, *replaceAt*, which for a given tree position (represented with a list of selectors *pos*) and two normal-form expressions, generates a new expression, which has the result of the second expression pushed at position *pos* in the result of the first expression.

```
Fixpoint replaceAt (pos: list Selector) (t trep: NTrm) {struct pos}: NTrm :=
  match pos with
    | nil ⇒ trep
    | sel::sels ⇒ match sel with
      | HD ⇒ NCons (replaceAt sels (normSelNCmp HD t) trep)
                   (normSelNCmp TL t)
      | TL ⇒ NCons (normSelNCmp HD t)
                   (replaceAt sels (normSelNCmp TL t) trep)
    end
  end.
```

The action of this function is best illustrated with a couple of examples. If we have 2 values packed in a pair as input – say *x1 # x2* – we can fix the value of *x1* to *Nil # Nil* in the following way:

```
Eval compute in (ntrm2trm (replaceAt (HD::nil) (normConv Id) (normConv
(Nil # Nil)))).
```

```
= (Nil # Nil) # Tl : Trm
```

We can substitute not only constant values but also arbitrary expressions with *replaceAt* and *normNCmp*. If we consider again the expression *IfNil x1 x2 x3* with the given encoding of variables (*x1 # x2 # x3*), we can substitute *Tl $ Hd $ Tl # Hd $ Hd $ Tl* for *x2* thusly:

```
Eval compute in (let nt1 := normConv (IfNil Hd (Hd $ Tl) (Tl $ Tl)) in
  let nt2 := normConv (Tl $ Hd $ Tl # Hd $ Hd $ Tl) in
  ntrm2trm (normNCmp nt1 (replaceAt (TL::HD::nil) (normConv Id) nt2))).

  = IfNil Hd (Tl $ Hd $ Tl # Hd $ Hd $ Tl) (Tl $ Tl) : Trm
```

We now establish some properties of *replaceAt* that will prove useful later.

```
Lemma replaceAt_id: ∀ sels: list Selector, ∀ t trep: NTrm,
  normSelsNCmp sels (replaceAt sels t trep) = trep.

Lemma replaceAt_app: ∀ (sels1 sels2: list Selector) (nt ntrepl: NTrm),
  replaceAt (sels1 ++ sels2) nt ntrepl
  = replaceAt sels1 nt (replaceAt sels2 (normSelsNCmp sels1 nt) ntrepl).
```

For the next property, we need to compute the common prefix and the different suffixes of 2 lists. We shall need also to compute equivalence of selectors.

```
Definition Sel_eq_dec (sel1 sel2: Selector) : {sel1 = sel2} + {sel1 ≠ sel2}.
  decide equality.
Defined.
```

This is just a nice trick to let Coq deduce the equality predicate for us. The type $\{sel1 = sel2\} + \{sel1 \neq sel2\}$ is a sum type than not only gives the outcome of the test, but also contains a proof of the corresponding equality/inequality. For simplicity, we can cast this result to a simple *bool*, using the fact that Coq if expressions apply generically to any type with 2 constructors:

```
Definition eqSel sel1 sel2 := if Sel_eq_dec sel1 sel2 then true else false.

Lemma eqSel_reflx: ∀ sel, eqSel sel sel = true.

Fixpoint commonPrefix (X: Set) (eqX: X → X → bool) (l1 l2: list X)
  {struct l1} : (list X) × (list X) × (list X) := match l1, l2 with
  | nil, _ ⇒ (nil, nil, l2)
  | _, nil ⇒ (nil, l1, nil)
  | x::xs, y::ys ⇒ if eqX x y then
      let cp := commonPrefix X eqX xs ys in let pr := fst (fst cp) in
      let l1a := snd (fst cp) in let l2a := snd cp in (x::pr, l1a, l2a)
    else (nil, l1, l2)
  end.
```

```
Lemma commonPrefix_X_XappY: ∀ X: Set, ∀ eqX: X → X → bool,
  (∀ x: X, eqX x x = true) → ∀ xs ys: list X,
  commonPrefix X eqX xs (xs ++ ys) = (xs, nil, ys).
```

```
Lemma normSelsNCmp_ReplaceAt: ∀ (sels1 sels2: list Selector),
  ∀ (nt ntrepl: NTrm), normSelsNCmp sels1 (replaceAt sels2 nt ntrepl) =
  let cp := commonPrefix _ eqSel sels1 sels2 in let csels := fst (fst cp) in
  let usels1 := snd (fst cp) in let usels2 := snd cp in
  normSelsNCmp usels1 (replaceAt usels2 (normSelsNCmp csels nt) ntrepl).
```

```
Lemma replaceAt_NSelCmp: ∀ (sels1 sels2: list Selector) (nt: NTrm),
  replaceAt sels1 (NSelCmp sels2) nt
  = normSelsNCmp sels2 (replaceAt (sels2 ++ sels1) (NSelCmp nil) nt).
```

2.3 Positive Information Propagation

We can use object-level substitution, as implemented by *replaceAt* and *normNCmp*, to propagate information about the test result inside the branches of a conditional expressions. This transformation is one of the key differences that distinguish supercompilation from weaker optimizations like classical partial evaluation and deforestation [5,18]. The definition is greatly simplified by the fact that normal-form tests can only take the form of selector compositions.

```
Definition setNilAt (sels: list Selector): NTrm :=
  replaceAt sels (NSelCmp nil) NNil.
```

```
Definition setConsAt (sels: list Selector) : NTrm :=
  replaceAt sels (NSelCmp nil)
    (NCons (NSelCmp (sels ++ HD::nil)) (NSelCmp (sels ++ TL::nil))).
```

Once we have an expression encoding the substitution of the test result, what remains is to compose it with the corresponding if-branch, as in our case substitution composition is replaced by simple function composition.

```
Fixpoint propagateIfCond (nt: NTrm) {struct nt} : NTrm :=
  match nt with
  | NCons nt1 nt2 ⇒ NCons (propagateIfCond nt1) (propagateIfCond nt2)
  | NIfNil sels nt1 nt2 ⇒
    let nt1a := propagateIfCond nt1 in let nt2a := propagateIfCond nt2 in
    let nt1b := normNCmp nt1a (setNilAt sels) in
    let nt2b := normNCmp nt2a (setConsAt sels) in NIfNil sels nt1b nt2b
  | _ ⇒ nt
  end.
```

Establishing the correctness of *propagateIfCond* is once again decomposed into a sequence of lemmas.

```
Lemma setNilAtPreservesEvalAux: ∀ (sels1 sels2: list Selector),
```

replaceAt sels1 (*NSelCmp sels2*) *NNil*
= *normNCmpSels* (*replaceAt sels1* (*NSelCmp nil*) *NNil*) *sels2*.

`Lemma` *setConsAtPreservesEvalAux*: ∀ (*sels1 sels2*: *list Selector*),
replaceAt sels1 (*NSelCmp sels2*) (*NCons* (*NSelCmp*
(*sels2*++sels1++HD::nil)) (*NSelCmp* (*sels2*++sels1++TL::nil)))
= *normNCmpSels* (*replaceAt sels1* (*NSelCmp nil*) (*NCons*
(*NSelCmp* (*sels1*++HD::nil)) (*NSelCmp* (*sels1*++TL::nil)))) *sels2*.

`Lemma` *setNilAtPreservesEvalAux2*: ∀ (*v*: *Val*), ∀ (*sels1 sels2*: *list Selector*),
evalSels sels1 (*evalNT* (*setNilAt* (*sels1*++sels2)) *v*)
= *evalNT* (*setNilAt sels2*) (*evalSels sels1 v*).

`Lemma` *setConsAtPreservesEvalAux2*: ∀ (*v*: *Val*), ∀ (*sels1 sels2*: *list Selector*),
evalSels sels1 (*evalNT* (*setConsAt* (*sels1*++sels2)) *v*)
= *evalNT* (*setConsAt sels2*) (*evalSels sels1 v*).

`Lemma` *setNilAtPreservesEval*: ∀ *sels*: *list Selector*, ∀ *v*: *Val*,
evalSels sels v = *VNil* → *evalNT* (*setNilAt sels*) *v* = *v*.

`Lemma` *setConsAtPreservesEval*: ∀ *sels*: *list Selector*, ∀ *v v1 v2*: *Val*,
evalSels sels v = *VCons v1 v2* → *evalNT* (*setConsAt sels*) *v* = *v*.

`Lemma` *condPropagatorsPreserveEval*: ∀ (*sels*: *list Selector*) (*nt1 nt2*: *NTrm*),
∀ (*v*: *Val*), *evalNT* (*NIfNil sels* (*normNCmp nt1* (*setNilAt sels*))
(*normNCmp nt2* (*setConsAt sels*))) *v* = *evalNT* (*NIfNil sels nt1 nt2*) *v*.

The proofs of these lemmas involve some tricky rewrites, using the established properties of *replaceAt*. Details can be found in the actual Coq sources. The main theorem can now be proved easily by induction, using the last lemma *condPropagatorsPreserveEval*.

`Theorem` *propagateIfCondPreservesEval*: ∀ *nt*: *NTrm*, ∀ *v*: *Val*,
evalNT (*propagateIfCond nt*) *v* = *evalNT nt v*.

We can combine the first two stages – normalization and positive information propagation – into a single function, and trivially establish its correctness.

`Definition` *norm* (*t*: *Trm*) := *propagateIfCond* (*normConv t*).

`Theorem` *normPreservesEval*: ∀ *t v*, *evalNT* (*norm t*) *v* = *evalT t v*.

Recalling the example from the introduction, we can see that *norm* also eliminates redundant tests, besides other reductions:

`Eval compute in` (*ntrm2trm* (*norm* (*IfNil Hd* (*Nil* # *Nil*) (*IfNil Hd Nil Tl*)))).

```
  = IfNil Hd (Nil # Nil) Tl : Trm
```

3 A Turing-complete Imperative Language

While our simple expression language has helped us to successfully study some key aspects of supercompilation, it is obvious that we cannot write many interesting programs in it. Not only it is far from being Turing-complete, but it even lacks full-blown primitive recursion. However, we can build upon this language to obtain a larger, Turing-complete one. For example, we can embed the language of simple expressions inside a small imperative language with assignments and while-loops (called here SWhile):

```
Inductive SWhileStmt: Set :=
  | Assign: Trm → SWhileStmt
  | Seq: SWhileStmt → SWhileStmt → SWhileStmt
  | While: Trm → SWhileStmt → SWhileStmt.
```

As a further simplification, we assume that the language has a single variable, similar to other research languages like I and LOOP [8,7]. This variable is implicitly used in assignments and while tests. As this language is Turing-complete, we cannot specify its evaluator directly as a total Coq function, like we did for the language of simple expressions. We can specify its semantics as a logical relation, which is encoded in Coq as a (dependent) inductive family living in Prop:

```
Inductive SWhileEvalRel: Val → SWhileStmt → Val → Prop :=
  | SWhileEvalAssign: ∀ e v, SWhileEvalRel v (Assign e) (evalT e v)
  | SWhileEvalSeq: ∀ st1 st2 v1 v2 v3,
     SWhileEvalRel v1 st1 v2 → SWhileEvalRel v2 st2 v3 →
     SWhileEvalRel v1 (Seq st1 st2) v3
  | SWhileEvalWhileNil: ∀ cond st v,
     evalT cond v = VNil → SWhileEvalRel v (While cond st) v
  | SWhileEvalWhileBottom: ∀ cond st v,
     evalT cond v = VBottom → SWhileEvalRel v (While cond st) VBottom
  | SWhileEvalWhileCons: ∀ cond st v1 v2 v3 vh vt,
     evalT cond v1 = VCons vh vt → SWhileEvalRel v1 st v2 →
     SWhileEvalRel v2 (While cond st) v3 →
     SWhileEvalRel v1 (While cond st) v3.
```

We can further simplify our task, by considering only programs containing a single while loop. This can be seen as an analog of Kleene's normal form (KNF) from recursion theory, and there are well-known proofs (not repeated here) that limiting ourselves to a single while loop implies no loss of generality [6]. The "Kleene normal form" analog for SWhile programs can be represented as a record of 4 simple expressions:

```
Record KNFProg : Set := MkKNFProg {
  initExp: Trm; condExp: Trm; bodyExp: Trm; finalExp: Trm }.
```

The meaning is obvious by the injection into the full syntax of SWhile programs:

Definition *KNFtoProg knf* :=
  Seq (*Assign* (*initExp knf*))
  (*Seq* (*While* (*condExp knf*) (*Assign* (*bodyExp knf*)))
    (*Assign* (*finalExp knf*))).

We can introduce a bit of syntactic sugar for SWhile constructs (at the expense of a conflict with the Record syntax).

Infix ";" := *Seq* (at *level* 65, *right associativity*).
Notation "'VAR' '←' e" := (*Assign e*) (at *level* 64).
Notation "'WHILE' cond 'DO' body 'DONE'" := (*While cond body*) (at *level* 0).

As a simple example, here is a program that reverses its input (assuming the usual Lisp encoding of lists as binary trees).

Definition *revList_knf* := *MkKNFProg*
  (*Id* # *Nil*) *Hd* (*Tl* $ *Hd* # *Hd* $ *Hd* # *Tl*) *Tl*.
Eval compute in (*KNFtoProg revList_knf*).

```
= VAR  <- Id # Nil;
  WHILE Hd DO VAR  <- Tl $ Hd # Hd $ Hd # Tl DONE;
  VAR  <- Tl : SWhileStmt
```

We see here one important drawback of the simplifications we introduced: our language is very difficult to program in, and very unreadable. To make the meaning of the code clearer, we can rewrite it by hand to a version of SWhile with many variables; in our case 2 suffice – *input* and *output*:

```
  output  <- Nil;
  WHILE input DO
    (input # output) <- (Tl $ input) # (Hd $ input # output) DONE;
```

While the abstract syntax of SWhile permits arbitrary expressions as while-loop conditions, many optimizing transformations that follow are valid only if the condition of the loop is strict, according to the following definition:

Definition *strictTrm* (*t*: *Trm*) := *evalT t VBottom* = *VBottom*.

We can easily see that strictness for our expression language amounts to a simple syntactic check on the normal form of the expression:

Lemma *strictTrm_SyntaxCriterion*: ∀ (*t*: *Trm*), *strictTrm t* ↔
  (match *normConv t* with | *NNil* | *NCons* _ _ ⇒ *false* | _ ⇒ *true* end) = *true*.

So it is obviously reasonable to consider only programs with strict loop conditions as otherwise the loop degenerates to either an infinite or an empty one.

While the relational specification of SWhile semantics is elegant, it is not executable (at least not inside Coq). We can build an approximation to an evaluation function in Coq itself, using a standard trick for modeling partial functions – we add an extra parameter limiting the recursion depth, and the definition of the evaluation function can be done by structural recursion on that new parameter. We do so only for the KNF special case.

```
Fixpoint evalKNFCore (d: nat) (cond e: Trm) (v: Val) {struct d}
  : option Val := match d with
  | O ⇒ None
  | S d' ⇒ match evalT cond v with
    | VNil ⇒ Some v
    | VBottom ⇒ Some VBottom
    | VCons _ _ ⇒ evalKNFCore d' cond e (evalT e v)
    end
  end.
```

```
Definition evalKNF (d: nat) (knf: KNFProg) (v: Val) : option Val :=
  match evalKNFCore d (condExp knf) (bodyExp knf)
   (evalT (initExp knf) v) with
  | None ⇒ None
  | Some v ⇒ Some (evalT (finalExp knf) v)
  end.
```

We can now execute the example program above on some input:

```
Definition listToVal vs := fold_right VCons VNil vs.
Eval vm_compute in (evalKNF 3 revList_knf
  (listToVal (VNil::(VCons VNil VNil)::nil))).

  = Some (VCons (VCons VNil VNil) (VCons VNil VNil)) : option Val
```

In order to verify that the executable interpreter is correct with respect to the relational semantics given above, we first establish, that the evaluation of the loop by *evalKNFCore* respects the semantics, and then we prove the correctness of the main evaluation function – *evalKNF*

```
Lemma evalKNFCore_SWhileEvalRel: ∀ cond e v1 v2,
  SWhileEvalRel v1 (While cond (Assign e)) v2 ↔
  ∃ d: nat, evalKNFCore d cond e v1 = Some v2.
```

```
Theorem evalKNF_SWhileEvalRel: ∀ knf v1 v2,
  SWhileEvalRel v1 (KNFtoProg knf) v2 ↔
  ∃ d: nat, evalKNF d knf v1 = Some v2.
```

3.1 Loop Unrolling

The principal additional optimization that we can perform on loop programs – on the top of the already existing optimizations for the expression sub-language – is loop unrolling. We can study different forms of while-loop unrolling; here we shall limit ourselves to one simple form of unrolling – trying to execute the body of the loop once before entering the loop itself, provided the condition of the loop holds. Of course, we cannot expect spectacular optimizations from this form of unrolling; in the very least, it leaves the loop itself unmodified. It is sufficient, however, to demonstrate the power of positive information propagation in some simple cases. Later in the paper we discuss possibilities for more powerful forms of loop unrolling.

```
Definition unrollToInit knf := let nrm t := ntrm2trm (norm t) in
  let newInit := nrm ((IfNil (condExp knf) Id (bodyExp knf)) $ (initExp knf))
  in MkKNFProg newInit (condExp knf) (bodyExp knf) (finalExp knf).
```

We can verify that unrolling the loop once respects the semantics. It turns easier to use *evalKNFCore* and *evalKNF* as semantics specifications; it is OK as we have already verified that they are faithful to the original specification by a logical relation.

```
Lemma normPreservesEval': ∀ t v, evalT (ntrm2trm (norm t)) v = evalT t v.
```

```
Lemma evalKNFCore_Bottom: ∀ d cond e v, strictTrm cond →
  evalKNFCore d cond e VBottom = Some v → v = VBottom.
```

```
Lemma evalKNFCore_unrollToInit_fw: ∀ d knf v1 v2, strictTrm (condExp knf)
  → evalKNFCore d (condExp knf) (bodyExp knf) v1 = Some v2 →
  ∃ d2: nat, evalKNFCore d2 (condExp knf) (bodyExp knf) (evalT
    (IfNil (condExp knf) Id (bodyExp knf)) v1) = Some v2.
```

```
Lemma evalKNF_unrollToInit_fw: ∀ d knf v1 v2, strictTrm (condExp knf)
  → evalKNF d knf v1 = Some v2 →
  ∃ d2: nat, evalKNF d2 (unrollToInit knf) v1 = Some v2.
```

```
Lemma evalKNFCore_unrollToInit_bw: ∀ d knf v1 v2, strictTrm (condExp knf)
  → evalKNFCore d (condExp knf) (bodyExp knf) (evalT
    (IfNil (condExp knf) Id (bodyExp knf)) v1) = Some v2 →
  ∃ d2: nat, evalKNFCore d2 (condExp knf) (bodyExp knf) v1 = Some v2.
```

```
Lemma evalKNF_unrollToInit_bw: ∀ d knf v1 v2, strictTrm (condExp knf)
  → evalKNF d (unrollToInit knf) v1 = Some v2 →
  ∃ d2: nat, evalKNF d2 knf v1 = Some v2.
```

```
Theorem evalKNF_unrollToInit: ∀ knf v1 v2, strictTrm (condExp knf) →
  ((∃ d: nat, evalKNF d knf v1 = Some v2) ↔
   (∃ d2: nat, evalKNF d2 (unrollToInit knf) v1 = Some v2)).
```

To deal with repeated unrollings and to lay the background for termination verification of the whole supercompiler, we need streams (infinite sequences). A simple function-based definition suffices for our purposes.

```
Definition Stream A := nat → A .
```

We define a couple of basic operations on streams – the well-known *map* and *unfold* from the functional programming repertoire.

```
Definition streamMap A B (f: A → B) (s: Stream A) : Stream B :=
  fun n ⇒ f (s n).
```

```
Definition streamUnfold X (seed: X) (f: X → X) : Stream X :=
  fix streamUnfold' (n: nat) {struct n} : X := match n with
  | O ⇒ seed | S n' ⇒ f (streamUnfold' n') end.
```

3.2 Homeomorphic Embedding for Ensuring Termination

The so-called ”whistle” of our supercompiler uses the now-standard approach of relying on homeomorphic embedding and the Kruskal's tree theorem [17] to ensure termination of the process. To formulate this theorem in its general form, we introduce a type of arbitrary first-order terms. The Coq `Section` mechanism allows to specify only once parameters common for a whole set of definitions – in our case the types for term variables and function symbols, as well as the fact that function symbols have decidable equality. (Variables of first-order terms typically also have decidable equality, but it is not needed in the current development.)

```
Section FOTerms.
```

```
Variable V: Set. Variable F: Set.
Variable F_eq_dec: ∀ f g: F, {f = g} + {f ≠ g}.
```

We adopt a slightly non-standard definition of first-order terms, which is however easier to work with in Coq:

```
Inductive FOTerm : Set :=
  | FOVar: V → FOTerm
  | FOFun0: option F → FOTerm
  | FOFun2: option F → FOTerm → FOTerm → FOTerm.
```

```
Definition optionF_eq_dec (f1 f2: option F): {f1 = f2} + {f1 ≠ f2}.
  decide equality.
Defined.
```

Even with this definition of first-order terms, defining an executable version of homeomorphic embedding in Coq is a little tricky – we need two nested structural recursions, like in the case of *normNCmp*.

```
Definition homemb (t1 t2: FOTerm) : bool :=
  (fix homemb1 (t1: FOTerm): FOTerm → bool :=
  (fix homemb2 (t2: FOTerm): bool :=
  match t1 with
  | FOVar _ ⇒ match t2 with | FOVar _ ⇒ true | _ ⇒ false end
  | FOFun0 f1 ⇒ match t2 with
    | FOFun0 f2 ⇒ if optionF_eq_dec f1 f2 then true else false
    | FOFun2 _ t21 t22 ⇒ orb (homemb2 t21) (homemb2 t22)
    | _ ⇒ false
    end
  | FOFun2 f1 t11 t12 ⇒ match t2 with
    | FOFun2 f2 t21 t22 ⇒ orb (if optionF_eq_dec f1 f2
        then andb (homemb1 t11 t21) (homemb1 t12 t22)
        else false) (orb (homemb2 t21) (homemb2 t22))
    | _ ⇒ false
    end
  end
  )) t1 t2.
```

We can now give a formulation of Kruskal's theorem. It is beyond the scope of the current work to give a formal proof of this result, so we just take it as an assumption.

```
Theorem Kruskal: ∀ s: Stream FOTerm,
  ∃ i: nat, ∃ j: nat, i < j ∧ homemb (s i) (s j) = true.
Admitted.
```

```
End FOTerms.
```

We mark some arguments as implicit so that they are inferred by the Coq typechecker.

```
Implicit Arguments FOVar [V F]. Implicit Arguments FOFun0 [V F].
Implicit Arguments FOFun2 [V F]. Implicit Arguments homemb [V F].
```

To use Kruskal's theorem for online termination, we need a few additional ingredients. Firstly, a function that actually computes (the index of) the first of the two terms in a sequence, that are related by homeomorphic embedding. For simplicity, we limit the search to a finite initial fragment of the sequence and prove separately that there is always such initial fragment that will produce results.

```
Definition isNthEmbeddedBelow V F fn_eq_dec (n m: nat)
  (s: Stream (FOTerm V F)) : bool :=
  existsb (fun i ⇒ homemb fn_eq_dec (s n) (s i)) (seq (S n) (m - n)).
Implicit Arguments isNthEmbeddedBelow [V F].
```

```
Definition firstEmbedded V F fn_eq_dec (n: nat) (s: Stream (FOTerm V F))
: option nat :=
  find (fun i ⇒ isNthEmbeddedBelow fn_eq_dec i n s) (seq 0 n).
Implicit Arguments firstEmbedded [V F].
```

We use list functions from the standard library, like *existsb*, *find*, *seq*, with hopefully obvious meanings. Some of their useful properties are missing from the library, and we have to prove them first:

```
Lemma find_Some: ∀ X (f: X → bool) x xs,
  In x xs → f x = true → ∃ y, find f xs = Some y.
```

```
Lemma In_seq: ∀ n m l, In n (seq m l) ↔ m ≤ n < m + l.
```

With these properties in addition to Kruskal's theorem, we easily establish that *firstEmbedded* is total.

```
Theorem firstEmbedded_total: ∀ V F F_eq_dec (s: Stream (FOTerm V F)),
  ∃ n, ∃ m, firstEmbedded F_eq_dec n s = Some m.
```

Another helper function we need is an injection from simple expressions into first-order terms. We first define an enumeration of the constructors of expressions, together with their decidable equality predicate. Then the definition of the injection is straightforward.

```
Inductive TrmCons: Set := | TCNil | TCCons | TCSelHd
  | TCSelTl | TCId | TCCmp | TCIfNil | TCBottom.
```

```
Definition TrmCons_eq_dec (t1 t2: TrmCons) : {t1 = t2} + {t1 ≠ t2}.
  decide equality.
Defined.
```

```
Fixpoint TrmToFOTerm (e: Trm) : FOTerm Empty_set TrmCons :=
  match e with
  | Nil ⇒ FOFun0 (Some TCNil)
  | Cons e1 e2 ⇒ FOFun2 (Some TCCons)
      (TrmToFOTerm e1) (TrmToFOTerm e2)
  | Sel sel ⇒ if sel then FOFun0 (Some TCSelHd)
        else FOFun0 (Some TCSelTl)
  | Id ⇒ FOFun0 (Some TCId)
  | Cmp e1 e2 ⇒ FOFun2 (Some TCCmp)
      (TrmToFOTerm e1) (TrmToFOTerm e2)
  | IfNil e1 e2 e3 ⇒ FOFun2 (Some TCIfNil) (TrmToFOTerm e1)
      (FOFun2 (Some TCCons) (TrmToFOTerm e2) (TrmToFOTerm e3))
  | Bottom ⇒ FOFun0 (Some TCBottom)
  end.
```

3.3 Simple Supercompiler using Loop Unrolling

Now we can assemble all previously defined components into a finished basic supercompiler. It first builds a stream of iterated unrollings of the program in

KNF. Then it looks for pairs of initializer expressions related by homeomorphic embedding in an initial fragment of the stream (the length of this fragment being specified by an input parameter – n). We use only initializer expressions when checking for termination, because they are the only KNF part changed by the simple loop unrolling used here. To aid the experimentations on practical examples, there is also an input option, *alwaysSome*, which can be used to force a result even if no homeomorphic embedding is found in the specified initial stream segment.

```
Definition sscpCore (alwaysSome: bool) unroll knf2trm n (knf: KNFProg) :=
  let knfs := streamUnfold knf unroll in
  let ts := streamMap (fun knf ⇒ TrmToFOTerm (knf2trm knf)) knfs in
  match firstEmbedded TrmCons_eq_dec n ts with
  | None ⇒ if alwaysSome then Some (knfs n) else None
  | Some m ⇒ Some (knfs m)
  end.
```

```
Definition sscp (alwaysSome: bool) n knf :=
  sscpCore alwaysSome unrollToInit initExp n knf.
```

Alternatively, we define a cut-down version, which uses *normConv* instead of *norm* during loop unrolling. In essence it is a simple deforestation analog of the simple supercompiler above:

```
Definition unrollToInit' knf :=
  let nrm t := ntrm2trm (normConv t) in
  let newInit := nrm ((IfNil (condExp knf) Id (bodyExp knf)) $ (initExp knf))
  in MkKNFProg newInit (condExp knf) (bodyExp knf) (finalExp knf).
```

```
Definition sscp' (alwaysSome: bool) n knf :=
  sscpCore alwaysSome unrollToInit' initExp n knf.
```

Now we can see both methods at work, demonstrating the usefulness of even this limited form of supercompilation. Consider again the usual Lisp-like encoding of booleans and lists in the domain of binary trees. The task of checking if an input list of booleans contains at least one false value can be performed by the following program:

```
Definition listHasWFalse_knf :=
  let WFalse := Nil in let WTrue := Nil # Nil in MkKNFProg
  (Id # WFalse) Hd (IfNil (Hd $ Hd) (Nil # WTrue) ((Tl $ Hd) # Tl)) Tl.
```

```
Eval compute in (KNFtoProg listHasWFalse_knf).
```

```
  = VAR  <- Id # Nil;
    WHILE Hd DO
      VAR  <- IfNil (Hd $ Hd) (Nil # Nil # Nil) (Tl $ Hd # Tl)
    DONE;
    VAR  <- Tl : SWhileStmt
```

A few explanations are in order. We extend the computation state with a flag to hold the final result – at position *Tl* – while keeping the original input list at position *Hd*. Then we loop while the list is not empty, and test its head. If it is *VNil*, we make the list empty to force an exit of the loop, and set the result to true, otherwise we remove the list head and continue.

Next, we introduce a specialized version of this program, which, if the input list is not empty, adds a negated copy of the head of the list. The idea is clearly that this specialized version should return true on all non-empty lists, and false only on the empty list.

```
Definition modifyKNFinput knf modifierExp := MkKNFProg
  ((initExp knf) $ modifierExp) (condExp knf) (bodyExp knf) (finalExp knf).

Definition listHasWFalse_knf_negdupHd :=
  let WFalse := Nil in let WTrue := Nil # Nil in
  let negate x := IfNil x WTrue WFalse in
  modifyKNFinput listHasWFalse_knf (IfNil Id Id (negate Hd # Id)).

Eval vm_compute in (match sscp false 3 listHasWFalse_knf_negdupHd with
  | Some knf ⇒ Some (KNFtoProg knf) | None ⇒ None end).
```

```
  = Some (VAR <- IfNil Id (Nil # Nil)
            (IfNil Hd (Nil # Nil # Nil) (Nil # Nil # Nil));
          WHILE Hd
          DO VAR  <- IfNil (Hd $ Hd)
            (Nil # Nil # Nil) (Tl $ Hd # Tl)
          DONE; VAR  <- Tl) : option SWhileStmt
```

While the resulting program may not look simplified at first, if we remove by hand the second if-expression with equal branches, we can see that the loop will never be entered. The final correct result will be computed directly by the initializer expression. The combination of deforestation, positive information propagation and simple loop unrolling has resulted in an almost optimal specialized program in this case.

```
  = Some (VAR <- IfNil Id (Nil # Nil) (Nil # Nil # Nil);
          WHILE Hd
          DO VAR  <- IfNil (Hd $ Hd)
            (Nil # Nil # Nil) (Tl $ Hd # Tl)
          DONE; VAR  <- Tl) : option SWhileStmt
```

In contrast, if we remove just positive information propagation from the mix, the end result is much less satisfactory:

```
Eval vm_compute in (match sscp' false 2 listHasWFalse_knf_negdupHd with
  | Some knf ⇒ Some (KNFtoProg knf) | None ⇒ None end).
```

```
  = Some (VAR  <- IfNil Id
          (IfNil Id (IfNil Id Id (IfNil Hd (Nil # Nil) Nil # Id) # Nil)
      (IfNil Id (IfNil Hd (Nil # Nil # Nil) (IfNil Id Tl Id # Nil))
         (IfNil Hd (IfNil Id Tl Id # Nil) (Nil # Nil # Nil))))
   (IfNil Id (IfNil Hd (Nil # Nil # Nil) (IfNil Id Tl Id # Nil))
      (IfNil Hd (IfNil Id Tl Id # Nil) (Nil # Nil # Nil)));
WHILE Hd
DO VAR  <- IfNil (Hd $ Hd) (Nil # Nil # Nil) (Tl $ Hd # Tl) DONE;
VAR  <- Tl) : option SWhileStmt
```

3.4 Proof of Correctness of the Full Supercompiler

We consider two aspects of supercompiler correctness - totality and preservation of semantics. Totality of the supercompiler function is a direct consequence of totality of *firstEmbedded* (`Theorem` *firstEmbedded_total*).

`Lemma` *sscpCore_total*: $\forall$ *b unroll knf2trm knf*, $\exists$ *n*, $\exists$ *knf1*,
sscpCore b unroll knf2trm n knf = *Some knf1.*

`Theorem` *sscp_total*: $\forall$ *b knf*, $\exists$ *n*, $\exists$ *knf1*, *sscp b n knf* = *Some knf1.*

Preservation of semantics, on the other hand, is established through a sequence of lemmas, essentially relying only on correctness of one-step loop unrolling (*evalKNF_unrollToInit*). We can say that we have achieved one of the main goals of this study - maximum modularity in proving different aspects of supercompiler correctness.

`Lemma` *condExp_unrollToInitStream*: $\forall$ *knf n*,
condExp (*streamUnfold knf unrollToInit n*) = *condExp knf.*

`Lemma` *unrollToInitStream_evalKNF_fw*: $\forall$ *knf v1 v2 n d1*,
strictTrm (*condExp knf*) $\rightarrow$ *evalKNF d1 knf v1* = *Some v2* $\rightarrow$
$\exists$ *d2*, *evalKNF d2* (*streamUnfold knf unrollToInit n*) *v1* = *Some v2.*

`Lemma` *unrollToInitStream_evalKNF_bw*: $\forall$ *knf v1 v2 n d1*,
strictTrm (*condExp knf*) $\rightarrow$ *evalKNF d1* (*streamUnfold knf unrollToInit n*)
v1 = *Some v2* $\rightarrow$ $\exists$ *d2*, *evalKNF d2 knf v1* = *Some v2.*

`Lemma` *sscpCore_correct_fw*: $\forall$ *b knf knf1 n d1 v1 v2*, *strictTrm* (*condExp knf*)
$\rightarrow$ *sscpCore b unrollToInit initExp n knf* = *Some knf1* $\rightarrow$
evalKNF d1 knf v1 = *Some v2* $\rightarrow$ $\exists$ *d2*, *evalKNF d2 knf1 v1* = *Some v2.*

`Lemma` *sscpCore_correct_bw*: $\forall$ *b knf knf1 n d1 v1 v2*, *strictTrm* (*condExp knf*)
$\rightarrow$ *sscpCore b unrollToInit initExp n knf* = *Some knf1* $\rightarrow$
evalKNF d1 knf1 v1 = *Some v2* $\rightarrow$ $\exists$ *d2*, *evalKNF d2 knf v1* = *Some v2.*

`Lemma` *sscpCore_correct*: $\forall$ *b knf knf1 n v1 v2*, *strictTrm* (*condExp knf*)
$\rightarrow$ *sscpCore b unrollToInit initExp n knf* = *Some knf1* $\rightarrow$
(($\exists$ *d1*, *evalKNF d1 knf v1* = *Some v2*) $\leftrightarrow$

$(\exists\ d2,\ evalKNF\ d2\ knf1\ v1 = Some\ v2))$.

`Theorem` $sscp\_correct$: $\forall\ b\ knf\ knf1\ n\ v1\ v2,$
$strictTrm\ (condExp\ knf) \rightarrow sscp\ b\ n\ knf = Some\ knf1 \rightarrow$
$((\exists\ d1,\ evalKNF\ d1\ knf\ v1 = Some\ v2) \leftrightarrow$
$(\exists\ d2,\ evalKNF\ d2\ knf1\ v1 = Some\ v2))$.

4 Related Work

Since Turchin's ground-breaking work on supercompilation of Refal has gained popularity [21], a number of supercompilers have appeared for different languages ([5,17,18,3,13], to mention just a few). The supercompiler described in this work is most closely related to the formulation of positive supercompilation by Sørensen at al [18]. In contrast to other treatments of supercompilation, which typically use either substitutions or environments on the meta-level in order to propagate information in conditional branches, we use a form of object-level explicit substitutions. Explicit substitutions (introduced by Abadi et al [1]) are by no means a new technique, and have been used, with varying details, in many other contexts. In the context of supercompilation, they were previously applied by the author in his PhD thesis [11], but not with the aim to simplify formal proofs of correctness.

Studies have been published on general frameworks for proving semantics preservation and termination of supercompilers and similar program transformers - such as [15,16]. To the best of our knowledge, there has been no previous work on formally verifying the correctness of a supercompiler implementation. At the same time, as numerous formal proof assistants grow mature, we see more and more computer-checked proofs of correctness for practical systems. We have already mentioned two impressive examples – the Compcert compiler from a large subset of C to a real assembler language [12], and the seL4 operating system microkernel [10].

Many treatments of supercompilation (and related studies like optimal self-application) use either a small subset of an existing language (like Core Haskell [13] or FlatCurry [3]), or a tiny language operating on Lisp-like well-founded binary trees. Languages like I/LOOP [8,7] and S-Graph/TSG [5,2] were an important source of inspiration for the development of SWhile. Non-Turing-complete languages have long been a subject of research in computability and computational complexity theory (e.g. [9]). Our language of simple expressions can, in particular, be seen as a generalization – from the domain of natural numbers to the domain of binary trees – of the language of "simple programs" of Tsichritzis [20]. The explicit use of a non-Turing-complete language to formulate parts of the driving process in supercompilation appears to be a new result of this work.

The standard formulation of positive supercompilation distinguishes 4 phases - driving (including unfolding), "whistle", folding, and generalization [18]. The separation of unfolding from the rest of the driving transformations, and their splitting in two parts (normalization + positive information propagation) is a new result, although its roots can be traced to the author's previous work [11]. As for the definition of normalization itself, similar transformations have been used in many different contexts (e.g. [7]), unrelated to supercompilation.

5 Conclusions and Future Work

We have achieved a full formal verification, in Coq, of a greatly simplified supercompiler for a basic imperative language operating on binary-tree data. To the best of our knowledge, this is the first attempt of a computer-checked proof of correctness of a supercompiler or a similar transformer. An advantage of our method is that it leads to a small-size formalization and verification source – about 1100 non-empty, non-comment lines of Coq code[3], of which about 45% are definitions, and the rest are proofs. As a comparison, the formal verification of the Compcert compiler amounts to about 42000 lines of Coq code [12].

As another advantage, our verification is organized in a very modular way, thanks to a new refactoring of the supercompilation process into smaller pieces that can be checked independently. We believe that this modularity makes the approach more re-usable in different contexts.

There are a couple of important directions for future improvements. Firstly, our object language is so simple that it is hard to read and very hard to program in. While this is normal for a toy language crafted for research purposes, the effort to improve its usability may be worthwhile. Making the language easier to program in is the simpler task – we can always add arbitrary amount of syntax sugar in a preprocessing phase, or even a compiler from a higher-level language to SWhile. The more challenging task is to make the program resulting from supercompilation easier to understand. Some form of post-processing transformations may help, but probably will not be sufficient by themselves.

Secondly, our supercompiler is crippled in its current form, due to its very limited definition of loop unrolling. We have tried some other, seemingly more powerful forms of unrolling, omitted from this text. Unfortunately the preliminary experimentation on simple examples does not show good results. Further research is needed to find a more powerful form of loop unrolling, if we want to pass the "KMP test" [5,18]. Another interesting option to pursue is to switch from an imperative language with a single while loop to a functional one with a single recursive function (like the language F of N. D. Jones [8]). This switch might also be beneficial for the readability of the resulting programs. A challenge for this approach would be to keep a clean separation between unfolding and the other parts of driving, but it appears feasible.

Beside the improvements of the method suggested above, some practical applications might be interesting to study. For example, we could re-use parts of the current development as new proof tactics in Coq, using the mechanism of proof by reflection [4]. Another idea is to re-use the decomposition of supercompilation in smaller parts, that has helped the current development, for repeating

[3] as reported by the coqwc tool

the verification in another proof system, possibly one based on supercompilation itself.

6 Acknowledgments

The author would like to express his gratitude to the four anonymous referees, to Prof. Iwan Tabakow and to Prof. Andrei Klimov, whose comments and suggestions helped improve the article. This study would not be possible without the great work of the team developing Coq [19]. The author found both the introductions of B. C. Pierce et al [14] and A. Chlipala [4] invaluable in learning Coq and automated theorem proving in general.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *J. Funct. Program.*, 1(4):375–416, 1991.
2. S. M. Abramov. *Metavychisleniya i ih primenenie (Metacomputation and its applications).* Nauka, Moscow, 1995.
3. Elvira Albert, Michael Hanus, and Germán Vidal. A practical partial evaluation scheme for multi-paradigm declarative languages. *Journal of Functional and Logic Programming*, 2002, 2002.
4. Adam Chlipala. Certified programming with dependent types (book draft). `http://adam.chlipala.net/cpdt/`, 2010.
5. Robert Glück and Andrei V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, M. Falaschi, G. Filé, and A. Rauzy, editors, *Static Analysis. Proceedings*, volume 724 of *Lecture Notes in Computer Science*, pages 112–123. Springer-Verlag, 1993.
6. David Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, 1980.
7. Lars Hartmann, Neil D. Jones, and Jakob Grue Simonsen. Interpretive overhead and optimal specialisation. or: Life without the pending list (workshop version). In V.F. Turchin, editor, *International Workshop on Metacomputation in Russia, Meta2008*, pages 1–12, 2008.
8. Neil D. Jones. *Computability and Complexity from a Programming Perspective.* Foundations of Computing. MIT Press, Boston, London, 1 edition, 1997.
9. Neil D. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, january 2001. Special Issue on Functional Programming and Computational Complexity.
10. Gerwin Klein, Philip Derrin, and Kevin Elphinstone. Experience report: seL4: formally verifying a high-performance microkernel. In Graham Hutton and Andrew P. Tolmach, editors, *ICFP*, pages 91–96. ACM, 2009.
11. Dimitur N. Krustev. Software test generation using program skeletons. PhD thesis Technical Report PRE 23/01, Wroclaw Polytechnic, Wroclaw, Poland, 2001.
12. Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
13. Neil Mitchell and Colin Runciman. A supercompiler for core Haskell. In Olaf Chitil et al., editor, *IFL 2007*, volume 5083 of *LNCS*, pages 147–164. Springer-Verlag, May 2008.

14. Benjamin C. Pierce, Chris Casinghino, and Michael Greenberg. Software foundations (book draft). `http://www.cis.upenn.edu/~bcpierce/sf/`, 2010.
15. David Sands. Proving the correctness of recursion-based automatic program transformations. *Theor. Comput. Sci.*, 167(1&2):193–233, 1996.
16. Morten Heine Sørensen. Convergence of program transformers in the metric space of trees. In J. Jeuring, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Computer Science*, pages 315–337. Springer-Verlag, 1998.
17. Morten Heine Sørensen and Robert Glück. An algorithm of generalization in positive supercompilation. In J.W. Lloyd, editor, *Logic Programming: Proceedings of the 1995 International Symposium*, pages 465–479. MIT Press, 1995.
18. Morten Heine Sørensen and Robert Glück. Introduction to supercompilation. In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation: Practice and Theory*, volume 1706 of *Lecture Notes in Computer Science*, pages 246–270. Springer-Verlag, 1999.
19. Coq Development Team. The Coq proof assistant reference manual, version 8.2, February 2009.
20. Dennis Tsichritzis. The equivalence problem of simple programs. *J. ACM*, 17(4):729–738, 1970.
21. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.
22. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

Supercompilation and Normalisation By Evaluation

Gavin E. Mendel-Gleason and Geoff Hamilton

Lero@DCU
School of Computing
Dublin City University

Abstract. It has been long recognised that partial evaluation is related to proof normalisation. Normalisation by evaluation, which has been presented for theories with simple types, has made this correspondence formal. Recently Andreas Abel formalised an algorithm for normalisation by evaluation for System F. This is an important step towards the use of such techniques on practical functional programming languages such as Haskell which can reasonably be embedded in relatives of System F_ω. Supercompilation is a program transformation technique which performs a super-set of the simplifications performed by partial evaluation. The focus of this paper is to formalise the relationship between supercompilation and normalisation by evaluation for System F with recursive types and terms.

1 Introduction

Partial evaluation has arisen in two rather distinct settings. The first setting is in practical attempts to improve program performance. The second is the use of evaluation to produce normal forms for proofs in a Curry Howard setting [20].

The use of a term language with recursion, however, puts fundamental limits on the practical use of evaluation as a tool for normalisation or optimisation. Due to this fact the program transformation community has developed a number of tools for improving the performance of programs, including deforestation [27], fusion [14] and supercompilation [18].

Supercompilation [24] is a program transformation which performs a super-set of the optimisations performed by fusion and deforestation. Supercompilation is a complex program transformation making using of *folds* [3]. Folds, which introduce new recursive structures, can sometimes introduce non-termination so certain side conditions must be met in order to ensure their correctness.

When using program transformations for languages which are not strongly normalising, it is important to ensure that the transformation preservers the meaning of the program. Bisimilarity, a technique developed by Milner, [13] was used by Gordon [7] as an alternative to context equivalence for showing semantic equivalence of programs in all contexts. This is achieved by associating terms with transition systems, and showing bisimilarity of the transition systems. The technique can be usefully applied to automatic program transformations in order to prove correctness.

This paper introduces several novel developments. It demonstrates a transition system framework for System F with recursive types. This allows us to define a bisimilarity relation on recursive terms in System F which demonstrates behavioural equivalence. We use these transition systems as a semantic domain to present a system which closely resembles Normalisation by Evaluation (NbE) [2] [1]. The techniques which are already common place in the supercompilation and meta-computation communities of creating partial process trees and then extracting programs are formalised in such a way as to demonstrate the connection with NbE.

2 Normalisation By Evaluation

NbE makes use of two functions: a semantic interpretation function $f : Syntax \rightarrow Semantics$, and a reification function $g : Semantics \rightarrow Syntax$. The idea behind the technique is that we can arrive at a normal form for a term t (a unique syntactic description) by transforming into the semantic domain $f(t)$ and then reifying as a term $t' = g(f(t))$. This normalisation corresponds with eliminations of *cuts* from the type tree. We will see this in more detail later.

In our presentation, the semantic interpretation function is the supercompilation algorithm, which presents a transition system as the semantic representative. The reason for this is that labelled transition systems serve as a compact representation of potential program traces. When we want to establish bisimilarity, we need a formalism in which to show that we have an observable trace equivalence between programs. This idea was expressed by Turchin in [22]. This trace equivalence can be established between two systems using a bisimulation of transition systems.

Reification is performed by *program extraction* which provides us, again, with a term in our original source language.

While the techniques given here are very similar to ones used in NbE, it must be stressed that positive supercompilation will not provide unique normal forms for arbitrary terms. While it provides normal forms for any finitary expression without function symbols, it can not provide a *unique* finite representation for all transition systems. A simple proof based on the Full-Employment theorem suffices to show that any attempt to do so, except for sub-Turing complete languages will fail.

Theorem 1. *There is no normal form, giving syntactic equivalence modulo α-renaming and function symbol renaming, for arbitrary terms t for a Turing complete functional programming language.*

Proof. Assume a normalisation function f. We can apply this function to a program Ω, known not to terminate, to obtain a canonical term $f(\Omega) = \Omega_c$. We may now test any term t for halting by applying f and comparing to Ω_c syntactically, violating the Halting theorem. □

So if we can not obtain normal forms, the question naturally arises, why should we view more general program transformation through a similar lens to the one given by NbE? One answer is that in some instances the technique can be used for deciding the equality of terms as in [10] [11]. The fragment of applicability may in fact be quite large and could include languages with infinite transition systems. In addition, the use of transition systems for the semantic domain means that we can use bisimulation equivalence of transition systems as a means by which to justify the substitution of programs generally enabling us to use it as a general tool for showing semantic equivalence of program transformations.

3 Language

The language we present is a functional programming language, which we will call Λ_F with a type system based on System F with recursive types. The use of System F typing allows us to ensure that transitions can be found for any term. Our term language will follow closely on the one used by Abel [1]. We will use two distinct sets of variables for our exposition, term variables x, y drawn from the set **Var** and type variables X drawn from the set **TyVar**.

$$\begin{array}{lll}
\mathbf{Fun} \ni f, g & & \text{Function Symbols} \\
\mathbf{Ty} \ni A, B, C & ::= 1 \mid X \mid A \to B \mid \forall X.A \mid A + B \mid A \times B & \text{Types} \\
& \mid \nu X.A & \\
\mathbf{Tr} \ni r, s, t & ::= x \mid f \mid () \mid \lambda x : A.t \mid \Lambda X.t \mid r\ s \mid r\ A & \text{Terms} \\
& \mid \mathrm{inl}(t) \mid \mathrm{inr}(t) \mid (t, s) & \\
& \mid \mathrm{in}(t, A) \mid \mathrm{out}(t, A) & \\
& \mid \mathrm{case}\ r\ \mathrm{of}\ \mathrm{inl}(x_1) \Rightarrow s\ ;\ \mathrm{inr}(x_2) \Rightarrow t & \\
& \mid \mathrm{split}\ r\ \mathrm{as}\ x_1, x_2\ \mathrm{in}\ s & \\
\mathbf{Ctx} \ni \Gamma & ::= \cdot \mid \Gamma, X \mid \Gamma, x : A & \text{Contexts}
\end{array}$$

We will describe *substitutions* using the map σ which will represent assignment of variables to terms and type variables to types. Extension of a substitution will be written as $\sigma \cup (x, t)$ or $\sigma \cup (X, A)$. We will use a function $FV(t)$ to obtain the free type and term variables from a term. Substitutions of a single variable will be written $[X := A]$ or $[x := t]$ for type and term variables respectively.

In order to simplify our presentation, we will also need to introduce recursive terms. This change is the point of departure between this work and standard presentations of NbE and our framework for supercompilation.

Recursive terms will be represented using function constants. Function constants will be drawn from a set **F**. We will couple our terms with a function Δ which associates a function constant f with a term e, $\Delta(f) = e$, where e may itself contain any function constants in the domain of Δ.

The use of Δ will allow us to use arbitrary recursive and mutually recursive function definitions. In so doing, however, we will need to add a rule to System F which will make our type theory potentially unsound in a way which depends on the definition of Δ. The simplest example is given by $\Delta(f) = f$ which will clearly be well typed in our system for an arbitrary type A. This is the usual case for functional programming languages, and we hope to demonstrate how unsoundness can sometimes be remedied to produce a constructive type theory in a future paper.

For a term t with type T in a context Γ we will write $\Gamma \vdash t : T$. The type derivation is given by the following rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x : A.t) : A \to B}$$

$$\frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash r\ s : B} \qquad \frac{\Gamma, X \vdash t : A}{\Gamma \vdash \Lambda X.t : \forall X.A}\ X \notin \mathbf{FV}(\Gamma)$$

$$\frac{\Gamma \vdash t : \forall A.T}{\Gamma \vdash t\ B : A[X := B]} \qquad \frac{\Gamma, f : A \vdash \Delta(f) : A}{\Gamma \vdash f : A}$$

$$\frac{}{\Gamma \vdash () : 1} \qquad \frac{\Gamma \vdash r : A \qquad \Gamma \vdash s : B}{\Gamma \vdash (r, s) : A \times B}$$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \mathrm{inl}(t) : (T + S)} \qquad \frac{\Gamma \vdash t : S}{\Gamma \vdash \mathrm{inr}(t) : (T + S)}$$

$$\frac{U = \nu X.T \qquad \Gamma \vdash t : T}{\Gamma \vdash \mathrm{out}(t, U) : T[X := U]}\ X \notin \mathbf{FV}(\Gamma) \qquad \frac{U = \nu X.T \qquad \Gamma \vdash t : T[X := U]}{\Gamma \vdash \mathrm{in}(t, U) : U}$$

$$\frac{\Gamma \vdash e : T + S \qquad \Gamma, x : T \vdash t : U \qquad \Gamma, y : S \vdash s : U}{\Gamma \vdash (\mathrm{case}\ e\ \mathrm{of}\ \mathrm{inl}(x) \Rightarrow t\ ;\ \mathrm{inr}(y) \Rightarrow s) : U}$$

$$\frac{\Gamma \vdash s : T \times S \qquad \Gamma, x : T, y : S \vdash t : U}{\Gamma \vdash (\mathrm{split}\ s\ \mathrm{as}\ x_1, x_2\ \mathrm{in}\ t) : U}$$

System F without recursive types is strongly normalising, however due to the inclusion of infinite types via the $\nu X.\phi(X)$ type constructor, we can lose the strong normalisation property, even in the absence of function constant unfolding if our types are not restricted [26]. This can be seen by the simple example of the data type $D := \nu X.X \to X$ which. A concrete representative of this type which does not terminate, despite being well typed and having no function constants, is given by the term:

$$(\lambda f.\mathrm{out}(f\ \mathrm{in}(f, D), D))(\mathrm{in}((\lambda f.\mathrm{out}(f\ \mathrm{in}(f, D), D)), D))$$

We can, however, recover the normalisation property by imposing a positivity restriction on types [6].

With this restriction we can be assured that never encounter an infinite number of *transient* reductions [17], where *transient* reductions are those which do not result in a transition. This effectively allows us to ensure that all infinite behaviour will be present in the graph only, allowing us to carefully segregate out non-termination by other means.

$$(\lambda x : T.t)\ s \rightsquigarrow_\beta t[x := s] \qquad (\Lambda A : \kappa.t)\ T \rightsquigarrow_\tau t[A := T]$$

$$\frac{t \rightsquigarrow s}{\text{split } t\ \cdots \rightsquigarrow_\alpha \text{ split } s\ \cdots} \qquad \frac{t \rightsquigarrow s}{\text{case } t\ \cdots \rightsquigarrow_\alpha \text{ case } s\ \cdots}$$

$$\frac{t \rightsquigarrow t'}{t\ s \rightsquigarrow_\alpha t'\ s} \qquad \text{out}(\text{in}(s, T), T) \rightsquigarrow_\nu s$$

$$\frac{f \triangleq e \in \Delta}{f \rightsquigarrow_\delta e} \qquad \text{split } (r, s) \text{ as } x, y \text{ in } t \rightsquigarrow_\pi t[x := r, y := s]$$

$$\text{case inr}(t) \text{ of inl}(x) \Rightarrow r\ ;\ \text{inr}(y) \Rightarrow s \rightsquigarrow_\iota s[y := t]$$

$$\text{case inl}(t) \text{ of inl}(x) \Rightarrow r\ ;\ \text{inr}(y) \Rightarrow s \rightsquigarrow_\iota r[x := t]$$

$$\rightsquigarrow \equiv \rightsquigarrow_\beta \cup \rightsquigarrow_\alpha \cup \rightsquigarrow_\tau \cup \rightsquigarrow_\iota \cup \rightsquigarrow_\pi \cup \rightsquigarrow_\nu$$

We define a reduction relation which will allow us to simplify our proofs of a given type using a relation $\rightsquigarrow$. Notice the $\rightsquigarrow$ relation does not make use of function unfolding. The reason for this is that the $\rightsquigarrow_\delta$ relation may not reduce finitely, as with the example given previously $\Delta(f) = f$. By omitting $\rightsquigarrow_\delta$ we ensure that we can always obtain a head normal form which we will describe more completely later.

$t \rightsquigarrow^+ t'$	iff	$t \rightsquigarrow t' \vee t \rightsquigarrow t'' \wedge t'' \rightsquigarrow^+ t'$
$t \rightsquigarrow^* t'$	iff	$t = t' \vee t \rightsquigarrow^+ t'$
$t \not\Downarrow$	iff	$\neg\exists s.t \rightsquigarrow s$
$t \Downarrow h$	iff	$t \rightsquigarrow^* h$
$t \Downarrow$	::=	h where $t \Downarrow h$

Table 1: Derived Relations

In Table 1 we give some relations which are derived from the $\rightsquigarrow$ evaluation relation. Here $\rightsquigarrow^+$, the transitive closure of $\rightsquigarrow$ is taken to be the least fixed point of the recursive equation. The transitive reflexive closure $\rightsquigarrow^*$ is defined in terms of the transitive closure.

Because of our careful definition of $\rightsquigarrow$, we can use the relation $\Downarrow$ to arrive at a *head normal form*. This consists of an outermost syntactic form which has is either a value or a context.

Formally, divide the grammar of our language into the following classes:

$$\begin{array}{lll}
\mathbb{O} \ni o, p := () \mid \lambda x : A.t \mid \Lambda A.t \mid (r, s) & & \text{Observable} \\
\quad\quad\quad \mid \text{inl}(t) \mid \text{inr}(t) \mid \text{in}(t, A) & & \\
\mathbb{V} \ni v \quad := x \mid f \mid v\ t \mid v\ A & & \text{Irreducible} \\
\mathbb{E} \ni e \quad := - \mid \text{out}(e, A) & & \text{Context} \\
\quad\quad\quad := \text{case } e \text{ of inl}(x) \Rightarrow t\ ;\ \text{inr}(y) \Rightarrow s & & \\
\quad\quad\quad := \text{split } e \text{ as } x, y \text{ in } s & &
\end{array}$$

Where we view $e[t]$ as shorthand for the replacement of the privileged variable $-$ with t in e, $e[- := t]$. Using this we can derive the following decomposition lemma.

Lemma 1 (Unique Decomposition). *A term s such that $t \Downarrow s$ can be written as $e[v]$ or $s \in \mathbb{O}$.*

Proof (Proof Sketch). We proceed by induction on type derivations for a term t, starting with the empty context.

If $t \in \mathbb{O}$ then the context is empty. This follows from the fact that any type correct context for an element of $\mathbb{O}$ would lead to a reduction.

If $t \in \mathbb{V}$ then we are done.

For the case $t = \text{case } t' \text{ of inl}(x) \Rightarrow s\ ;\ \text{inr}(y) \Rightarrow r$: By the inductive hypothesis, $t' = e[v]$ since otherwise $t' \in \mathbb{O}$ which would reduce. Therefore $t = (\text{case } e \text{ of inl}(x) \Rightarrow s\ ;\ \text{inr}(y) \Rightarrow r)[v]$, and $(\text{case } e \text{ of inl}(x) \Rightarrow s\ ;\ \text{inr}(y) \Rightarrow r) \in \mathbb{E}$.

The remaining cases follow similarly. □

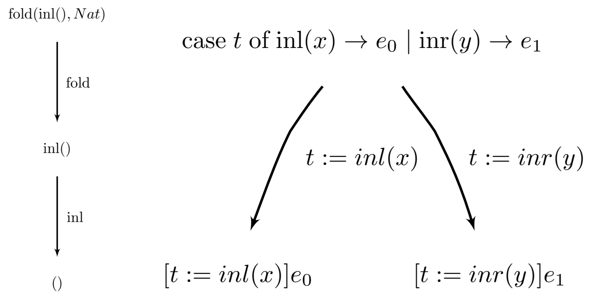
4 Transition System

For the purposes of reasoning about functional programs, and indeed the meaning of types themselves it is useful to use transition systems as a semantic domain. This approach is related to the infinite histories approach taken by Turchin [23] and is also quite close to the approach taken in process calculi such as CCS [12] and CSP [8]. It is also similar to the account given by monoidal histories [5]. The framework given here is based on the one given by Gordon in [7].

The basic idea behind the approach presented here is that terms are potentially infinite trees, and *variables* represent parametrisation with respect to an unknown transition system of appropriate type. Types themselves can be interpreted as restrictions on the form of the transition system.

An example of a value term which is represented by a finite tree is the inhabitant of the type $Nat := \nu X.1 + X$ which we can call zero, $\text{in}(\text{inl}(()), Nat)$, given in a Church-numeral style encoding, which is demonstrated in Fig. 1a.

In the interpretation of a term with variables, we will view the variables' semantics as being a parametrisation of transition system at the type of that variable. The parametrisation with respect to a transition system may be seen as an *external choice* non-determinism. That is, operationally, an equivalent program must have the same behaviour given the same external choices. Our various available reductions which perform substitution are an internalisation of



(a) Zero (b) Case

Fig. 1: Transition Systems

a choice which has become known. We see an example of this in the transition system in Fig. 1b.

Formally, a transition system is a structure which consists of a collection of states and actions and a relation which associates states via some action. Formally such a system is described by a tuple as follows:

$$\mathcal{T} = (\mathcal{S}, \mathcal{A}, \delta : \mathcal{S} \times \mathcal{A} \times \mathcal{S}),$$

Where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions and $\delta(s, \alpha, s')$ is a relation representing potential transitions from a state s to some state s' by way of some action $\alpha \in A$.

For our purposes, sets of states will be represented by programs, and transitions will be generated according to the operational behaviour of the program. Intuitively, we mean that the behaviour of a program makes *choices* for a calling program.

Transitions resulting from δ-reductions are not *observable* in the sense that they do not have any visible operational behaviour to the caller. This might seem odd in that a non-terminating program is certainly different than one which terminates. However, an infinite number of δ transitions is a failure to make a choice and is equivalent (that is, bisimilar) to a transition system with no further edges.

Formally we mean that we will not distinguish two transition systems which have transitions with arbitrarily many different transitions resulting from *delta*-reductions. This will be made explicit when we talk about *bisimiliarity* later. We will however explicitly notate them in our graphs for book keeping, in order to help us to reason about termination behaviour.

We define L_Γ as the set of terms in some typing context Γ having type derivations.

We define the function $\Xi : L_\Gamma \to \mathcal{T}$ as a function taking derivations in our language to a transition system. We will abbreviate the application of Ξ to some context Γ and a well typed term $\Gamma \vdash t : T$ as $\Xi[t]_\Gamma$.

We assume that variables are renamed to avoid capturing as is the case with application of lambda terms. In practice this can be done using a locally nameless representation.

We will overload the meaning of the $\cup$ operator in order that we can use the notation $(r, a, s) \cup \Xi[t]$ to denote the transition system $(\{t, t'\} \cup S, \{a\} \cup A, \{(r, a, s)\} \cup \delta)$ where $\Xi[t] = (S, A, \delta)$. The application of $\cup$ to two transition systems will be given by their component-wise unions. We will write the empty transition system as Ω.

We will also abuse the notation for substitution, writing $[v := t]$ for the replacement of some irreducible term v with a term t. This replacement is justified since we will retain the same termination behaviour as v is in a reduct position.

We can now generate the transition system of a term t by applying Ξ to the irreducible term $t' = t \Downarrow$. Ξ will convert terms in head normal form to transition systems.

$$
\begin{aligned}
\Xi[\lambda x : A.t]_\Gamma &::= (\lambda x : A.t, x : A, t \Downarrow) \cup \Xi[\Downarrow]_{\Gamma, x:T} \\
\Xi[\Lambda A.t]_\Gamma &::= (\Lambda A.t, A, t \Downarrow) \cup \Xi[t \Downarrow]_{\Gamma, A} \\
\Xi[()]_\Gamma &::= \Omega \\
\Xi[(r, s)]_\Gamma &::= ((r, s), \mathrm{fst}, r \Downarrow) \cup ((r, s), \mathrm{snd}, s \Downarrow) \cup \Xi[r \Downarrow]_\Gamma \cup \Xi[s \Downarrow]_\Gamma \\
\Xi[\mathrm{inl}(t)]_\Gamma &::= \mathrm{inl}(t) \xrightarrow{\mathrm{inl}} \Xi[t \Downarrow]_\Gamma \\
\Xi[\mathrm{inr}(t)]_\Gamma &::= inr(t) \xrightarrow{\mathrm{inr}} \Xi[t \Downarrow]_\Gamma \\
\Xi[\mathrm{in}(t, A)]_\Gamma &::= \mathrm{in}(t, A) \xrightarrow{out} \Xi[t \Downarrow]_\Gamma \\
\Xi[\mathrm{out}(t, A)]_\Gamma &::= (\mathrm{out}(t, A), out, t \Downarrow) \cup \Xi[t \Downarrow]_\Gamma \\
\Xi[e[v]]_\Gamma &::= (\{v\}, v, \emptyset) \cup \Omega \\
\Xi[e[f]]_\Gamma &::= (e[f], \delta f, e[\Delta(f)]) \cup \Xi[e[\Delta(f)]]_\Gamma
\end{aligned}
$$

$$
\begin{aligned}
&\Xi[e[\text{case } v \text{ of inl}(x) \Rightarrow r \;;\; \mathrm{inr}(y) \Rightarrow s]]_\Gamma ::= \\
&\quad (e[\text{case } v \text{ of inl}(x) \Rightarrow r \;;\; \mathrm{inr}(y) \Rightarrow s], (v := \mathrm{inl}(x_1)), e[r[v := \mathrm{inl}(x)]]) \cup \\
&\quad (e[\text{case } v \text{ of inl}(x) \Rightarrow r \;;\; \mathrm{inr}(y) \Rightarrow s], (v := \mathrm{inr}(x_2)), e[s[v := \mathrm{inr}(y)]]) \cup \\
&\quad \Xi[e[r[v := \mathrm{inl}(x)]]]_\Gamma \cup \Xi[e[s[v := \mathrm{inl}(y)]]]_\Gamma
\end{aligned}
$$

$$
\begin{aligned}
&\Xi[e[\text{split } v \text{ as } x, y \text{ in } r]]_\Gamma ::= \\
&\quad (e[\text{split } v \text{ as } x_1, x_2 \text{ in } e], (v := (x_1, x_2)), r[v := (x_1, x_2)]) \cup \\
&\quad \Xi[r[v := (x_1, x_2)]]_\Gamma
\end{aligned}
$$

Notice the addition of the δ transition for a function variable f. This transition will not be directly observable, but will be used for book keeping, so that we can avoid the problem of non-termination at nodes. Now that we have a transition system for each term in the above described normal form, we can be assured of a having a (potentially infinite) transition system for every program.

We carry the context Γ through the computation because it is needed later for generalisation and abstraction. In an implementation it is convenient to perform the transformation on derivations rather than terms, such that the appropriate context is always present.

Once transition systems are given for terms, we can proceed to define bisimilarity. Bisimilarity is a coinductive equality relation. If two terms are bisimilar, we should not be able to distinguish them by any number of *experiments* on the terms. This is effectively identical to contextual equivalence, but allows us to look directly at the transition systems to establish bisimilarity, rather than having to cope with quantification over contexts. The technical machinery is consequently less complex [7].

Bisimilarity is defined as a relation between two transition systems with the following definition.

Definition 1 (Bisimilarity). *A term s is* bisimilar *to a term t, written $s \sim t$, if the following two conditions hold:*

– $\forall (s, \alpha, s') \in \delta \to \exists (t, \alpha, t') \in \delta \wedge s' \sim t'$
– $\forall (t, \alpha, t') \in \delta \to \exists (s, \alpha, s') \in \delta \wedge s' \sim t'$

Since bisimilarity is the greatest fixed point of such a relation, we need only to produce a relation that demonstrates these properties, and it will be a subset of the bisimilarity relation.

In order to make use of transition systems for our theory however, we will also need to make use of a notion of composition. This will allow us to generalise transition systems and to make them parametric. The basic idea is to make explicit a notion of composition of transition systems such that the following theorem holds. This notion of composition is similar to the idea of composition for normalisation by partial evaluation [1].

Definition 2 (Composition). *Composition of trees is achieved by replacement of states in the transition system or replacement of labels on transitions.*

$$
\begin{aligned}
&\Xi[\lambda x : T.s]_\Gamma \cdot \Xi[t] ::= \Xi[s[x := t] \Downarrow]_\Gamma\\
&\Xi[\Lambda A.s]_\Gamma \cdot T ::= \Xi[s[A := T] \Downarrow]_\Gamma\\
&\Xi[v]_\Gamma \cdot \Xi[r]_\Gamma ::= (v\ r, v\ r, \emptyset) \cup \Omega\\
&\Xi[e[\mathit{case}\ v\ \mathit{of}\ \mathit{inl}(x) \Rightarrow r\ ;\ \mathit{inr}(y) \Rightarrow s]]_\Gamma \cdot \Xi[t]_\Gamma ::=\\
&\quad (e[\mathit{case}\ v\ \mathit{of}\ \mathit{inl}(x) \Rightarrow r\ ;\ \mathit{inr}(y) \Rightarrow s]\ t, (v := \mathit{inl}(x)), (e[r[v := \mathit{inl}(x)]]\ t) \Downarrow) \cup\\
&\quad \Xi[(e[r[v := \mathit{inl}(x)]]\ t) \Downarrow]_{\Gamma, x:A} \cup\\
&\quad (e[\mathit{case}\ v\ \mathit{of}\ \mathit{inl}(x) \Rightarrow r\ ;\ \mathit{inr}(y) \Rightarrow s]\ t, (v := \mathit{inr}(y)), (e[s[v := \mathit{inr}(y)]]\ t) \Downarrow) \cup\\
&\quad \Xi[(e[s[v := \mathit{inr}(y)]]\ t) \Downarrow]_{\Gamma, y:B}\\
&\Xi[e[\mathit{split}\ v\ \mathit{as}\ x, y\ \mathit{in}\ s]]_\Gamma \cdot \Xi[r]_\Gamma ::=\\
&\quad (e[\mathit{split}\ v\ \mathit{as}\ x, y\ \mathit{in}\ e]\ r, (v := (x, y)), (e[v := (x, y)]\ r) \Downarrow) \cup\\
&\quad \Xi[(e[v := (x, y)]\ r) \Downarrow]_{\Gamma, x:A, y:B}\\
&\Xi[e[f]]_\Gamma \cdot \Xi[r]_\Gamma ::= (e[f], \delta f, e[\Delta(f)]) \cup (\Xi[e[\Delta(f)]]_\Gamma \cdot \Xi[r]_\Gamma)
\end{aligned}
$$

Theorem 2 (Composition is Bisimilar). *$\Xi[t]\cdot\Xi[s]$ is defined whenever $\Xi[t\ s]$ is defined and enjoys the property that $\Xi[t\ s] \sim \Xi[t] \cdot \Xi[s]$.*

Proof. The proof is by construction making use of substitution and the λ and Λ cases are therefore trivial. The case and split cases also provide identical transitions in both cases. The remaining case is function constant unfolding in a context.

If we have $t = e[f]$ then $t\ s$ is $e[f]\ s$. Unfolding f in either situation must lead to either a new function constant to unfold, $t' = e[g]$ and $t'\ s$ is $e[g]\ s$ (with f not necessarily different than g). Either we eventually encounter a λ or Λ leading to a reduction, or we encounter an infinite number of function constant un-foldings in either case, which, since delta transitions are not distinguished by bisimilarity, are bisimilar.

We would now like to provide a reification of transition systems back into terms. However, in general these terms may be of infinite size. In order to ensure finite terms we will need to find a finite representation of our transition system. Essentially this requires the production of a graph using our transition system function Ξ and composition. Practically this can be achieved using supercompilation.

5 Supercompilation

Supercompilation is a program transformation framework first developed by Turchin [25]. Sørensen, Glück and Jones defined positive supercompilation [19], which is an algorithm for program transformation. We will present a system modeled on the positive supercompilation algorithm extended to deal explicitly with types in System F. We then show the correctness of this algorithm using bisimilarity.

Supercompilation uses the concepts of *driving*, *generalisation* and *folding*. *Driving* is the production of a *process tree* by way of normal order evaluation. For those familiar with supercompilation, the above descriptions of transition systems will look very familiar.

The difference between the two is largely in the explicit labeling of transitions, allowing bisimilarity to be defined, and the use of folding. To simplify the presentation we will not use the traditional formulation of driving, but we will proceed to define supercompilation directly in terms of transition systems.

Since process trees are potentially infinite, we will require some mechanism of creating a finitary representation. *Folding* involves describing a transition system in terms of states in the transition system which are α-equivalent, that is, equivalent modulo bound variable renaming. Instead of representing the entire potentially infinite unfolding, we can now *point back* to a prior state set in the transition system.

Definition 3 (Abstraction). *Given a type derivation $\Gamma \vdash t : T$, We can form a term, $\vdash \lambda\Gamma.s : \overrightarrow{S} \to T$ by taking each variable $(x, S) \in \Gamma$ and abstracting over t with $x : S$ and for each type variable X with $\Lambda X.t$.*

Theorem 3 (Abstraction is Bisimilar). *If a term t is a renaming of a term s, such that $t\sigma = s$ where σ are variables drawn from a typing context Γ, then $\Xi[t] \sim \Xi[\lambda\Gamma.t\ \overrightarrow{x}] \sim \Xi[\lambda\Gamma.t] \cdot \Xi[x_0] \cdot \dots \cdot \Xi[x_n]]$ where x_i is the ith variable in $\overrightarrow{x}$.*

The proof of this theorem follows directly from the composition lemma. This will allow us to re-use elements of the transition system previously encountered.

Generalisation can be considered the dual of unification and is sometimes called anti-unification [16]. Generalisation is defined over the semi-lattice induced by the *instantiation* ordering.

Definition 4 (Instantiation Ordering). *A term $t : A$ is said to be an instance of a term $s : B$, if there is a substitution θ such that $t : A = (s : B)\theta$. We write that t is an instance of s or $t : A \preceq s : B$. Similarly, a typing context Γ is said to be an instance of a typing context Γ' if each variable is an instance, or $\Gamma \preceq \Gamma'$.*

The least general generalisation of terms for System F is undecidable [9]. In our implementation we have restricted ourselves to generalisation of higher order patterns using a method similar to the one described in [15].

It is the case that any generalisation can be used for the purposes of assisting in the creation of a finite graph, but the particulars of the generalisation will affect the final form of our transition system.

Definition 5 (Generalisation). *A generalisation operator $r \sqcup s = (t, \theta_1, \theta_2)$ is defined such that $t\theta_1 = r$ and $t\theta_2 = s$.*

Now, to control the process of generating the process tree, we need to use the composition property and some relation that ensures we can find a finite representation of our potentially infinite transition system. It is typical to make use of the homeomorphic embedding [4] for this purpose. It is a well quasi-order and ensures that there are no infinite sequences of terms which can not be ordered provided that the set of function constants is finite. Any relation which ensures that unfolding is restricted to finite sequences is sufficient. We will call this our *whistle* relation.

It is useful to have a reflection to the original syntax which reproduces the original term, modulo naming of functions and variables. This will be used to ensure that we can recover a finite transition system even if we are unable to find new folds. We will use this in our supercompilation algorithm in order to *give up* when we may no longer proceed.

Definition 6 (Identity). *$Id[t]$ is defined as the transition system $\Xi[t]_\Gamma$ where every function call in reduct position is replaced with a composition $\Xi[f]_\Gamma \cdot \Xi[s]_\Gamma$. We may use this composition to produce a finite graph for any term, since the original term is itself finite.*

Definition 7 (Reachable). *A term t is said to be reachable from s if there is a sequence of terms t_i such that $(s, a, t_0) \in \delta$ and $(t_n, a, t) \in \delta$ and $(t_i, a, t_{i+1}) \in \delta$.*

Definition 8 (Ancestor). *A term t is said to be an* ancestor *of a term s if s is reachable by delta from t.*

The general form of supercompilation can now be described as follows.

Definition 9 (Supercompilation). *The positive supercompilation of a term t can be produced by lazily producing the transition system $\Xi[t']$ where $t' = t \Downarrow$. We will write the resulting finite representation of the transition system as $S[t']$.*

When a term s is encountered which has an ancestor r which satisfies the whistle relation, we have a number of cases:

If $s \preceq r$, then we abstract $\Xi[s]$ to obtain $\Xi[r] \cdot \Xi[\theta(x_0)] \cdot \cdots \cdot \Xi[\theta(x_n)]$ where θ is the substitution that proves the instantiation and x_i are the variables in theta. We continue the algorithm on each of the $\Xi[\theta(x_0)]$.

Otherwise, generalisation is applied to s and the term r, $s \sqcup r = (e_g, \theta_1, \theta_2)$. We use abstraction to write: $\Xi[r]_\Gamma$ $\Xi[\lambda\Gamma.e_g]_\Gamma \cdot \Xi[\theta_1(x_0)]_\Gamma \cdot \cdots \cdot \Xi[\theta_1(x_n)]_\Gamma$ and continue the algorithm on $\Xi[e_g]$ and each of the $\theta_1(x_i)$.

If generalisation fails, we can give up *using $Id[s]$. Otherwise we proceed on each term reachable from s by δ.*

6 Reification

Now that we have a suitable definition of bisimilarity, which captures the notion of even infinite program behaviours being identical, we can give a definintion for the reification of a term. This reflection back into terms is usually called program extraction or residuation in the meta-computation community.

We define the function $K(\tau)$ to return the set of transition systems starting at child nodes for some transition system τ. We will use the notation $r \xrightarrow{a} \tau$ to mean that τ is the transition system starting from $\delta(r, a)$.

Definition 10 (Reification). *Reification is defined on the structure of process trees in the following way.*

$if\, \tau\sigma \in K(\tau)\ then\ \Delta = (f, \Pi[\tau])$

$\Pi[\Xi[t] \cdot \Xi[s]] = \Pi[\Xi[t]]\ \Pi[\Xi[s]]$

$\Pi[r \xrightarrow{x:T} \tau]_\Delta = \lambda x : T.\Pi[\tau]_\Delta$

$\Pi[r \xrightarrow{A} \tau]_\Delta = \Lambda A.\Pi[\tau]_\Delta$

$$\begin{aligned}\Pi[r &\xrightarrow{x\ \overrightarrow{t}:=inl(x_1)} \tau_1,\\ r &\xrightarrow{y\ \overrightarrow{t}:=inr(x_2)} \tau_2]_\Delta = case\ (x\ \overrightarrow{t})\ of\ inl(x) \Rightarrow \Pi[\tau_1]_\Delta\ ;\ inr(y) \Rightarrow \Pi[\tau_2]_\Delta\end{aligned}$$

$\Pi[r \xrightarrow{x\ \overrightarrow{t}:=(x_1,x_2)} \tau]_\Delta = split\ (x\ \overrightarrow{t})\ as\ x_1, x_2\ in\ \Pi[\tau]_\Delta$

$\Pi[r \xrightarrow{fst} \tau, r \xrightarrow{snd} \psi] = (\Pi[\tau]_\Delta, \Pi[\psi]_\Delta)$

$\Pi[r \xrightarrow{\kappa} s] = \kappa(s)$
Where $\kappa \in \{inl, inr, in, out\}$

Theorem 4 (Reification). *The reification function* Π *of a transition system* τ *associates a term and function constant relation* Δ *with the transition system such that the following holds:*

$$S[\Pi[S[t]]] \sim S[t]$$

This follows by construction by the definition of $\sim$, Π and S.

7 Example

The following program which represents the double append problem is by now well known [17]. Our example, however is slightly different than former presentations in that the types are explicitly represented, and the semantics are intended to be captured by the transition labels.

```
$List = \forall A.\nu Y.1 + (A \times Y)$
$\Delta = \{$
$(app,$
$\Lambda A.\lambda xs : List\ A.\lambda ys : List\ A.$
   case $\mathrm{out}(xs, 1 + A \times Y)$ of
      $\mathrm{inl}(u) \Rightarrow ys$
    ; $\mathrm{inr}(p) \Rightarrow \mathrm{in}(\mathrm{inr}(\mathrm{split}\ p\ \mathrm{as}\ x, xs'\ \mathrm{in}\ (x, app\ xs'\ ys)), Y)$
$)$
$\}$
```

$$\begin{tikzcd}
 & & & \mathit{app}\ A\ (\mathit{app}\ A\ \mathit{xs}\ \mathit{ys})\ \mathit{zs} \arrow[dl, dotted, "\delta \mathit{app}"] \\
 & & \cdots \arrow[d, dotted, "\delta \mathit{app}"] & \\
 & & \cdots \arrow[dl, "\mathit{xs} := \mathit{inl}(u)"'] \arrow[d, "\mathit{xs} := \mathit{inr}(p)"] & \\
 & \cdots \arrow[dl, "\mathit{ys} := \mathit{inl}(u)"'] \arrow[d, "\mathit{ys} := \mathit{inr}(p)"] & \cdots \arrow[d, "\text{fold}"] & \\
 \mathit{zs} & \mathit{app}\ \mathit{ys}\ \mathit{zs} & \cdots \arrow[d, "\text{inr}"] & \\
 & & \cdots \arrow[d, "p := (x, \mathit{xs}')"] & \\
 & & \text{split}\ p\ \mathit{as}\ (x, \mathit{xs}')\text{in}\ \cdots \arrow[d, "\text{fst}"] \arrow[dr, "\text{snd}"] & \\
 & & x & \mathit{app}\ A\ (\mathit{app}\ A\ (\mathit{xs}'\ \mathit{ys}))\ \mathit{zs} \arrow[uuuuuuu, dotted, "\theta \equiv (\mathit{xs}', \mathit{xs})"']
\end{tikzcd}$$

Fig. 2: Double Append

Now, we wish to find the map Δ and term for $\Pi[\Xi[t]]$ where

$$t = app\ A\ (app\ A\ xs\ ys)\ zs.\Xi[t]$$

This yields the transition system given in Fig. 2, with the following program yielded by $\Pi[\Xi[t]]$.

```
t = (appapp xs ys zs) Δ = {
 (appapp,
 ΛA.λxs : List A.λys : List A.λzs : List A.
     case out(xs, List A) of
        inl(u) ⇒ app ys zs
      ; inr(p) ⇒ case out(ys, List A) of
                    inl(u) ⇒ zs
                  ; inr(p) ⇒ in(inr(split p as x,xs′ in (x,appapp xs′ ys zs)), Y)
 ),
 (app,
 ΛA.λxs : List A.λys : List A.
     case out(xs, List A) of
        inl(u) ⇒ ys
      ; inr(p) ⇒ in(inr(split p as x, xs′ in (x, app xs′ ys)), Y)
 )
 }
```

Here we can see that $S[\Pi[S[t]]] \sim S[t]$ by inspection, a feature that follows from the idempotence of the composition $S \circ \Pi$.

8 Conclusion and Related Work

We have demonstrated a parallel between NbE and supercompilation for a System F with sums, products and recursive types. The intent is to clarify exactly in what way it can serve as a form of normalisation. In addition we have used bisimilarity of transition systems, which we use as the semantic domain, to show the correctness of our transformations. This simplifies the presentation, but also allows us to think more generally about term equivalence and the establishment of correct program transformations.

This work also makes a first step to including languages with polymorphic type systems directly into the supercompilation procedure while including type information. As we can see from the generalisation procedure, type information can not be ignored in the context of polymorphic types as it has a bearing on the form that generalisations will take and when generalisation can occur. If supercompilation is to be applied in the context of languages such as the Calculus of Constructions this will be even more critical.

Normalisation by Evaluation for a simple type theory is presented in [2]. A similar system defined for System F is given by Abel in [1]. Our approach differs in that we introduce a (potentially unsound) type theory based on System F

which uses transition systems as the semantic domain. Our system does not produce true normal forms as these NbE systems do, but is schematically quite similar.

Supercompilation was first described by Turchin [22]. Turchin is the first to recognise program configurations as representing some number of (potentially infinite) states, and process trees as being representations of potentially infinite traces. The system we use here is modeled on a description of positive supercompilation given by Sørensen, Glück and Jones [19].

The similarity of NbE and supercompilation have been described by Lisitsa and Webster [11] as well as Romanenko and Kluchnikov [10].

This work differs in spelling out the correspondence more completely. In addition it includes explicit type information. Instead of using the traditional process tree or partial process tree, we present transition systems as a semantic domain. This serves much the same purpose as a process tree; however the presentation varies slightly in that it provides us with a direct means of showing equivalence in the semantic domain. This is used to motivate the notion of our reflection operator.

In addition, to make explicit the connections with NbE it is important that supercompilation is done on terms with type information included. This paper is a preliminary step in this direction. In developing the use of supercompilation for constructive type theories [6], this will be particularly important.

In future work we hope to describe conditions which ensure that the type system is sound with respect to the function Δ. In addition it would be useful to extend the system to System F_ω to bring it closer to being of direct use for the Haskell Core which uses a variant of System F_ω [21].

It is also conceivable that normal forms of transition systems actually *do* exist in the context of constructive type theories with infinite terms, at least since the languages are necessarily sub-Turing complete and no simple application of the Full Employment Theorem can be made. It would be of value to explore this potentiality.

In the presentation given here, iso-recursive infinite types are given using explicit constructors representing the isomorphism. This may not be strictly necessary as it is possible to encode least and greatest fixed points directly in System F using universal quantification and impredicativity [28]. Indeed, the introduction and elimination rules for sums and products can also be encoded leaving only β, τ-reduction. While, for efficiency reasons and convenience, it is useful to have these constructors and destructors represented explicitly, it would be interesting to see how an exposition without them compares.

It would also be useful to include η-normalisation and variants which include function-symbol application, into the scheme, which has not been dealt with in this work. The inclusion of η would increase the number of programs which could find a *normal form* for the purpose of deducing equality [10].

Acknowledgements. This work is supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero, the Irish Software Engineering Research Centre.

References

1. Andreas Abel. Typed applicative structures and normalization by evaluation for system f^ω. In Erich Grädel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2009.
2. T. Altenkirch, P. Dybjer, M. Hofmannz, and P. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 303, Washington, DC, USA, 2001. IEEE Computer Society.
3. Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.
4. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier, MIT Press, 1990.
5. Volker Diekert and Yves Métivier. Partial commutation and traces. pages 457–533, 1997.
6. Eduardo Giménez. Structural recursive definitions in type theory. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 397–408, London, UK, 1998. Springer-Verlag.
7. Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999.
8. C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.
9. G. Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.
10. Ilya Klyuchnikov and Sergei Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947, pages 193–205. Springer, 2009.
11. Alexei Lisitsa and Matt Webster. Supercompilation for equivalence testing in metamorphic computer viruses detection. In *Proceedings of First International Workshop on Metacomputation in Russia, META'2008*, pages 113–118, 2009.
12. R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
13. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.
14. Y. Onoue, Z. Hu, H. Iwasaki, and M Takeichi. A calculational fusion system hylo. In *Algorithmic Languages and Calculi*, pages 76–106. Chapman and Hall, 1997.
15. Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*, pages 74–85. IEEE Computer Society Press, July 1991.
16. G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
17. Morten H. Sørensen and Robert Glück. Introduction to supercompilation. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, June 1998.
18. Morten Heine Sørensen. Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen, 1994. DIKU-rapport 94/17.
19. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

20. Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA, 2006.
21. Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM.
22. Valentin F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–548. Elsevier Science Publishers B.V. (North–Holland), 1988.
23. V.F. Turchin. The language refal–the theory of compilation and metasystem analysis. Technical Report 18, Curant Institute of Mathematics, 1980.
24. V.F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. *Lecture Notes in Computer Science*, 85:645–657, 1980.
25. V.F. Turchin. The Algorithm of Generalization in the Supercompiler. In *Proceedings of the IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pages 531–549, 1988.
26. D. A. Turner. Total functional programming. *Journal of Universal Computer Science*, 10(7):751–768, 2004.
27. P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–248, 1990.
28. Philip Wadler. Recursive types for free!, 1990.

A Method of Verification of Functional Programs Based on Graph Models

Andrew M. Mironov

Moscow State University, Russia

Abstract. In the paper we introduce a concept of a graph model of a functional program. We show how to use this model for verification of functional programs.

Keywords: functional programs, graph model, verification.

1 Introduction

1.1 The problem of verification of programs

The problem of verification of programs has the following form:

– given a program P and its formal specification S,
– prove that $P \models S$ (i.e. P meets S).

In this paper we consider the following special case of this problem:

– P is a functional program (FP)
– S is another FP, and
– $P \models S$ means that functions computed by P and S are equal.

For example, S is a description of a required function, and P is an implementation of this function.

1.2 Comparison with related works

The problem of verification of FPs is considered in many works (see, for example, [4]–[9]). The main methods of verification of FPs are computational induction and structural induction (a bried description of these methods can be found in [4], ch. 5). Verification of FPs with use of these methods is based on construction of logical assertions related to verified FPs, which are called *admissible predicates*. Because there is no a general algorithm of construction of such predicates, then the most realistic way of verification of FPs based on these methods is an elaboration of interactive procedures of synthesis of corresponding admissible predicates.

The main advantage of verification method of FPs proposed in the present paper in comparison with other works is the following. If verified FPs are represented by graph models, then the problem of construction of a corresponding admissible predicate can be decomposed to the problem of construction of logical assertions related to some pairs of nodes of these graph models, and a complexity of these assertions can be essentially less than a complexity of the admissible predicate.

2 A notion of a functional program

In this paper we consider a simplified notion of a FP. Without loss of generality, we assume that FPs under consideration does not contain nested recursions.

For precise definition of a notion of a FP we need in auxiliary notions described below.

2.1 Variables, constants, functional symbols

We assume that there are given

- a set $\mathcal{D}$ of **values**, which contains all natural numbers
- a set Con of **constants**, and every constant $c \in Con$ is associated with a value which is denoted by the same symbol c
- a set Var of **variables**, and every variable $x \in Var$ is associated with a set $\mathcal{D}_x \subseteq \mathcal{D}$ of values, which can be substituted instead of this variable
- a set Fun of **function symbols (FSs)**, and every $f \in Fun$ is associated with an integer number $ar(f) \geq 1$, which is called an **arity** of the FS f.

Some of FSs are called **primary** FSs. Every primary FS is associated with a partial function, which is denoted by the same symbol f, and has the form

$$f : \mathcal{D}^{ar(f)} \to \mathcal{D}$$

We assume that the set `Fun` contains the following FSs.

- FS `if_then_else` of the arity 3.
  A corresponding function maps every triple
  $$(d_1, d_2, d_3) \in \mathcal{D}^3$$
  - to d_2, if $d_1 = 1$, and
  - to d_3, otherwise.

  For every triple $(d_1, d_2, d_3) \in \mathcal{D}^3$ the expression
  $$\texttt{if\_then\_else}\ (d_1, d_2, d_3)$$
  will be denoted briefly as
  $$d_1 \ ?\ d_2 : d_3$$
- FS `eq` of arity 2. A corresponding function maps every pair (d_1, d_2)
  - to 1, if $d_1 = d_2$, and
  - to 0, otherwise.

  For every pair $(d_1, d_2) \in \mathcal{D}^2$ the expression `eq` (d_1, d_2) will be denoted briefly as $d_1 = d_2$.

Below the symbol Fun^+ denotes a set of all FSs which differ from `if_then_else`.

2.2 Expressions, substitutions, evaluations

Expressions are constructed from variables, constants and FSs. The set of all expressions is defined as follows.

1. Every variable and every constant is an expression.
2. For
   - every list of expressions $e_1, \dots, e_n$, and
   - every FS f of arity n

   the string
   $$f(e_1, \dots, e_n)$$
   is an expression.

A **substitution** is a string θ of the form

$$\theta \;=\; [x_1 := e_1, \dots, x_k := e_k] \tag{1}$$

where

- $x_1, \dots, x_k$ is a list of distinct variables, and
- $e_1, \dots, e_k$ are expressions.

For every substitution θ of the form (1), and every expression e the string θe denotes the result of replacement in e every variable x_i with corresponding expression e_i.

An **evaluation** is a partial function of the form

$$\xi : Var \to \mathcal{D}$$

For every pair ξ_1, ξ_2 of evaluations, and every variable $x \in Var$, the string

$$\xi_1(x) = \xi_2(x)$$

denotes the fact that values $\xi_1(x)$ and $\xi_2(x)$ either are both undefined, or are both defined and equal.

For every expression e a **value** $\xi(e)$ of e on evaluation ξ is defined iff one of the following conditions holds.

1. $e = x \in Var$, and $\xi(x)$ is defined. In this case $\xi(e) \overset{\text{def}}{=} \xi(x)$.
2. $e = c \in Con$. In this case $\xi(e) \overset{\text{def}}{=} c$.
3. e has the form $e_1 \; ? \; e_2 \; : \; e_3$, and $\xi(e_1)$ is defined. In this case
   - if $\xi(e_1) = 1$, then $\xi(e) \overset{\text{def}}{=} \xi(e_2)$, i.e.
     - either $\xi(e)$ and $\xi(e_2)$ are both undefined ,
     - or $\xi(e)$ and $\xi(e_2)$ are both defined and equal,
   - if $\xi(e_1) \neq 1$, then $\xi(e) \overset{\text{def}}{=} \xi(e_3)$
4. e has the form $f(e_1, \dots, e_k)$, and
   - $f \in Fun^+$
   - FS f is associated with a partial function
   - values $\xi(e_1), \dots, \xi(e_k)$ are defined,
   - a value of function f on the tuple $(\xi(e_1), \dots, \xi(e_k))$ is defined.

   In this case
   $$\xi(e) \overset{\text{def}}{=} f(\xi(e_1), \dots, \xi(e_k))$$

2.3 Functional programs

A **functional program (FP)** is a system of equations of the form

$$\begin{cases} f_1(x_{11}, \ldots, x_{1k_1}) = e_1 \\ \ldots \\ f_n(x_{n1}, \ldots, x_{nk_n}) = e_n \end{cases} \tag{2}$$

where

- $f_1, \ldots, f_n$ are distinct FSs, which are names of **functions, defined by this FP**,
  (all the FSs $f_1, \ldots, f_n$ are not primary FSs)
- $x_{11}, \ldots, x_{nk_n}$ are distinct variables, which are formal parameters of the defined functions, and
- $e_1, \ldots, e_n$ are expressions with the following properties: for every $i = 1, \ldots, n$
  - the set of all variables occurring in e_i, is equal to the set
    $$\{x_{i1}, \ldots, x_{ik_i}\}$$
  - every FS occurred in e_i,
    - either is a primary FS,
    - or belongs to the set $\{f_1, \ldots, f_n\}$.
  - every subexpression of e_i of the form $f_j(\ldots)$ contains only one non-primary FS (i.e. a FP does not contain nested recursions).

2.4 Functions defined by FPs

Functions defined by FPs are computed by a standard recursion: if a function f_i is defined by system (2), then its value on the tuple $(d_1, \ldots, d_{k_i})$ is equal to a value of expression

$$[x_{i1} := d_1, \ldots, x_{ik_i} := d_{k_i}]e_i \tag{3}$$

A value of expression (3) is computed as follows.

- If (3) is a constant, then its value is equal to this constant.
- If (3) has the form
  $$u_1 \ ? \ u_2 \ : \ u_3 \tag{4}$$
  then at first a value of u_1 is computed, and
  - if a value of u_1 is undefined, then a value of (4) is undefined,
  - if a value of u_1 is equal to 1, then a value of u_2 is computed, and
    - if a value of u_2 is undefined, then a value of (4) also is undefined,
    - otherwise a value of u_2 is equal by definition to a value of (4)
  - if a value of u_1 is not equal to 1, then a value of u_3 is computed, and
    - if a value of u_3 is undefined, then a value of (4) is undefined,
    - otherwise a value of u_3 is equal by definition to a value of (4)

- If (3) has the form
$$f(u_1, \dots, u_m) \tag{5}$$
where f is a primary FS, which is not equal to `if_then_else`, then values of $u_1, \dots, u_m$ are computed, and after this a value of function f on the tuple of values of $u_1, \dots, u_m$ is computed. By definition, this value is equal to a value of (5).
If
  - either one of values of $u_1, \dots, u_m$ is undefined,
  - or a value of f on a tuple of values of u_1, …, u_m is undefined

  then a value of (5) is undefined.
- If (3) has the form
$$f_j(u_1, \dots, u_{k_j})$$
where f_j is a FS, which is a name of a function defined by system (2), then its value is equal to a value of expression
$$[x_{j1} := u_1, \dots, x_{jk_j} := u_{k_j}]e_j \tag{6}$$
which is computed by the same way as a value of (3) is computed.

If the above process is not terminated, then a value of f_i on the tuple $(d_1, \dots, d_{k_i})$ is undefined.

3 Graph models of functional programs

3.1 Graph models of FPs

For every FP (2) we can construct a graph which is called **a graph model (GM)** of FP (2), in which

- every node has a label which is equal to some subexpression of one of the expressions $e_1, \dots, e_n$, and
- every edge has a label of one of the following types:
  1. **a condition**: a label of this type has the form
  $$\varphi\,? \quad \text{or} \quad \neg\varphi\,?$$
  where φ
     - either is some subexpression of one of expressions from (2),
     - or is constant 1.
  2. **a substitution**: a label of this type has the form
  $$[x_1 := u_1, \dots, x_m := u_m]$$
  where
     - $x_1, \dots, x_m$ are distinct variables which have occurrences in (2), and
     - $u_1, \dots, u_m$ are some subexpressions of one of expressions from (2).

3. **a link to a subexpression**: a label of this type has the form

$$(i)$$

and has the following meaning: for every edge with a label (i)
   - label of a start node of this edge has the form $f(u_1, \dots, u_m)$, and
   - label of an end node of this edge is u_i.

GM of FP (2) is defined as follows:

– A set of its nodes is equal to the set of subexpressions of expressions from (2). A label of every node is equal to the corresponding subexpression.
– For every equation $f_i(x_{i1}, \dots, x_{ik_i}) = e_i$ from (2), GM has an edge with label 1? from a node with the label $f_i(x_{i1}, \dots, x_{ik_i})$ to a node with the label e_i.
– For every node N with a label of the form

$$e_1 \; ? \; e_2 \; : \; e_3$$

GM contains
   - an edge from N to a node with the label e_2, and a label of this edge is $e_1?$, and
   - an edge from N to a node with the label e_3, and a label of this edge is $\neg e_1?$.

– For every node N with a label of the form

$$f_i(u_1, \dots, u_{k_i})$$

where f_i is a name of some function which is defined by FP (2), GM contains an edge from N to a node with the label $f_i(x_{i1}, \dots, x_{ik_i})$, and label of this edge has the form

$$[x_{i1} := u_1, \quad \dots, \quad x_{ik_i} := u_{k_i}]$$

– For
   - every node N with a label of the form

$$f(u_1, \dots, u_m) \tag{7}$$

   where $f \in Fun^+$, and
   - every $i = 1, \dots, m$

GM contains an edge from N to a node with the label u_i, and a label of this edge is (i).

3.2 Simplification of GMs

GMs of FPs can be transformed to equivalent (in certain sense) GMs. The transformations consists of a removing of edges and nodes, as following. Let a GM has a node N, such that a label of every edge incoming in N and every edge outgoing from N has a type "a condition". In this case

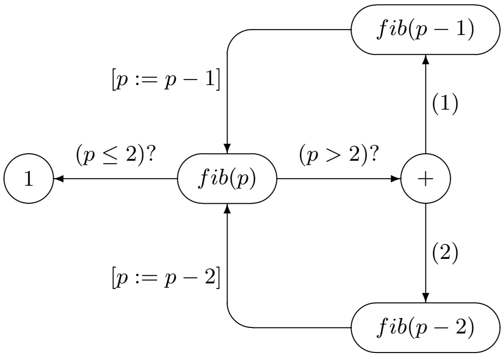


Fig. 1: A simplified GM of FP fib.

– node N can be removed, and
– every pair of edges of the form

$$N_1 \xrightarrow{\varphi?} N, \quad N \xrightarrow{\varphi_2?} N_2$$

can be changed on one edge of the form

$$N_1 \xrightarrow{(\varphi_1 \wedge \varphi_2)?} N_2$$

This operation can be made several times.

Also it is possible

– to change labels of nodes of the form (7), where f is a primary FS, on labels of the form f
(i.e. not write a list of arguments after symbol f)
– to remove every edge with a label of the form (i), and to remove a node which is an end of this edge.

A GM, which is a result of such transformations, is called **a simplification** of an original GM.

3.3 Examples of GMs of FPs

Consider a FP, which defines a Fibonacci function

$$fib(p) = (p \leq 2)?1 : fib(p-1) + fib(p-2)$$

where the set of values, associated with variable p, is the set of natural numbers $(1, 2, \ldots)$.

A simplified GM, which corresponds to this FP, has the form depicted in the figure 1.

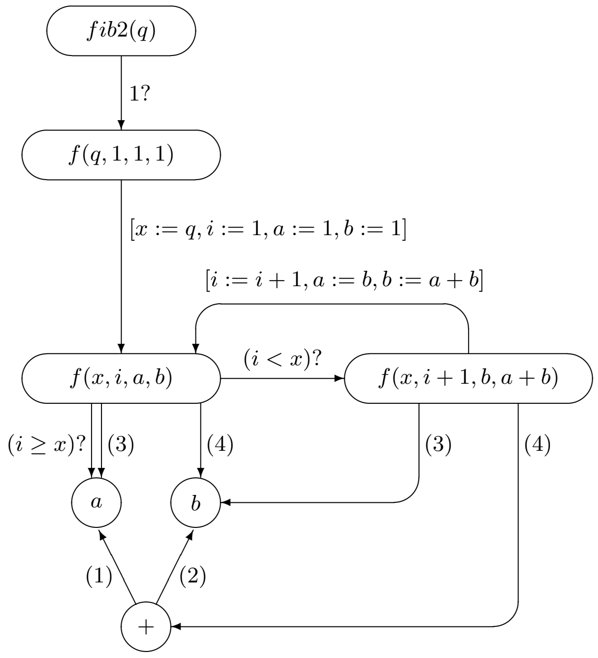


Fig. 2: A simplified GM of a FP, which defines function fib2.

Consider other FP, which also defines a Fibonacci function, but computes it by the method "from bottom to top" (the previous FP computes Fibonacci function by the method "from top to bottom"):

$$fib2(q) = f(q,1,1,1)$$
$$f(x,i,a,b) = (i<x)?f(x,i+1,b,a+b) : a$$

where all variables range over the set of integer numbers.

A simplified GM, which corresponds to this FP, has the form, which is depicted in figure 2.

4 A method of proving of equality of functions defined by FPs

Assume that two FP are given, and

- first FP defines functions $f_1,\dots,f_n$, and
- second FP defines functions $g_1,\dots,g_m$.

If we would like to prove that functions f_1 and g_1, which are defined by these FPs, are equal (under a condition that arguments of these functions satisfy certain relation which is called a **precondition**), then it is possible to prove as follows.

At first we rename variables in these FPs, such that every variable has an occurrence in only one equation in these FPs. Second, we construct GMs of these FPs.

Theorem.

Let there is a set of arcs, which connect nodes of first GM with nodes of second GM, and satisfy the following conditions.

1. *Every arc has a label, which is a boolean-valued expression.*
2. *There is an arc,*
   - *which connects a node in first GM with a label of the form $f_1(x_{11}, \ldots, x_{1n_1})$, with a node in second GM with a label of the form $g_1(y_{11}, \ldots, y_{1m_1})$, and*
   - *a label of which is equal to a precondition.*
3. *If*
   - *one GM has an edge with label θ from a node v_1 to a node v_1',*
   - *v_2 is a node of other GM*
   - *node v_1 is connected with a node v_2 by an arc with a label α,*

   then node v_1' is connected with a node v_2 by an arc with a label α', such that the following implication holds:

$$\alpha \to \theta\alpha'$$

4. *If*
   - *one GM has an edge with a label $\varphi?$ from a node v_1 to a node v_1',*
   - *v_2 is a node of other GM*
   - *node v_1 is connected with node v_2 by an arc with label α,*

   then node v_1' is connected with node v_2 by an arc with label α', such that the following implication holds:

$$(\alpha \wedge \varphi) \to \alpha'$$

5. *If*
   - *node v_1 of one GM is connected with a node v_2 of other GM by an arc with a label α,*
   - *v_1 and v_2 have no outgoing edges*
   - *labels of v_1 and v_2 are expressions e_1 and e_2 respectively*

   then the following implication holds:

$$\alpha \to (e_1 = e_2) \tag{8}$$

6. *If*
   - *node v_1 of one GM is connected with a node v_2 of other GM by an arc with a label α,*
   - *label v_1 is a primary FS*

then in this case

- *label v_2 is the same FS,*
- *let*
  - *a set of ends of edges outgoing from v_1 has the form*

    $$\{v_{11}, \ldots, v_{1n}\}$$

    and for every $i = 1, \ldots, n$ node v_1 is connected with node v_{1i} by an edge with a label (i)
  - *a set of ends of edges outgoing from v_2 has the form*

    $$\{v_{21}, \ldots, v_{2n}\}$$

    and for every $i = 1, \ldots, n$ node v_2 is connected with node v_{2i} by an edge with a label (i),

  then for every $i = 1, \ldots, n$ node v_{1i} is connected with a node v_{2i} by an arc with a label α_i, such that the following implication holds:

  $$\alpha \to (\alpha_1 \wedge \ldots \wedge \alpha_n)$$

7. *If*
   - *one GM has*
     - *an edge with a label θ from a node v_1 to a node v'_1,*
     - *there are edges outgoing from v_1 such that*
       - *labels of these edges are $(1), \ldots, (n)$, and*
       - *ends of these edges are nodes $v_{11}, \ldots, v_{1n}$ respectively,*
     - *there are edges outgoing from a node v'_1 such that*
       - *labels of these nodes are $(1), \ldots, (n)$, and*
       - *ends of these edges are $v'_{11}, \ldots, v'_{1n}$ respectively*
   - *v_2 is another node of second GM, such that for every $i = 1, \ldots, n$ there is an arc which connects v_2 with v_{1i}, and a label of this arc is α_i*

   then for every $i = 1, \ldots, n$ node v_2 is connected with a node v'_{1i} by an arc with a label α'_i, such that the following implication is hold:

   $$\alpha_i \to \theta\alpha'_i$$

Then, if both FPs terminate their computation on every tuple of values of arguments which satisfy a precondition, then values of functions f_1 and g_1 on these tuples of values of arguments are equal.

Proof.

If an arc with label α connects nodes with labels e_1 and e_2 respectively, then we shall interprete this arc as a proposition that the following implication holds:

$$\alpha \to (e_1 = e_2) \tag{9}$$

According to this interpretation, an arc which connects

– a node of first GM with a label of the form

$$f_1(x_{11}, \ldots, x_{1n_1})$$

and

– a node of second GM with a label of the form

$$g_1(y_{11}, \ldots, y_{1m_1})$$

(such arc exists by assumption 2 of a condition of the theorem)

represents a goal proposition.

For proving that goal proposition is true for every evaluation ξ_0, we

– assume that a condition of this implication holds on this evaluation, and
– prove that a conclusion of this implication also holds on this evaluation.

In this case truth of a condition of implication (9) means that evaluation ξ_0 is a list of values of arguments of the FP, which satisfies a precondition. By assumption, functioning of every FP on this list terminates. This functioning can be represented as a finite tree,

– nodes of which correspond to nodes of GM (several nodes of the tree can correspond to one node of the GM), and are labeled by expressions of the form ξe, where
  - e is an expression, which is a label of a corresponding node GM, and
  - ξ is a substitution, which associates every variable from e with a value.
– edges of which correspond to edges of the GM, and have the same labels as corresponded edges of the GM.

Roots of these trees correspond to nodes of the GMs with labels $f_1(x_{11}, \ldots, x_{1n_1})$ and $g_1(y_{11}, \ldots, y_{1m_1})$.

If

– node w_1 of first tree corresponds to node v_1 of first GM,
– node w_2 of second tree corresponds to node v_2 of second GM, and
– among arcs mentioned in the theorem, there is an arc which connects v_1 and v_2 with a label α,

then we draw an arc with the same label α, which connects w_1 and w_2.

The condition of the theorem and a definition of functioning of a FP imply the following propositions.

– If
  - there is an arc with a label α, which connects a node of first tree with a node of other tree, and
  - labels of these nodes have the form $\xi_1 e_1$ and $\xi_2 e_2$ respectively

then expression α holds on evaluation

$$\xi_1 \cup \xi_2$$

which is equal to ξ_1 on variables occurring in e_1, and to ξ_2 on variables occurring in e_2 (this can be proved by induction, where a base of induction is an assumption that the precondition holds on evaluation ξ_0, and inductive steps can be founded by the corresponding assumptions in the theorem).
– Every terminal node of one of the trees is connected by an arc with a terminal node of other tree.

Because every arc, which connects terminal nodes, satisfies condition (8), in which

– α is a label of this arc, and
– e_1 and e_2 are labels of nodes of a GM, which corresponds to these terminal nodes of the trees

then (8) implies the implication

$$(\xi_1 \cup \xi_2)\alpha \to (\xi_1 e_1 = \xi_2 e_2)$$

where $\xi_1 e_1$ and $\xi_2 e_2$ are labels of the above terminal nodes. As it was stated above, expression α holds on the evaluation $\xi_1 \cup \xi_2$. Consecuently, values of the expressions $\xi_1 e_1$ and $\xi_2 e_2$ (which are labels of terminal nodes) are equal.

Moving from terminal nodes to root nodes, we can use the above propostions for proving that for every pair of nodes which are connected by an arc, values of expressions, which are labels of these nodes, are equal.

Because root nodes of these trees also are connected by an arc, then values of expressions, which are labels of these nodes, are equal. ■

The proposed method can be modified in such a manner that for proving a equality of functions defined by different FPs it is necessary to define not all arcs mentioned in the theorem, but only several of them (omitted arcs can be generated automatically). We shall not describe in detail the modified method. We only note that for proving of equality of functions which are defined by FPs in section 3.3, it is enough to define only the following arcs:

1. an edge with a label $p = q$, which connects
   – a node with a label $fib(p)$ of first GM, and
   – a node with a label $fib2(q)$ of second GM
2. an edge with a label $(p = x) \wedge (i \leq x)$, which connects
   – a node with a label $fib(p)$ of first GM, and
   – a node with a label $f(x, i, a, b)$ of second GM
3. an edge with a label $p = i$, which connects
   – a node with a label $fib(p)$ of first GM, and
   – a node with a label a of second GM
4. an edge with a label $p = i + 1$, which connects
   – a node with a label $fib(p)$ of first GM, and
   – a node with a label b of second GM.

References

1. K.G. van den Berg, P.M. van den Broek: Static analysis of functional programs. Information and Software Technology, 1995, Volume 37, Number 4, Pages 213-224.
2. A.Ireland, A.Bundy: Automatic verification of functions with accumulating parameters. Journal of Functional Programming, 9(2):225–245, March 1999.
3. J. Loeckx and K. Sieber: The Foundations of Program Verification. Teubner, second edition, 1987.
4. Z. Manna: Mathematical Theory of Computation. McGraw-Hill Inc., 1974.
5. Manna, Z., Ness, S., and Vuillemin, J.: Inductive methods for proving properties of programs. Commun. ACM 16, 8 (Aug. 1973), 491–502.
6. Z. Manna and J. McCarthy: Properties of Programs and Partial Function Logic. Machine Intelligence, 5:27–37.
7. Z. Manna and A. Pnueli: Formalization of Properties of Functional Programs. J. ACM, 17(3):555–569, 1970.
8. N. Popov and T. Jebelean: Using Computer Algebra Techniques for the Specification, Verification and Synthesis of Recursive Programs. Mathematics and Computers in Simulation, pages 1–13, 2007.
9. N. Popov and T. Jebelean: A Prototype Environment for Verification of Recursive Programs. In Z. Istenes, editor, Proceedings of FORMED08, pages 121–130, March 2008.

Supercompilation and the Reduceron

Jason S. Reich, Matthew Naylor and Colin Runciman

Department of Computer Science, University of York
{jason,mfn,colin}@cs.york.ac.uk

Abstract. This paper explores some of the performance-enhancing features of supercompilation in the context of the Reduceron — a special-purpose graph-reduction machine. Two small examples are discussed in detail, highlighting areas where the two technologies interact. A strategy is introduced for countering a situation where supercompilation adversely affects Reduceron execution time. Performance results and other metrics are presented across a range of nineteen benchmarks highlighting the synergistic properties of supercompilation on the Reduceron. This paper represents work in progress.

1 Introduction

Functional programming is a distinctive paradigm that has scope for exploiting non-standard technologies at every stage of computation. Supercompilation and the Reduceron are two such technologies.

Supercompilation [1,2] is a metaprogramming technique that, at compile-time, evaluates (drives) programs until an unknown is required and then proceeds by case analysis (residuates). Among other benefits, it can remove intermediate data structures and specialise higher order functions, with corresponding performance gains at execution time.

The Reduceron is an FPGA-based soft processor for executing lazy functional programs by graph reduction [3,4]. The special-purpose processor can perform in parallel many of the steps required for each reduction, whereas conventional architectures need to perform these steps serially.

Does a combination of these technologies lead to further improvements in performance? Are these techniques conflicting, compatible or even mutually beneficial? In this paper, we discuss how the two may interact and present preliminary findings from a prototype supercompiler for the Reduceron source language.

2 Our Source Language

Our source language [5] is close to subsets of both Haskell 98 [6] and Clean [7]. It supports algebraic data types, uniform pattern matching by construction, local variable bindings, and various primitive integer operations.

$$prog := \overline{f\ \overline{vs} = x}$$

$$\begin{aligned} exp := {} & v\ (\mathit{variables}) \\ & \mid c\ (\mathit{constructors}) \\ & \mid f\ (\mathit{functions}) \\ & \mid n\ (\mathit{integers}) \\ & \mid x\ \overline{xs}\ (\mathit{applications}) \\ & \mid \textbf{case}\ x\ \textbf{of}\ \overline{c\ \overline{vs} \to x} \\ & \mid \textbf{let}\ \overline{v = x}\ \textbf{in}\ y \end{aligned}$$

Fig. 1. Abstract syntax for our source language.

Abstract syntax for our source language is given in Figure 1. In addition to the annotated symbols, x and y range over expressions. Overlining and pluralisation indicate sequences of productions. For example $\overline{vs}$ represents a sequence of variable names. All programs contain a function named `main` of arity zero.

Listing 1 shows an encoding of a program in our source language. This program (somewhat inefficiently) doubles each element in a range, calculates the sum and prints the result.

The basic compiler (before the introduction of supercompilation) first reduces all pattern matching to combinations of one-level case distinctions. A case-elimination phase then translates algebraic data constructors and case expressions to functions and function applications respectively, using a variation of the Scott encoding, e.g. after case elimination our `range` function is as displayed in Listing 2. In every function body, a bottom-up traversal inlines saturated applications of non-primitive functions. Finally, the compiler generates a compact encoding of function body.

The encoding phase must take into account the design parameters of the Reduceron. Encoded forms of function bodies are constrained by limits on the size of the top-level spine, the number and size of nested applications, and the number of case-table arguments in an application. Encoded bodies are split if necessary, with the introduction of auxiliary combinators.

It should be stressed that the compiler *is not* generating circuitry for the FPGA. Rather, it is generating a representation of the program, suitable for execution on a template instantiation machine [8]. In this case, the template instantiation machine, the Reduceron, has been realised on an FPGA.

Listing 1. An example program, in our source language.

```
{
    foldl f z xs = case xs of {
        Nil        -> z;
        Cons y ys -> foldl f (f z y) ys;
    };

    map f xs = case xs of {
        Nil        -> Nil;
        Cons y ys -> Cons (f y) (map f ys);
    };

    plus x y = (+) x y;
    sum = foldl plus 0;

    double x = (+) x x;

    sumDouble xs = sum (map double xs);

    range x y = case (<=) x y of {
        True  -> Cons x (range ((+) x 1) y);
        False -> Nil;
    };

    main = emitInt (sumDouble (range 0 10000)) 0;
}
```

3 The Reduceron Architecture

The Reduceron features broad memory channels to 'widen the von Neumann bottleneck.' [3] Many of the operations required to perform each graph reduction step are simultaneously performed in a single clock cycle.

Instantiation of a function body takes $\lceil n/2 \rceil$ clock ticks, where n is the number of applications in the body. Establishing the environment for function applications, updating the heap-graph to prevent repeated evaluations and applying primitive functions each take just one clock cycle. Dynamically maintained sharing information allows the Reduceron to avoid a high proportion of redundant updates where no sharing can occur. Constructor reductions (selection of the appropriate case alternative function from a case table) take place in zero clock cycles.

Listing 3 shows the evaluation of `range 0 10` to head normal form. Each reduction step is annotated with the operation that is being performed and how many cycles are required. This example takes four clock cycles, under the scheme outlined so far.

Listing 2. `range` function after case elimination.

```
range x y = (<=) x y [range#1,range#2] x y;
range#1 alts x y = Nil;
range#2 alts x y = Cons x (range ((+) x 1) y);
```

Listing 3. Reduction of `range 0 10` without PRS.

```
  range 0 10
= { Instantiate function body (1 cycle) }
  (<=) 0 10 [range#1,range#2] 0 10
= { Primitive application (1 cycle) }
  True [range#1,range#2] 0 10
= { Constructor reduction (0 cycle) }
  range#2 [range#1,range#2] 0 10
= { Instantiate function body (2 cycles) }
  Cons 0 (range ((+) 0 1) 10)
```

4 Primitive Redex Speculation

If primitive applications in body have fully evaluated arguments at instantiation time, the Reduceron can evaluate them speculatively during instantiation. Primitive redexes need not be constructed in memory, nor fetched again when needed. Even if the result of a primitive redex is not needed, reducing it is no more costly than constructing it.

Once again consider the reduction of the expression `range 0 10`, now with primitive redex speculation (PRS) enabled (Listing 4). One clock cycle is avoided for the comparison. Further clock cycles will be saved if the tail of the result is needed, as the addition to form the lower bound of the range has also been speculatively evaluated.

The beneficial effect of PRS is quite marked. For the example program in Listing 1, the Reduceron takes 230,029 clock cycles to execute *without* PRS. With PRS enabled the Reduceron only takes 150,024 clock cycles, a 35% reduction.

However, the number of PRS reductions in each instantiation is limited by a Reduceron design parameter. Currently, this limit is two.

5 Benchmark Programs

A selection of nineteen programs are used to test and benchmark the Reduceron platform. These programs range from small, toy examples that demonstrate specific effects, such as the `sum`– series, to significant computations like `Knuthbendix`.

Listing 4. Reduction of `range 0 10` with PRS.

```
  range 0 10
= { Instantiate function body (1 cycle) }
  True [range#1,range#2] 0 10
= { Constructor reduction (0 cycle) }
  range#2 [range#1,range#2] 0 10
= { Instantiate function body (2 cycles) }
  Cons 0 (range 1 10)
```

These programs are described below in terms of their purpose, characteristics and code size. Line counts are for sources including all required auxiliary functions.

Adjoxo An adjudicator for the game noughts and crosses, a.k.a. tic-tac-toe. The input is a game position, and the output is one of the three values — Win, Draw or Loss — indicating the outcome with best play for each of the players whose turn it might be. The method is the usual minimax recursive evaluation of completed game trees. *(106 lines)*

Braun A Braun tree is a balanced binary tree offering an efficient yet simple implementation of flexible arrays. The program tests the property that converting a list to a Braun tree and back again is equivalent to the identity function. *(51 lines)*

Cichelli Finds a perfect hash function for Haskell keywords [9]. It uses a backtracking search to find an assignment of natural-number values to each letter that starts or ends a keyword such that hash values for keywords, computed as start-value + end-value + length, are unique and occupy a small integer range without gaps. *(200 lines)*

Clausify Puts propositional formulae in clausal form using a multi-stage transformation of formula-trees [9]. Almost a purely symbolic application, with hardly any arithmetic. *(131 lines)*

Fib Computes the Nth number in the fibonacci sequence using a simple but naive doubly-recursive function definition. A purely arithmetic program involving no data structures at all. *(10 lines)*

Knuthbendix The Knuth-Bendix completion method tries to derive a convergent term-rewriting system for a given equational theory and symbol-weighting scheme. It is a typical symbolic computing application from computer algebra. The example input used in the program gives group-theoretic axioms from which ten rewriting rules are derived. *(533 lines)*

MSS Computes the maximum segment sum of a list of integers. Works by dividing the input list into all sub-lists, computing the sum of each, and returning the maximum. *(47 lines)*

Mate Solves chess end-game problems of the form "P to move and mate in N" [9]. The method is brute-force search in an explicit AND-OR game tree developing the given position to depth $2N - 1$. Boards are represented by a square-piece assocation list for each player, where squares are coded as rank-file numeric pairs, so there is a fair amount of primitive arithmetic and comparison. *(393 lines)*

OrdList Checks the property that insertion of a number into an ordered list of numbers results in a list that is still ordered. Numbers are represented as Peano numerals, so this is a purely symbolic program. *(46 lines)*

Parts Computes a celebrated number-theoretic function, the number of partitions of n, where a partition is a bag of positive integers that sum to n. There is a sophisticated closed formula for this number, but the method here is to list and count partitions explicitly. *(54 lines)*

PermSort Enumerates the permutations of a list of numbers, and returns the first ordered permutation. *(39 lines)*

Queens Solves a programming problem made famous by Wirth: place N queens on an $N \times N$ chess board so that no two queens occupy a common rank, file or diagonal [9]. The solution involves backtracking, list processing and an inner recursive loop that tests the safety of each candidate position for a new queen by primitive arithmetic comparisons with the coded positions of queens already in place. *(47 lines)*

Queens2 A purely symbolic solution to the N-queens problem. Represents the board as a list of lists. Places a queen on one row at a time, maintaining a grid of threatened squares, and backtracks if a queen cannot be placed. *(62 lines)*

Sudoku A Sudoku solver due to Richard Bird [10]. Fills the blank cells on a Sudoku board with valid digits, pruning many possible choices that cannot possibly lead to a solution. *(209 lines)*

Taut A tautology checking program based on an example from Hutton's book. The method is a brute-force evaluation for all possible boolean assignments to variables. *(95 lines)*

While A structural operational semantics of Nielson and Nielson's While language [11] applied to a program that computes the number of divisors of given integer. *(96 lines)*

sumDouble Computes $\sum_{i=0}^{10000} 2i$ by generating the list of the numbers between 0 and 10,000, doubling each element and then computing the sum using the higher-order function, `foldl`. Contains intermediate data structures and primitive operations. The program in Listing 1. *(20 lines)*

sumSquares Computes $\sum_{i=0}^{100} i^2$. This is done in a similar fashion to the previous example. The square function consists of replicating its input n, n times and summing the result using `foldl`. Contains intermediate data structures, primitive operations and nested loops. *(23 lines)*

sumSumEnum Computes $\sum_{i=0}^{100} \sum_{j=0}^{i} j$ by generating the list of the numbers between 0 and 100, mapping a function, `sumEnum`, over the list and summing the resulting list. The `sumEnum` function sums the numbers between 0 and its input. Contains intermediate data structures, primitive operations and nested loops. *(22 lines)*

Listing 5. A supercompiled form of `sumDouble`.

```
sumDouble = sumDoubleAc 0;
sumDoubleAc z xs = xs [sumDoubleAc#1, sumDoubleAc#2] z;
sumDoubleAc#1 y ys alts z = sumDoubleAc ((+) z ((+) y y)) ys;
sumDoubleAc#2 alts z = z;
```

6 A Synergistic Effect of Supercompilation

The basic Reduceron compiler currently performs very little optimisation. We are developing a supercompiler targeted at the Reduceron platform. The starting point for our current prototype was a previous positive supercompiler for a core functional language by Mitchell [12].

Mitchell's design inserted a supercompilation phase between core generation and compilation by the optimising Glasgow Haskell Compiler (GHC). In all but one of the published benchmarks, Mitchell's supercompiler demonstrated at least equal and often significantly improved performance when compared with GHC alone.

One reason for the improvement is that supercompilation fuses away intermediate data structures. In its original form, the function `sumDouble` (Listing 1) maps `double` over its list input, only to apply `foldl plus` to the newly constructed list to calculate the sum. The supercompiler fuses this composition to a residual function that does not produce the intermediate list but performs the `double` operation as it sums. For both conventional implementations and the Reduceron, fewer reductions are needed to construct and deconstruct data structures.

Some effects of supercompilation particularly benefit the features of the Reduceron architecture. For example, the program in Listing 1 cannot, as it stands, benefit from PRS during the `sum` function because the primitive addition is not apparent in the body of the higher-order `foldl`. However, if supercompiled, `foldl plus` is specialised to a first-order equivalent. A considerable reduction in clock cycles is obtained because PRS now applies.

The original program evaluates 20,003 expressions by PRS, compared with the 29,995 for the supercompiled program. While it is possible in some cases, for this example, no more primitive reductions were performed overall than we performed originally. A further performance gain achieved on top of the fusion effects. Following supercompilation, the Reduceron takes 159,970 clock cycles to execute the program in Listing 1 without PRS and only 39,996 clock cycles with PRS. Compared with the original program executed without PRS, this is a 87% performance increase.

Listing 5 shows the combined effects of supercompilation fusion and specialisation on `sumDouble`. In `sumDoubleAc#1`, the dependency of the outer addition on the inner one means that PRS requires an extra clock cycle to evaluate the

Listing 6. Original and supercompiled piece safety in the n-queens problem.

```
and x y = case x of { True -> y; False -> False };

safe x d qs = case qs of {
  Nil      -> True;
  Cons q l ->
    and ((/=) x q) (
    and ((/=) x ((+) q d)) (
    and ((/=) x ((-) q d)) (
    safe x ((+) d 1) l)));
  };

safeSC x d qs = case qs of {
  Nil      -> True;
  Cons q l ->
    case (/=) x q of {
      True ->
         case (/=) x ((+) q d) of {
           True ->
             case (/=) x ((-) q d) of {
               True -> safeSC x ((+) d 1) l;
               False -> False
             };
           False -> False
        };
      False -> False
    }
  };
```

expression fully. However, overall cycles are still saved in comparison with the reduction of a separate function application.

7 A Potentially Obstructing Effect of Supercompilation

There are circumstances where the process of supercompilation might impede PRS. Consider Listing 6, an extract from the `Queens` example. The function `safe` computes whether it is 'safe' to place a queen in rank x, at a distance of d files away from the queens currently placed on the board. These queens are specified by their rank positions in the list qs.

Listing 7 shows the original and supercompiled definitions following case elimination and inlining. Notice that in `safe`, all of the primitive reducible expressions are in one case alternative. On the other hand, in the supercompiled version, `safeSC`, the expressions are split over separate case alternatives, and therefore, instantiations after Scott encoding.

Listing 7. Listing 6 after case elimination.

```
and v0 v1 = v0 [and#1,and#2] v1;
and#1 v0 v1 = False;
and#2 v0 v1 = v1;

safe v0 v1 v2 = v2 [safe#1,safe#2] v0 v1;
safe#1 v0 v1 v2 v3 v4 = let {
    v5 = (+) v4 1;
    v6 = (/=) v3 ((-) v0 v4);
    v7 = (/=) v3 ((+) v0 v4);
    v8 = (/=) v3 v0
  } in v8 [and#1,and#2]
        (v7 [and#1,and#2]
          (v6 [and#1,and#2]
            (v1 [safe#1,safe#2] v3 v5)));
safe#2 v0 v1 v2 = True;

safeSC v0 v1 v2 = v2 [safeSC#7,safeSC#8] v0 v1;
safeSC#1 v0 v1 v2 v3 = False;
safeSC#2 v0 v1 v2 v3
  = let { v4 = (+) v2 1 } in v3 [safeSC#7,safeSC#8] v1 v4;
safeSC#3 v0 v1 v2 v3 v4 = False;
safeSC#4 v0 v1 v2 v3 v4
  = (/=) v1 ((-) v2 v3) [safeSC#1,safeSC#2] v1 v3 v4;
safeSC#5 v0 v1 v2 v3 v4 = False;
safeSC#6 v0 v1 v2 v3 v4
  = (/=) v1 ((+) v2 v3) [safeSC#3,safeSC#4] v1 v2 v3 v4;
safeSC#7 v0 v1 v2 v3 v4
  = (/=) v3 v0 [safeSC#5,safeSC#6] v3 v0 v4 v1;
safeSC#8 v0 v1 v2 = True;
```

This leads to a situation where the execution of the original can speculatively evaluate a number of expressions simultaneously, whereas `safeSC` evaluates them separately at each function body instantiation.

To alleviate this issue, primitive expressions can be lifted as far as their variables are bound. The lifting process can take into account the maximum number of PRS reductions at instantiation and only lift to where there is spare capacity.

However, if we naively lift all primitive redex expressions, we may cause duplicate computation to occur. The supercompiler is permitted to duplicate *code* as long as it does not duplicate *computation*, under lazy evaluation. For example, our supercompiler may replicate bindings from outside a case expression down each case alternative. As only one alternative is evaluated, only one of the duplicate bindings will be evaluated under both lazy and speculative evaluation.

	Original		Supercompiled		SC + PRS lift	
	a	*b*	*c*	*d*	*e*	*f*
	(No PRS)	(PRS)	(No PRS)	(PRS)	(No PRS)	(PRS)
`Adjoxo`	1.000	0.799	0.866	0.671	0.707	**0.405**
`Braun`	1.000	1.000	**0.769**	**0.769**	**0.769**	**0.769**
`Chichelli`	1.000	**0.990**	1.000	**0.990**	1.013	0.998
`Clausify`	1.000	1.000	1.050	1.050	1.051	**0.951**
`Fib`	1.000	0.445	1.000	0.445	0.907	**0.353**
`KnuthBendix`	1.000	0.900	0.896	0.833	0.876	**0.779**
`MSS`	1.000	0.864	0.995	**0.858**	0.997	0.861
`Mate`	1.000	0.867	0.912	0.838	0.916	**0.827**
`OrdList`	1.000	1.000	**0.662**	**0.662**	0.678	0.678
`Parts`	1.000	0.746	0.933	**0.679**	1.029	0.753
`PermSort`	1.000	0.962	0.861	0.861	0.759	**0.727**
`Queens`	1.000	0.421	0.850	0.489	0.811	**0.325**
`Queens2`	1.000	0.996	0.989	0.985	0.966	**0.961**
`Sudoku`	1.000	0.936	0.955	0.892	0.922	**0.815**
`Taut`	1.000	1.004	**0.700**	**0.700**	0.944	0.859
`While`	1.000	0.947	0.996	**0.942**	1.047	1.005
`sumDouble`	1.000	0.652	0.695	0.174	0.739	**0.130**
`sumSquares`	1.000	0.541	0.726	0.206	0.793	**0.205**
`sumSumEnum`	1.000	0.481	0.847	0.455	0.960	**0.454**
Geometric Mean	*1.000*	*0.788*	*0.871*	*0.647*	*0.881*	***0.598***

Table 1. Execution time as multiples of that for pipeline *a*, the program before supercompilation executed without PRS. *(Best results are in bold.)*

However, if replicated primitive redexes are lifted above a case distinction, they may be evaluated speculatively, taking away capacity that other primitive expressions could have used. A solution is to detect these replicated expressions and merge them into a single binding.

Our original supercompiler worsened PRS-enabled results for `Queens` by 16%. With the primitive redex lifting strategy applied, supercompilation improves results by 33%.

8 Performance Results

8.1 Compared to without PRS and Supercompilation

Each example described in Section 5, is compiled with six variations of compilation pipeline. These are; *a*) normal compilation, *b*) normal compilation with PRS, *c*) supercompilation and normal compilation, *d*) supercompilation and normal compilation with PRS, *e*) supercompilation, primitive redex lift and normal compilation, *f*) supercompilation, primitive redex lift and normal compilation with PRS.

The compiled output is executed on a Reduceron simulator. The simulator returns various profiling measurements such as total clock cycles, the number of PRS evaluated expressions and the proportion of time spent on individual functions and reduction operations. Table 1 presents the performance of our test programs relative to that of pipeline *a*, the original program executed without PRS.

PRS Only — Primitive redex speculation (pipeline *b*) shows an average performance increase of 21%. All but four of the examples achieve some improvement. The only example that suffers under PRS is `Taut`, likely due to the structure of the program. The drop in performance is only very slight, however.

Supercompilation Only — Supercompilation (pipeline *c*) shows an average performance boost of 13%. Only three examples do not benefit from supercompilation. While we do not expect much improvement on `Fib` due to its simple structure, the highly symbolic programs `Chichelli` and `Clausify` might have shown some fusion.

Supercompilation and PRS — The combination of PRS and supercompilation (pipeline *d*) largely gives results as expected (PRS factor × supercompilation factor) with a few notable exceptions. `Adjoxo`, `Parts`, `sumDouble` and `sumSquares` all show better than expected performance, mainly for the reasons described in Section 6. However, `Queens` and `sumSumEnum` show considerably worse than expected performance, likely for the reasons outlined in Section 7.

Supercompilation, Lifting and PRS — The results of PRS, supercompilation *and* primitive redex lifting (pipeline *f*) indicate that this strategy is effective. Every program except for `While` performs better than the original program without PRS. However, `Taut` performs significantly worse than under our original supercompiler strategy. Across all our test programs, this strategy gives an average performance boost of 40%.

8.2 Compared to without Supercompilation on a PRS-enabled Platform

Table 2 gives another view on the impact of the supercompiler. It compares the number of Reduceron combinators produced (size), time taken to execute (cycles), number of case tables evaluated (cases) and the proportion of primitive operations performed by PRS. These are recorded for pipeline *f* and shown as multiples of the results for pipeline *b*, to the original program executed under PRS.

	Relative			Primitives by PRS		
	Size	Cycles	Cases	Original	Residual	Increase
`Adjoxo`	1.500	0.507	0.692	16.3%	87.0%	+81.3%
`Braun`	2.526	0.769	0.810	99.2%	100.0%	+0.8%
`Chichelli`	1.494	1.008	0.989	2.7%	2.6%	-5.2%
`Clausify`	5.879	0.951	0.829	0.0%	94.0%	$+\infty$
`Fib`	1.588	0.792	1.000	77.9%	77.9%	0.0%
`KnuthBendix`	2.508	0.866	0.855	63.8%	72.4%	+11.8%
`MSS`	1.920	0.996	0.991	0.0%	0.0%	-0.4%
`Mate`	2.423	0.954	0.918	54.8%	55.1%	+0.6%
`OrdList`	3.887	0.678	0.771	0.0%	0.0%	0.0%
`Parts`	1.733	1.009	0.983	38.8%	56.0%	+30.8%
`PermSort`	2.000	0.756	0.853	100.0%	41.8%	-139.2%
`Queens`	2.535	0.773	0.988	99.5%	85.4%	-16.5%
`Queens2`	2.068	0.966	0.947	4.0%	4.0%	0.0%
`Sudoku`	1.793	0.870	0.911	30.3%	56.1%	+45.9%
`Taut`	2.057	0.856	0.718	0.0%	99.9%	$+\infty$
`While`	3.030	1.062	0.989	60.2%	55.7%	-8.1%
`sumDouble`	0.450	0.200	0.333	50.0%	100.0%	+50.0%
`sumSquares`	0.900	0.378	0.494	66.2%	98.7%	+32.9%
`sumSumEnum`	1.435	0.943	0.494	66.2%	66.9%	+0.9%
Geometric Mean	*1.936*	*0.759*	*0.790*	*7.3%*	*21.7%*	

Table 2. Various metrics for the benchmark programs. Relative values are against the original program executed with PRS (pipeline *b*).

As would be expected, supercompilation can greatly increase the size of the compiled program. There does not seem to be a relationship between relative execution time performance and relative code size. A reduction in the number of cases table evaluated would likely indicate that fusion has taken place, as fewer data structures have been consumed.

In three cases, pipeline *f* still produces programs that perform worse than under PRS alone (pipeline *b*). In comparison to the gains made by other programs, these are only very small performance loses. The reason for these loses is currently unclear. Both for `Chichelli` and for `While`, the proportion of primitive operations performed by PRS has actually fallen. It is currently unclear why `Parts` performs worse when it has a large increase in the number of PRS candidates and a small amount of fusion.

Despite these results, the current prototype of our supercompiler gives a geometric mean speed-up of 24% for programs executed under PRS.

9 Conclusions and Future Work

The `sumDouble` example was chosen to demonstrate the benefits of both PRS and supercompilation. However, we did not see the full magnitude of the combined effect of the two technologies.

Other examples, such as `Queens`, did not benefit from supercompilation. This led to the development of the primitive redex lifting strategy that has largely permitted these examples to benefit from the same effects as `sumDouble`. This strategy does not seem to produce benefits for all results. Further investigation is required to discover why some results still do not improve and a small number get worse.

Still, based on the evidence detailed in this paper, it would appear that PRS and supercompilation can be synergistic, once certain primitive redexes are relocated to maximise design constraints.

There is further scope to exploit the Reduceron design characteristics with the supercompiler. A final inlining phase is required after supercompilation to reduce instantiations at run-time. The current method for selecting candidates for inlining is simplistic. Further performance gains can be made by an improved inlining strategy that considers the constraints on function bodies imposed by the design parameters of the Reduceron.

Future designs of the Reduceron will also permit even more primitive redexes to be speculatively evaluated in parallel. This will likely enable even further performance gains from supercompilation targeted at the Reduceron platform.

Acknowledgements The first author would like to thank Michael Banks and Chris Poskitt for their invaluable proof reading. Neil Mitchell is to be acknowledged for his helpful discussions on the topic. Many thanks also go to Richard Paige for his support and advice. This research was supported, in part, by the EPSRC through the Large-Scale Complex IT Systems project, EP/F001096/1.

References

1. Turchin, V.: A supercompiler system based on the language Refal. ACM SIGPLAN Notices **14**(2) (1979) 46–54
2. Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming **6**(06) (2008) 811–838
3. Naylor, M., Runciman, C.: The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. In: Implementation and Application of Functional Languages (IFL 2007, Revised Selected Papers), Springer LNCS 5083 (2008) 129–146
4. Naylor, M., Runciman, C.: The Reduceron Reconfigured. Available online at `http://www.cs.york.ac.uk/fp/reduceron/` (2010)

5. Naylor, M.: F-lite: a core subset of Haskell. Available online at `http://www.cs.york.ac.uk/fp/reduceron/memos/Memo9.txt` (2008)
6. Peyton Jones, S.L., et al.: The Haskell 98 language and libraries: The revised report. Journal of Functional Programming **13**(1) (Jan 2003)
7. van Eekelen, M., Plasmeijer, R.: Concurrent Clean Language Report (version 2.0). University of Nijmegen (2001)
8. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
9. Partain, W.: The nofib Benchmark Suite of Haskell Programs. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming, Springer-Verlag (1992) 202
10. Bird, R.: A program to solve Sudoku. Journal of Functional Programming **16**(6) (2006) 671
11. Nielson, H.R., Nielson, F.: Semantics with applications: a formal introduction. John Wiley & Sons, Inc., New York, NY, USA (1992)
12. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In: IFL 2007. Volume 5083 of LNCS., Springer-Verlag (May 2008) 147–164

A Note on three Programming Paradigms[⋆]

N.V. Shilov

A.P. Ershov Institute of Informatics Systems,
Lavren'ev av., 6, Novosibirsk 630090, Russia,
http://www.iis.nsk.su/persons/shilov/shilov.htm
shilov@iis.nsk.su

Abstract. This paper is about a puzzle to be solved in three programming paradigms: logic, functional and imperative. It can be considered as a case study of algorithm inversion, since we start with logic algorithm, that answers the question "Is balancing M times sufficient for detecting a single fake in a set of coins?", and finishes with imperative algorithm, that effectively computes the minimal number of balancing that is sufficient for detection the fake. Functional paradigm is used for developing an intermediate functional algorithm that also computes the minimal number of balancing, but inefficiently, while the efficient imperative algorithm is a "lazy memoization" of the functional one.

1 Introduction

1.1 A Puzzle

Let us start with the following fake-coin puzzle:

> A set of 15 coins consists of 14 valid coins and a false one; all valid coins have equal weights, while the false coin has a different weight, one of the valid coins is marked but all other coins (including the fake) are unmarked. Is it possible to identify the fake coin using a balance at most 3 times?

Of course, there is nothing to program in this puzzle: just to solve it "manually" (sorry, mentally) and write a program that outputs the appropriate answer "Yes" or "No". But the following parameterized version of the puzzle is a straightforward generalization of the above one:

> Write a program that inputs a number $N > 0$ of coins under question and a number $W \geq 0$ of marked valid coins ($N \geq W$), and outputs the least number of balancing M that is always sufficient for detecting the unique fake among N coins under question with aid of W valid ones. (Assume that all valid coins have one and the same weight while the false coin has a different weight.)

[⋆] Research is supported by grant of Russian Basic Research Foundation 08-01-00899-a.

We will discuss how to solve this parameterized puzzles in logic, functional and imperative programming in the further sections. But first we would like to provide a short remark about programming paradigms in general and their relations to programming languages.

1.2 What are "Programming Paradigms"?

Robert Floyd was the first who had explicitly used the concept of "Programming Paradigm" in his Turing Award Lecture in 1978 [1]. He referred to Thomas Kuhn's well-known book [2], published just 8 years before. According to T. Kuhn, a paradigm is a method, an approach to a problem formulation (statement) and the ways to solve the problem. R. Floyd had a similar understanding of programming paradigms. In particular, he advocated that a programming language should support one or several of programming paradigms, and wrote: "To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support my extending your language into one that does support my programming methods, I don't need your shiny new languages".

At present the number of essentially different paradigms of programming is already several dozens (see, for example, the list of "programming paradigms" at http://en.wikipedia.org/wiki/Programming paradigm), while the number of programming languages has overcome 2,500 (please refer poster "History of Programming Languages" by O'Reilly). Unfortunately, sometimes relations between programming languages and paradigms (that they support) are vague or not explicit. Due to this reason and due to the number of programming languages that support a paradigm (logic, functional and imperative in particular), we use in this paper logic, functional and imperative pseudocode for representing algorithms (instead of any particular logic, functional or imperative programming language).

1.3 Preliminaries

Let us start with discussion about information/knowledge that is available for a "balancing agent" (anyone who would like to detect the fake) after balancing coins several times.

Before the first balancing, the agent knows initial values of two variables: the first is variable U for the number of coins about which nothing is known, and the second is V for the number of coins which are known to be valid; their initial values are $(N - W)$ and W respectively.

During the first balancing the agent puts some coins at the first and the second pans of the balance; the outcome of the first balancing is either "pans are equal", "the first pan is lighter than the second", or "the first pan is heavier than the second".

The emerging knowledge after the first balancing are four integer values: an updated value of the variable U, the second is an updated value of the variable

V, and values of two new variables L and H that are for the numbers of coins that were balanced and were at lighter and, respectively, at heavier pans.

If to generalize the above observations, then the agent's knowledge about coins after a series of balancing can be represented by the following four non-negative integer values: U of the number of coins, about which nothing is known after the series, L of the number of coins that still have to be verified but were balancing and were at lighter pan, H of the number of coins that still have to be verified but were balancing and were at heavier pan, V of the number of coins, which are known (after this series) to be valid. Observe also that these values meet a natural constraint $U + L + H + V = N$.

Then we have to discuss how to represent balancing in terms of these four values. Let us observe that $L + H > 0$ implies that $U = 0$ (since $L + H > 0$ means that the fake has been balanced already, i.e. all other coins are valid). Due to this reason we can consider two disjoint cases:

1. $L + H = 0$ and $U > 0$,
2. $L + H > 0$ and $U = 0$.

In the first case any balancing consists in a choice of two numbers $u1$ and $u2$ of coins to be put at the first and the second pans, such that $0 < u1 + u2$ (since balancing set should include at least one coin in U), $u1 + u2 \leq U$ (since coins for balancing should be selected in U), $|u1 - u2| \leq V$ (since the oddity in coin numbers should be compensate by coins in V).

In the second case balancing consists in selection of numbers $l1$, $h1$, and $l2$, $h2$ of coins to put at the first and the second pans such that $0 < l1 + l2 + h1 + h2$ (since balancing set should include at least one coin in $L + H$), $l1 + l2 \leq L$ and $h1 + h2 \leq H$ (since coins for balancing should be selected in L and H), $|l1 + h1 - l2 - h2| \leq V$ (since the oddity in coin numbers should be compensate by coins in V).

1.4 Paper outlines

The rest of the paper is organized as follows. The next section 2 addresses logical approach to the parameterized puzzle. Then the section 3 develops a functional solution for the parameterized puzzle. Section 4 converts the functional solution into imperative one by memoization. Finally the last section 5 concludes by some related remarks.

2 Logic of Fake Detection

Problem formulation in logic paradigm is a sound axiomatization of desired data property, problem solution (logic algorithm or program) is a "transformation" of the axiomatization into a knowledge base which can be effectively used by some logic inference machine for proving correct answers for user queries. Below we will illustrate how the logic paradigm can work to solve our parameterized puzzle.

We would like to axiomatize the following data property: it is possible to detect a single fake in a set that consists of $N = U + L + H + V$ coins (where U, L, H, and V have meaning as in the Preliminaries) by balancing them M times at most. Let us denote the appropriate predicate by P and represent the corresponding data property by $P(U, L, H, V, M)$.

Our axiomatization AxP will consist of four axioms. The first axiom describes situation when the fake is detected already: $U+L+H < 1 \ \leftrightarrow \ P(U, L, H, V, M)$. It states that a single fake is already detected in the empty set $(U + L + H = 0)$ and in any singleton set of coins $(U + L + H = 1)$. The second axiom is also straightforward: $L+H > 0 \& P(0, L, H, V+U, M) \ \leftrightarrow \ P(U, L, H, V, M)$. We have discussed already its meaning: if the fake has been balanced already $(L+H > 0)$, then we can say without additional balancing that all other $(V + U)$ coins are valid.

The remaining two axioms establish connections between $P(U, L, H, V, M)$ and $P(..., ..., ..., ..., M - 1)$ and are based on the following observation:

a single fake can be detected in a set of coins
by balancing them M times

iff

there exists a balancing such that
in case of any possible outcome of this balancing
(a) pans are equal, (b) the first is lighter, (c) the first is heavier,
the fake can be detected by balancing coins $(M - 1)$ times.

In particular, the third axiom formalizes this observation in the case when $U > 0$ and $L = H = 0$:

$$\begin{aligned} P(U, 0, 0, V, M) \ \leftrightarrow \ \exists u1, u2\ (0 < u1 + u2 \le U \ \&\ |u1 - u2| \le V \ \&& \\ P(U - u1 - u2, 0, 0, V + u1 + u2, M - 1) \ \&& \\ P(0, u1, u2, V + (U - u1 - u2), M - 1) \ \&& \\ P(0, u2, u1, V + (U - u1 - u2), M - 1)).& \end{aligned}$$

In verbal form: balancing M times is sufficient for detecting the fake ($P(U, 0, 0, V, M)$) iff there exists two sets of coins for balancing at the first and the second pans ($\exists u1, u2$), that they are selected among “unknown” coins ($0 < u1+u2 \le U$) with aid of some oddity-compensating valid ones ($|u1 - u2| \le V$), in case of any possible outcome (‘a’, ‘b’ and ‘c’ from the above) balancing $(M - 1)$ times is sufficient for detecting the fake (‘a’ $\leftrightarrow$ $P(U - u1 - u2, 0, 0, V + u1 + u2, M - 1)$, ‘b’ $\leftrightarrow$ $P(0, u1, u2, V + (U - u1 - u2), M - 1)$, and ‘c’ $\leftrightarrow$ $P(0, u2, u1, V + (U - u1 - u2), M - 1)$ respectively).

The fourth axiom formalizes this connection in the case when $U = 0$ but $L + H > 0$:

$$\begin{aligned} &P(0, L, H, V, M) \ \leftrightarrow \ \exists l1, l2, h1, h2 \\ &(0 < l1 + l2 + h1 + h2 \ \&\ l1 + l2 \le L \ \&\ h1 + h2 \le H \ \&\ |l1 + h1 - l1 - l2| \le V \ \& \\ &\qquad P(0, L - l1 - l2, H - h1 - h2, V + l1 + l2 + h1 + h2, M - 1) \ \& \\ &\qquad P(0, l1, h2, V + (L - l1) + (H - h2), M - 1) \ \& \\ &\qquad P(0, l2, h1, V + (L - l2) + (H - h1), M - 1)). \end{aligned}$$

In words: balancing M times is sufficient for detecting the fake ($P(0, L, H, V, M)$) if there exists two sets of coins for balancing at the first and the second pans ($\exists l1, l2, h1, h2$), that are selected among “light” and “heavy” coins ($0 < l1+l2+h1+h2$, and $l1+l2 \leq L$, $h1+h2 \leq H$) with aid of some oddity-compensating valid ones ($|l1+h1-l1-l2| \leq V$), in case of any possible outcome (‘a’, ‘b’ and ‘c’ from the above) balancing $(M-1)$ times is sufficient for detecting the fake (‘a’ $\leftrightarrow$ $P(0, L-l1-l2, H-h1-h2, V+l1+l2+h1+h2, M-1)$, ‘b’ $\leftrightarrow$ $P(0, l1, h2, V+(L-l1)+(H-h2), M-1)$, and ‘c’ $\leftrightarrow$ $P(0, l2, h1, V+(L-l2)+(H-h1), M-1)$ respectively). Maybe, we have to comment why ‘b’ $\leftrightarrow$ $P(0, l1, h2, V+(L-l1)+(H-h2), M-1)$ and ‘c’ $\leftrightarrow$ $P(0, l2, h1, V+(L-l2)+(H-h1), M-1)$. The reason is very simple: the fake coin can not occur at lighter and heavier pans in series. Really, if the fake was lighter, then it can occur at lighter pans only; otherwise it can occur at heavier pans only.

The axiomatization AxP is finished. It is sound and complete in the following sense.

Proposition 1. *For every tuple of non-negative integer values (U, L, H, V, M), the following equivalence holds: $AxP \vdash P(U, L, H, V, M)$ iff it is possible to detect a single fake in any set that consists of $N = U+L+H+V$ coins by balancing them M times at most (where U, L, H, and V have the same meaning as in the preliminaries).*

Proof (sketch) by induction on $S = U+L+H$. The basic cases when $S \leq 1$ are straightforward. Induction step follows from the following simple argument: for every tuple (U, L, H, V, M), if $(U+L+H) > 1$, then there exists a “corresponding” axiom in AxP with the left-hand side (w.r.t. ‘$\leftrightarrow$’) that matches $P(U, L, H, V, M)$; in the right-hand side of the corresponding axiom P is applied to data (U', L', H', V', M') such that $(U'+L'+H') < (U+L+H)$. ■

This axiomatization AxP is a logic problem formulation for puzzles that are similar to our original non-parameterized one, i.e. queries of the following kind: Whether it is possible to detect fake in a set of coins by balancing them M times. The corresponding logic solution (a pseudocode of a logic algorithm) should be a knowledge base that comprises a list of facts and a list of executable inference rules, that can be used by logic inference machine for efficient deduction of valid queries.

In particular, the first axiom can be transformed into the following four facts: $P(0, 0, 0, V, M)$, $P(1, 0, 0, V, M)$, $P(0, 1, 0, V, M)$, and $P(0, 0, 1, M)$.

The second axiom can be converted into the following rule

$$P(U, L, H, V, M) \ :- \ L+H > 0 \ \& \ P(0, L, H, V+U, M)$$

and be placed in the list of rules at any place; but it makes sense to transform it into the following simpler rule

$$P(U, L, H, V, M) \ :- \ P(0, L, H, V+U, M)$$

and place this simpler rule into the end of the rule list (where it will be used after the rule that corresponds to the third axiom in the case when $L+H > 0$).

The remaining two axioms can be transformed it into the following rules:

$$\begin{aligned} P(U,0,0,V,M)\ :-\ & \exists u1,u2 : 0 < u1+u2 \le U\ (|u1-u2| \le V \& \\ & P(U-u1-u2,0,0,V+u1+u2,M-1)\ \& \\ & P(0,u1,u2,V+(U-u1-u2),M-1)\ \& \\ & P(0,u2,u1,V+(U-u1-u2),M-1)), \\ P(0,L,H,V,M)\ :-\ & \exists l1,l2 : l1+l2 \le L;\ \exists h1,h2 : h1+h2 \le H \\ & (0 < l1+l2+h1+h2\ \&\ |l1+h1-l1-l2| \le V\ \& \\ & P(0,L-l1-l2,H-h1-h2,V+l1+l2+h1+h2,M-1)\ \& \\ & P(0,l1,h2,V+(L-l1)+(H-h2),M-1)\ \& \\ & P(0,l2,h1,V+(L-l2)+(H-h1),M-1)), \end{aligned}$$

and can be put at the top of the rule list in any order (since they match disjoint cases). These rule are executable since all existential quantifiers are bounded.

This logic program is able to solve our initial non-parameterized puzzle by proving $P(14,0,0,1,3)$, and solve other puzzles of several different types. For example, it can solve how many balancing is sufficient for detecting a single fake in a set of 39 coins with aid of additional marked valid coin; if one would like to solve the puzzle than he/she may query this program P(39, 0, 0, 1, ?), and logic inference machine can answer by "finding" some value M, such that $P(39,0,0,1,M)$ is provable in AxP; unfortunately, this value could be any value greater than 4 (39 for example), but not the minimal one (that is 4). It means that we have solved the non-parameterized puzzle, but still have to solve the parameterized one.

3 Functional approach to balancing

Problem formulation in functional paradigm is a set of equations ("equational theory") that should to be valid equalities for functions that we want to compute; functional solution (functional algorithm or program) is set of function definitions that can be effectively used to compute function values for given arguments that convert the equational theory into valid equalities after instantiation. Let us apply below the functional paradigm to our parameterized puzzle.

Proposition 2. *Let us consider a function*[1] $M : \mathcal{N} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ *that for every tuple of arguments U, L, H, V returns the least number of balancing that is sufficient for detecting a single fake in any set of $N = U+L+H+V$ coins (where U, L, H and V have the same meaning as in the preliminaries). Then the following four equalities are valid:*

1. *if* $U+L+H \le 1$ *then* $M(U,L,H,V) = 0$*;*
2. *if* $L+H > 0$ *then* $M(U,L,H,V) = M(0,L,H,V+U)$*;*
3. $M(U,0,0,V) = 1 + \min_{0<u1+u2\le U,\ |u1-u2|\le V} \max\{M(U-u1-u2,0,0,V+u1+u2), M(0,u1,u2,V+U-u1-u2), M(0,u2,u1,V+U-u1-u2)\}$*;*

[1] $\mathcal{N}$ is the set of natural numbers, not a natural number (the amount of coins in particular).

4. $M(0, L, H, V) = 1 + \min_{l1+l2 \le L,\ h1+h2 \le H,\ 0 < l1+l2+h1+h2,\ |l1+h1-l1-l2| \le V}$
$$\max\{M(0, L-l1-l2, H-h1-h2, V+l1+l2+h1+h2),$$
$$M(0, l1, h2, V+(L-l1)+(H-h2)),$$
$$M(0, l2, h1, V+(L-l2)+(H-h1))\}$$

Proof (sketch). These equalities follows from corresponding axioms for predicate p from the previous section 2. Validity of the first two equalities are straightforward, but validity of the last two equalities requires some comments. These equalities start with increment '1+' since corresponding axioms reduce number of balancing from M to $(M-1)$. They have min instead of existential quantifiers, since the function returns the least number of balancing. Finally, max stays on place of conjunction since it represents the worst in three possible outcomes (cases 'a', 'b' and 'c' in the previous section). ■

Proposition 3. *There exists unique function* $M : \mathcal{N} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ *that satisfies equalities 1 – 4 in the proposition 2.*

Proof (sketch) Existence has been proved in the proposition 2. Uniqueness in this case means equality of any function M' (that satisfies equalities 1 – 4 in the proposition 2) to the function M that has been specified in the formulation of the proposition 2. But equality of two functions M and M' means equality of function values $M(U, L, H, V)$ and $M'(U, L, H, V)$ for every quadruple (U, L, H, V) of argument values; equality of these values can be proved by induction on $S = U + L + H$. ■

So we can adopt equalities 1 – 4 in the proposition 2 as the equational theory for the parameterized fake-coin puzzle; together with type definition $M : \mathcal{N} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ these equational theory becomes a functional problem formulation.

The corresponding functional solution (functional algorithm) is the following single function definition that comprises seven clauses:

$M : (1, 0, 0, V) = 0;$
$(0, 1, 0, V) = 0;$
$(0, 0, 1, V) = 0;$
$(0, 0, 0, V) = 0;$
$(U, 0, 0, V) = 1 + \min_{0 < u1+u2 \le U,\ |u1-u2| \le V}$
$$\max\{M(U-u1-u2, 0, 0, V+u1+u2),$$
$$M(0, u1, u2, V+U-u1-u2),\ M(0, u2, u1, V+U-u1-u2)\};$$
$(0, L, H, V) = 1 + \min_{l1+l2 \le L,\ h1+h2 \le H,\ 0 < l1+l2+h1+h2,\ |l1+h1-l1-l2| \le V}$
$$\max\{M(0, L\ l1-l2, H-h1-h2, V+l1+l2+h1+h2),$$
$$M(0, l1, h2, V+(L-l1)+(H-h2)),$$
$$M(0, l2, h1, V+(L-l2)+(H-h1))\};$$
$(U, L, H, V) = M(0, L, H, V+U).$

Here first four clauses represent all solutions of the equation 1, fifth and sixth clauses define how to compute function values to solve equations 3 an 4 respectively. The last clause corresponds to the equation 2. Let us remark that this equation could be converted into the following clause $(U, L, H, V) = \mathit{if}\ L + H > 0\ \mathit{then}\ M(0, L, H, V+U)$ and be placed in between the fourth and the fifth clauses, but (like in logic solution) we transform it into $(U, L, H, V) = M(0, L, H, V + U)$ and place at the end of the definition since at this place it will be used only in the case when $L + H > 0$.

Finally let us exercise manually the above functional solution for some simple data. For example, for values $U = 4$, $L = H = 0$ and $V = 1$, that correspond to question how many balancing is sufficient to detect a single fake in a set of 4 coins with use of one additional valid coin[2]:

$$\begin{aligned}M(4,0,0,1) &= 1 + \min_{0<u1+u2\leq 4,\ |u1-u2|\leq 1} \\ &\qquad \max\{M(4-u1-u2,0,0,1+u1+u2),\ M(0,u1,u2,5-u1-u2), \\ &\qquad\qquad M(0,u2,u1,5-u1-u2)\} = \\ &= 1 + min_{(u1,u2)\in\{(0,1),(1,0),(1,1),(1,2),(2,1),(2,2)\}} \\ &\qquad \max\{M(4-u1-u2,0,0,1+u1+u2),\ M(0,u1,u2,5-u1-u2), \\ &\qquad\qquad M(0,u2,u1,5-u1-u2)\} = \\ &= 1 + min\{\max\{M(3,0,0,2), M(0,0,1,4), M(0,1,0,4)\}, \\ &\qquad \max\{M(3,0,0,2), M(0,1,0,4), M(0,0,1,4)\}, \\ &\qquad \max\{M(2,0,0,3), M(0,1,1,3), M(0,1,1,3)\}, \\ &\qquad \max\{M(1,0,0,4), M(0,1,2,2), M(0,2,1,2)\}, \\ &\qquad \max\{M(1,0,0,4), M(0,2,1,2), M(0,1,2,2)\}, \\ &\qquad \max\{M(0,0,0,5), M(0,2,2,1), M(0,2,2,1)\}\}.\end{aligned}$$

Let us observe that in this exercise all but one functional calls are duplicated: $M(3,0,0,2)$, $M(0,1,1,3)$, $M(0,1,2,2)$, $M(0,2,1,2)$, and $M(0,2,2,1)$; the single call without duplicates is $M(2,0,0,3)$. Moreover, if to proceed further with our exercise, the number of duplicated (triplicated and so on) of calls will increase.

It leads to a natural idea to compute new emerging function calls once, save their values, and reuse saved values when a duplication call emerge. These technique is well-known optimization in functional programming as memoization [3]. Applying this optimization and omitting all these intermediate computations we can finish the above exercise as follows:

$$\begin{aligned}M(4,0,0,1) &= 1 + \min\{\max\{2,0,0\},\ \max\{2,0,0\}, \\ &\qquad \max\{1,1,1\},\ \max\{0,1,1\},\ \max\{0,1,1\},\ \max\{0,2,2\}\} = 2.\end{aligned}$$

4 Categorical Imperative: compute efficiently!

Memorization is not unique technique that can be applied to optimization of the execution of the above functional algorithm. Lazy computations can help also. In particular, lazy memoization consists in saving all emerging function calls, then (when no new calls emerge) sort all saved calls (according to call dependence), execute each saved call once, save all computed values, and use saved values when they become mandatory (inevitable) in computations.

For instance, in the above exercise with functional call $M(4,0,0,1)$ we had several emerging functional calls: $M(0,1,1,3)$, $M(0,1,2,2)$, $M(0,2,1,2)$, $M(3, 0, 0, 2)$, $M(0,2,2,1)$, ... Of course, $M(0,1,1,3)$ is "simpler" than $M(0,1,2,2)$ and $M(0,2,1,2)$, since in the former case just 2 coins are not-verified, while

[2] We omit technical details of this exercise.

there are 3 coins to be verified in the later cases. Call $M(3,0,0,2)$ is simpler that $M(0,2,2,1)$ due to the same reason. In contrast, calls $M(0,1,2,2)$ and $M(0,2,1,2)$ are simple that $M(3,0,0,2)$, since in the former cases the balancing agent has some knowledge about coins while in the later case any information is unavailable. So, for the sake of efficiency, values of $M(0,1,1,3)$, $M(0,1,2,2)$, $M(0,2,1,2)$, $M(3,0,0,2)$, $M(0,2,2,1)$ have to be computed and saved in this order.

But why effectiveness should rely upon computation optimization via lazy memoization? Why not to use memory explicitly and compute in advance all values for all possible (and, maybe, some extra) function calls (but in the right sequence: the simplest — the first, the hardest — the last)? In our particular case (the parameterized fake-coin puzzle) some upper approximation for the set of all possible function call is easy to predict.

Proposition 4. *Let $M : \mathcal{N} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ be a function defined by the functional algorithm in the previous section 3. Then for all non-negative integers U, L, H, and V the set $FCM(U,L,H,V)$ of all function calls that occur in the computation of $M(U,L,H,V)$ (according to the definition in the section 3) is contained in one of the following sets*

- $\{M(0,L,H,V') \ : \ V' \le V+U\} \cup$
  $\cup\{M(0,L',H',V') \ : L' < L, \ H' < H, \ V' \le V+U+L+H-L'-H'\},$
  if $U>0$ *and* $L+H>0$;
- $\{M(0,L',H',V') \ : \ L' < L, \ H' < H, \ V' \le V+U+L+H-L'-H'\},$
  if $U=0$ *and* $L+H>0$;
- $\{M(U',0,0,V') \ : \ U' < U, \ V' \le V+U-U'\} \cup$
  $\cup \ \{M(0,L',H',V') \ : \ U \ge L'+H', \ V' \le V+U-L'-H')\},$
  if $U>0$ *and* $L+H=0$.

Proof (sketch) by induction on $S=U+L+H$. ■

Imperative problem formulation consists in a command that decrees to compute a set of data values in accordance explicit rules of data manipulation and transformation. Imperative problem solution (imperative algorithm or program) is a well-defined sequence of operators that are commands of transformations of values that are stored in individual elements of a computer memory ("memory cells").

A preliminary imperative formulation of the parameterized puzzle can be as follows: for a given integer value $N \ge 0$ fill-in a four-dimensional table $T[0..N,0..N,0..N,0..N]$ by corresponding values of the function M specified in the proposition 2, i.e. by integers such that for every tuple (U,L,H,V) of non-negative integers, if $U+L+H+V \le N$, then $T[U,L,H,V]$ is the least number of balancing to detect a single fake in a set of $U+L+H+V$ coins (where U, L, H, V have the same meaning as in the preliminaries).

According to our functional solution of the parameterized fake-coin puzzle, this preliminary imperative formulation can be transformed into the following final one: for a given integer value $N \ge 0$ fill-in a four-dimensional table $T[0..N,0..N,0..N,0..N]$ in accordance with following rules (whenever the sum

of the indexes $\leq N$):
$T(1,0,0,V) = T(0,1,0,V) = T(0,0,1,V) = T(0,0,0,V) = 0$ for all $V \in [0..N]$;

$$T(U,0,0,V) \;=\; 1 + \min_{0<u1+u2\leq U,\ |u1-u2|\leq V} \max\{T(U-u1-u2,0,0,V+u1+u2),\ T(0,u1,u2,V+U-u1-u2),\ T(0,u2,u1,V+U-u1-u2)\};$$

$$T(0,L,H,V) \;=\; 1 + \min_{l1+l2\leq L,\ h1+h2\leq H,\ 0<l1+l2+h1+h2,\ |l1+h1-l1-l2|\leq V} \max\{T(0,L-l1-l2,H-h1-h2,V+l1+l2+h1+h2),\ T(0,l1,h2,V+(L-l1)+(H-h2)),\ T(0,l2,h1,V+(L-l2)+(H-h1))\};$$

$$T(U,L,H,V) \;=\; T(0,L,H,V+U).$$

The above Proposition 4 leads to an idea how to fill-in the table in the right sequence: in the order of ascending of sum of indexes $U+L+H+V$, but for every value of the sum in $[0..N]$, elements $T[0,L,H,V]$ should be filled first, next — elements $T[U,0,0,V]$, finally — elements $T[U,L,H,V]$. This idea can be implemented by imperative solution as follows:

```
$const n;$
$var\ u,l,h,v,s,l1,l2,h1,h2,u1,u2 :\ integer;$
$var\ t :\ integer\ array\ of\ [0..n,0..n,0..n,0..n];$
$begin$
$for\ v = 0\ to\ n\ do$
$begin\ t(1,0,0,v) := 0;\ t(0,1,0,v) := 0;\ t(0,0,1,v) := 0;\ t(0,0,0,v) = 0\ end;$
$for\ s = 2\ to\ n\ do$
    $begin$
    $for\ l = 0\ to\ s\ do$
        $for\ h = 0\ to\ (s-l)\ do$
            $t(0,l,h,s-l-h) := 1+$
                $\min_{l1+l2\leq l,\ h1+h2\leq h,\ 0<l1+l2+h1+h2,\ |l1+h1-l1-l2|\leq(s-l-h)} \max\{$
                    $t(0,l-l1-l2,h-h1-h2,s-l-h+l1+l2+h1+h2),$
                    $t(0,l1,h2,s-l1-h2),\ t(0,l2,h1,s-l2-h1)\};$
    $for\ u = 0\ to\ s\ do$
        $t(u,0,0,s-u) := 1+\min_{0<u1+u2\leq u,\ |u1-u2|\leq(s-u)} \max\{$
            $t(u-u1-u2,0,0,s-u+u1+u2),\ t(0,u1,u2,s-u1-u2),$
            $t(0,u2,u1,s-u1-u2)\};$
    $for\ u = 0\ to\ s\ do$
        $for\ l = 0\ to\ (s-u)\ do$
            $for\ h = 0\ to\ (s-u-l)\ do$
                $t(u,l,h,v) := t(0,l,h,s-l-h)$
    $end$
$end.$
```

Imperative solution is done. Let us remark, that it was extracted from a functional one by lazy memoization, which, in turn, was extracted from a logic solution.

5 Conclusion

We start this paper with logic solution for a puzzle "*Is balancing M-times sufficient for detecting the fake in a set of coins?*", and finishes with the imperative algorithm, that effectively computes the minimal number of balancing that is sufficient for detection the fake. Thus the paper can be considered as a kind of case-study of algorithm (a program) inversion.

At the same time some ideas (that have been exploited in this paper) can be generalized for inverting some logic and functional algorithms and programs. In particular, let us assume that a functional algorithm (program) defines a partial function $F : D \to R$ such that for every $X \in D$ it is easy to compute an upper approximation of the set of all function calls $FCF(X)$, that emerge in the computation of $F(X)$. Then the function F can be efficiently inverted in the following sense: for every $Y \in R$, for every finite $D' \subseteq D$ the set[3]

$$(F \upharpoonright D')^{-}(Y) \;=\; \{X \in D' \;:\; F(X) = Y\}$$

can be computed by dynamic programming. For instance, function $M : \mathcal{N} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N} \to \mathcal{N}$ (that was discussed in section 3) enjoys this property (see Proposition 4) and can be inverted by means of the table T (computed in the section 4 by dynamic programming).

Let us also remark that the parameterized puzzle has been used in Asian Regional ACM International Collegiate Programming Contest in year 2000 (problem H). One can try the following (more complicated) version of the puzzle [4]:

> Write a program with 3 inputs: a number $U \geq 0$ of coins under question, a number $V \geq 0$ of marked valid coins, and a limit $K \geq 0$ for the number of balancing, that outputs either the string impossible, or another executable interactive program ALPHA (in the same language) with respect to existence of a strategy to identify a single false coin among U coins with use of additional V marked valid coins and weighing coins K times at most. Your program should output impossible iff there is no such strategy. Otherwise it should output the program ALPHA which implements a strategy in the following settings.
> – All $(U + V)$ coins are enumerated by consecutive numbers from 1 to $(U + V)$, all marked valid coins are enumerated by initial numbers from 1 up to V.
> – Every interactive session with ALPHA begins with user's initial decision on the coin number of the false coin in $[(V + 1)..(V + U)]$ and whether it is lighter or heavier.
> – Every interactive session with ALPHA consists of a series of rounds and the number of rounds in the session can not exceed K.
> – In each round $i \in [1..K]$ the program ALPHA outputs two disjoint subsets of coin numbers to be placed on the first and the second pans of the balance and prompts the user with '?' for a reply.

[3] '$\upharpoonright$' denotes domain restriction of a function.

- The user in his/her turn replies with '<', '=', or '>' in accordance with the initial decision on the number of the false coin and its weight.
- Every interactive session with ALPHA finishes with the final output string "False coin number is " followed by the coin number of the false coin.

Since the problem is to write a program which generates another program we would like to refer to the first program as a *metaprogram* and to the problem as the *metaprogram* problem respectively. This problem has been discussed in the context of propositional program logics and solved in [4].

References

1. Floyd R.W. *The paradigms of Programming.* Communications of ACM. v.22, 1979, p.455-460.
2. Kuhn T.S. *The structure of Scientific Revolutions.* Univ. of Chicago Press, 1970.
3. Astapov D. *Recursion + Memoization = Dynamic Programming.* (in Russian) Practice of Functional Programming, n.3, 2009, p. 17-33. Available at http://fprog.ru/2009/issue3/.
4. Shilov N.V. and Yi K. *How to find a coin: propositional program logics made easy.* Current Trends in Theoretical Computer Science, World Scientific, v. 2, 2004, p.181-214.

A A Puzzle for You...

Finally I would like to draw attention to another coin puzzles, which can be called Find Money Puzzle. Again, let us start with a non-parameterized puzzle[4] and finish with a parameterized version[5]. Non-parameterized puzzle (enjoy!):

> A set consists of 40 coins, three of them are fake and 37 are valid ones. All coins look identical, all valid coins have equal weight, but all fakes are lighter than the valid ones. Is it possible to select 18 valid by balancing coins 3 times at most?

Parameterized Find Money Puzzle (logic, functional, and imperative feasible algorithms are welcome by e-mail to author):

> Write a program that inputs a number $N \geq 0$ of coins, a number $L \in [0..N]$ of fake coins (the remaining $(N - L)$ coins are valid), a number $V \in [0..(N - L)]$, and outputs the least number of balancing M that is always sufficient for selecting V valid coins in this set. (Assume that all valid coins have one and the same weight while fake coins are lighter.)

[4] I know how to solve it.
[5] I do not know how to solve it.

Научное издание

Труды конференции

Сборник трудов Второго международного памяти В.Ф. Турчина семинара по метавычислениям в России, г. Переславль-Залесский, 1-5 июля 2010 г.

Под редакцией А. П. Немытых.

Для научных работников, аспирантов и студентов.

Издательство **«Университет города Переславля»**,
152020 г. Переславль-Залесский, ул. Советская 2.

Гарнитура **Computer Modern**. Формат **60×84/16**.
Дизайн обложки: *Н. А. Федотова*. Уч. изд. л. **11.12**.
Усл. печ. л. **11.53**. Подписано к печати **22.05.2010**.
Ответственные за выпуск: *С. В. Знаменский*,
Н. А. Федотова.

Отпечатано в ООО ≪Регион≫. Печать цифровая. Бумага офсетная. Тираж 100 экз. Заказ 40.
152025 Ярославская область г. Переславль-Залесский ул. Строителей, д. 41