Russian Academy of Sciences
Ailamazyan Program Systems Institute

Fourth International Valentin Turchin Workshop on Metacomputation

Proceedings
Pereslavl-Zalessky, Russia, June 29 – July 3, 2014

Издательство
Университета города Переславля
Pereslavl-Zalessky

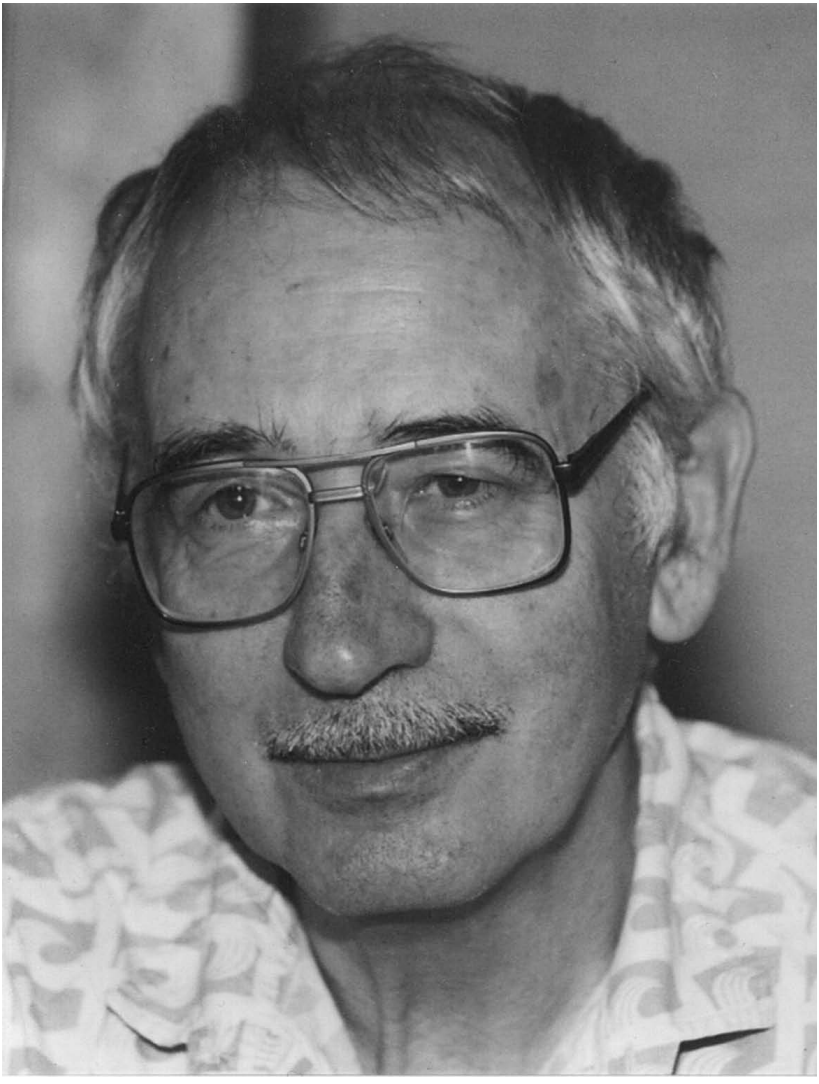
УДК 004.42(063)
ББК 22.18

Ч-52

Fourth International Valentin Turchin Workshop on Metacomputation // Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation. Pereslavl-Zalessky, Russia, June 29 – July 3, 2014 / *Edited by A. V. Klimov and S. A. Romamenko.* — Pereslavl Zalessky: Publishing House "University of Pereslavl", 2014, **256** p. — ISBN 978-5-901795-31-6

Четвертый международный семинар по метавычислениям имени В. Ф. Турчина // Сборник трудов Четвертого международного семинара по метавычислениям имени В. Ф. Турчина, г. Переславль-Залесский, 29 июня – 3 июля 2014 г. / *Под редакцией А. В. Климова и С. А. Романенко.* — Переславль-Залесский: Издательство «Университет города Переславля», 2014, **256** с. — ISBN 978-5-901795-31-6

ISBN 978-5-901795-31-6



Valentin Turchin
(1931–2010)

Preface

The Fourth International Valentin Turchin Workshop on Metacomputation, META 2014, was held on June 29 – July 3, 2014 in Pereslavl-Zalessky. It belongs to a series of workshops organized biannually by Ailamazyan Program Systems Institute of Russian Academy of Sciences and Ailamazyan University of Pereslavl.

The workshops are devoted to the memory of Valentin Turchin, a founder of *metacomputation*, the area of computer science dealing with manipulation of programs as data objects, various program analysis and transformation techniques.

The topics of interest of the workshops include supercompilation, partial evaluation, distillation, mixed computation, generalized partial computation, slicing, verification, mathematical problems related to these topics, their applications, as well as cross-fertilization with other modern research and development directions.

Traditionally each of the workshops starts with a Valentin Turchin memorial session, in which talks about his personality and scientific and philosophical legacy are given.

The papers in these proceedings belong to the following topics.

Valentin Turchin memorial

- Andrei V. Klimov. *On Valentin Turchin's Works on Cybernetic Philosophy, Computer Science and Mathematics*

Supercompilation and distillation

- Sergei A. Grechanik, Ilya G. Klyuchnikov, and Sergei A. Romanenko. *Staged Multi-Result Supercompilation: Filtering by Transformation*
- Jun Inoue. *Supercompiling with Staging*
- Neil D. Jones and G. W. Hamilton. *Towards Understanding Superlinear Speed-up by Distillation*

Verification and proving properties of programs and metaprograms

- Sergei A. Grechanik. *Inductive Prover Based on Equality Saturation for a Lazy Functional Language (Extended Version)*
- Dimitur Nikolaev Krustev. *An Approach for Modular Verification of Multi-Result Supercompilers (Work in Progress)*
- Andrew M. Mironov. *A Method of a Proof of Observational Equivalence of Processes*

Mathematical topics related to metacompuation

- Antonina N. Nepejvoda. *Turchin's Relation and Subsequence Relation on Traces Generated by Prefix Grammars*
- Nikolai N. Nepejvoda. *Algebraic Structures of Programs: First Steps to Algebraic Programming*

Practical metacompuation and applications

- Ilya G. Klyuchnikov. *Nullness Analysis of Java Bytecode via Supercompilation over Abstract Values*
- Michael Dever and G. W. Hamilton. *AutoPar: Automating the Parallelization of Functional Programs*
- Venkatesh Kannan and G. W. Hamilton. *Extracting Data Parallel Computations from Distilled Programs*
- Arkady V. Klimov. *Construction of Exact Polyhedral Model for Affine Programs with Data Dependent Conditions*

The files of the papers and presentations of this and the previous workshops as well as other information can be found at the META sites:

- META 2008: http://meta2008.pereslavl.ru/
- META 2010: http://meta2010.pereslavl.ru/
- META 2012: http://meta2012.pereslavl.ru/
- META 2014: http://meta2014.pereslavl.ru/

June 2014

Andrei Klimov
Sergei Romanenko

Organization

Workshop Chair

Sergei Abramov, Ailamazyan Program Systems Institute of RAS, Russia

Program Committee Chairs

Andrei Klimov, Keldysh Institute of Applied Mathematics of RAS, Russia
Sergei Romanenko, Keldysh Institute of Applied Mathematics 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
Arkady Klimov, Institute for Design Problems in Microelectronics of RAS
Ilya Klyuchnikov, JetBrains; Keldysh Institute of Applied Mathematics of RAS
Dimitur Krustev, IGE+XAO Balkan, Bulgaria
Alexei Lisitsa, Liverpool University, Great Britain
Neil Mitchell, Standard Charted, United Kingdom
Antonina Nepeivoda, Ailamazyan Program Systems Institute of RAS, Russia
Peter Sestoft, IT University of Copenhagen, Denmark
Alexander Slesarenko, Keldysh Institute of Applied Mathematics of RAS, Russia
Morten Sørensen, Formalit, Denmark

Invited Speaker

Neil D. Jones, Professor Emeritus of the University of Copenhagen, Denmark

Sponsoring Organizations

Russian Academy of Sciences

Russian Foundation for Basic Research (grant № 14-07-06026-г)

Table of Contents

AutoPar: Automatic Parallelization of Functional Programs

Michael Dever & G. W. Hamilton
{mdever, hamilton}@computing.dcu.ie

Dublin City University

Abstract. In this paper we present a novel, fully automatic transformation technique which parallelizes functional programs defined using any data-type. In order to parallelize these programs our technique first derives conversion functions which allow the data used by a given program to be well-partitioned. Following this the given program is redefined to make use of this well-partitioned data. Finally, the resulting program is explicitly parallelized. In addition to the automatic parallelization technique, we also present the results of applying the technique to a sample program.

1 Introduction

As the pervasiveness of parallel architectures in computing increases, so does the need for efficiently implemented parallel software. However, the development of parallel software is inherently more difficult than that of sequential software as developers are typically comfortable developing sequentially and can have problems thinking in a parallel setting [29]. Yet, as the limitations of single-core processor speeds are reached, the developer has no choice but to reach for parallel implementations to obtain the required performance increases.

Functional languages are well suited to parallelization due to their lack of side-effects, a result of which is that their functions are stateless. Therefore, one process executing a pure function on a set of data can have no impact on another process executing a function on another set of data as long as there are no data dependencies between them. This gives functional programs a semantically transparent *implicit task parallelism*.

Due to the nature of functional languages, many functions make use of intermediate data-structures to generate results. The use of intermediate data-structures can often result in inefficiencies, both in terms of execution time and memory performance [38]. When evaluated in a parallel environment, the use of intermediate data-structures can result in unnecessary communication between parallel processes. Elimination of these intermediate data-structures is the motivation for many functional language program transformation techniques, such as that of *distillation* [14, 16] which is capable of obtaining a super-linear increase in efficiency.

The automatic parallelization technique presented in this paper makes use of our previously published automatic partitioning technique [9], which facilitates

the partitioning of data of any type into a corresponding *join*-list. This is used to generate functions which allow data of any type to be converted into a well-partitioned *join*-list and also allow the data in a well-partitioned *join*-list to be converted back to its original form.

Following the definition of these functions, we distill the given program which results in an equivalent program defined in terms of a well-partitioned *join*-list, in which the use of intermediate data-structures has been eliminated. Upon distilling a program defined on a well-partitioned *join*-list, we then extract the parallelizable expressions in the program. Finally, we apply another transformation to the distilled program which parallelizes expressions operating on well-partitioned *join*-lists.

The remainder of this paper is structured as follows: Section 2 describes the language used throughout this paper. Section 3 describes how we implement explicit parallelization using Glasgow parallel Haskell. Section 4 presents an overview of the distillation program transformation technique and describes how we use distillation to convert programs into equivalent programs defined on well-partitioned data. Section 5 describes our automatic parallelization technique. Section 6 presents the application of the automatic parallelization technique to a sample program. Section 7 presents a selection of related work and compares our techniques with these works and Section 8 presents our conclusions.

2 Language

The simple higher-order language to be used throughout this paper is shown in Fig. 1. Within this language, a data-type T can be defined with the constructors $c_1, \ldots, c_m$ each of which may include other types as parameters. Polymorphism is supported in this language via the use of type variables, α. Constructors are of a fixed arity, and within $c\ e_1 \ldots e_k$, k must be equal to constructor c's arity. We use $(e :: t)$ to denote an expression e of type t. Case expressions must only have non-nested patterns. Techniques exist to transform nested patterns into equivalent non-nested versions [1, 37].

Within this language, we use **let** $x_1 = e_1\ \ldots\ x_n = e_n$ **in** e_o to represent a series of nested **let** statements as shown below:

$$\begin{aligned}\textbf{let}\ x_1 = e_1\ \ldots\ x_n = e_n\ \textbf{in}\ e_o\ \equiv\ \textbf{let}\ & x_1 = e_1 \\ & \vdots \\ & \textbf{let}\ x_n = e_n \\ & \textbf{in}\ \ e_0\end{aligned}$$

The type definitions for *cons*-lists and *join*-lists are as shown in Fig. 2. We use the usual Haskell notation when dealing with *cons*-lists: [] represents an empty *cons*-list, (Nil), $[x]$ represents a *cons*-list containing one element, $(Cons\ x\ Nil)$, and $(x : xs)$ represents the *cons*-list containing the *head* x and the *tail* xs, $(Cons\ x\ xs)$.

$$
\begin{array}{lll}
t ::= \alpha & & \text{Type Variable} \\
\quad \mid\ T\ t_1 \ldots t_g & & \text{Type Application} \\
 & & \\
\textbf{data}\ T\ \alpha_1 \ldots \alpha_g ::= c_1\ t_{1_1}\ \ldots\ t_{1_{n_1}} & & \text{Data-Type} \\
\quad \vdots & & \\
\quad \mid\ c_m\ t_{m_1}\ \ldots\ t_{m_{n_m}} & & \\
 & & \\
e ::= x & & \text{Variable} \\
\quad \mid\ c\ e_1 \ldots e_k & & \text{Constructor} \\
\quad \mid\ f & & \text{Function} \\
\quad \mid\ \lambda x.e & & \text{Lambda Abstraction} \\
\quad \mid\ e_0\ e_1 & & \text{Application} \\
\quad \mid\ \textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k & & \text{Case Expression} \\
\quad \mid\ \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_0 & & \text{Let Expression} \\
\quad \mid\ e_0\ \textbf{where}\ f_1 = e_1\ \ldots\ f_n = e_n & & \text{Where Expression} \\
 & & \\
p ::= c\ x_1\ \ldots\ x_k & & \text{Pattern}
\end{array}
$$

Fig. 1: Language Definition

$$
\begin{array}{ll}
data\ List\ a & ::= Nil \\
 & \quad \mid\ Cons\ a\ (List\ a) \\
 & \\
data\ JList\ a & ::= Singleton\ a \\
 & \quad \mid\ Join\ (JList\ a)\ (JList\ a)
\end{array}
$$

Fig. 2: *cons*-List and *join*-List Type Definitions

3 Glasgow Parallel Haskell

In order to parallelize the various programs described in this paper we make use of Glasgow parallel Haskell (GpH) [36] which is an extension to Haskell. GpH supports parallelism by using strategies for controlling the parallelism involved. Parallelism is introduced via *sparking* (applying the **par** strategy) and evaluation order is determined by applying the **pseq** strategy. As an example, the expression $x\ `par`\ y$ **may** spark the evaluation of x in parallel with that of y, and is semantically equivalent to y. As a result of this, when using $x\ `par`\ y$, the developer indicates that they believe evaluating x in parallel may be useful, but leave it up to the runtime to determine whether or not the evaluation of x is run in parallel with that of y [25]. *pseq* is used to control evaluation order as $x\ `pseq`\ y$ will strictly evaluate x before y. Usually, this is used because y cannot be evaluated until x has been.

As an example, the expression $x\ `par`\ (y\ `pseq`\ x + y)$ sparks the evaluation of x in parallel with the strict evaluation of y. After y has been evaluated, $x + y$ is then evaluated. If the parallel evaluation of x has not been completed at this

point, then it will be evaluated sequentially as part of $x + y$. As a result of this $x\ 'par'\ (y\ 'pseq'\ x + y)$ is semantically equivalent to $x + y$, but we may see some performance gain from sparking the evaluation of x in parallel. Below is a simple example of the use of GpH, which calculates fibonacci numbers in parallel:

$$
\begin{aligned}
fib = \lambda x.&\textbf{case}\ x\ \textbf{of} \\
&0 \rightarrow 1 \\
&1 \rightarrow 1 \\
&n \rightarrow \textbf{let}\ x\ = \mathit{fib}\ (n - 1) \\
&\quad\ \textbf{in let}\ y\ = \mathit{fib}\ (n - 2) \\
&\quad\quad\ \textbf{in}\ x\ `\mathit{par}'\ (y\ `\mathit{pseq}'\ x + y)
\end{aligned}
$$

Given a number x, the function fib sparks the evaluation of $fib\ (n - 1)$ to *weak-head normal form* in parallel with the full evaluation of $fib\ (n - 2)$. When $fib\ (n - 2)$ has been fully evaluated, it is then added to the result of the evaluation of $fib\ (n - 1)$. $fib\ (n - 1)$ can be fully evaluated in parallel by having the *rdeepseq* strategy applied to it.

We have selected Glasgow parallel Haskell for our implementation language, due to its conceptual simplicity, its semantic transparency and its separation of algorithm and strategy. Another reason for the selection of Glasgow parallel Haskell is its management of threads: it handles the creation/deletion of threads, and determines whether or not a thread should be sparked depending on the number of threads currently executing.

4 Transforming Data to Well-Partitioned *Join*-Lists

There are many existing automated parallelization techniques [2–7, 10, 19–22, 26, 30, 31, 35], which, while powerful, require that their input programs are defined using a *cons*-list, for which there is a straightforward conversion to a well-partitioned *join*-list. This is an unreasonable burden to place upon a developer as it may not be intuitive or practical to define their program in terms of a *cons*-list.

To solve this problem we have previously defined a novel transformation technique that allows for the automatic partitioning of an instance of any data-type [9]. A high-level overview of the automatic partitioning technique is presented in Figure 3. We combine this technique with distillation in order to automatically convert a given program into one defined on well-partitioned data. We do not give a full description of the automatic partitioning technique here, it is sufficient to know that it consists of the following four steps:

1. Given a program defined on an instantiated data-type, τ, we use the definition of τ to define a corresponding data-type, τ', instances of which will contain the non-inductive components from data of type τ.
2. Derive a *partitioning function*, $partition_\tau$, which will allow data of type τ to be converted into a well-partitioned *join*-list containing data of type τ'.

3. Derive a *rebuilding function*, $rebuild_\tau$, which will convert a *join*-list containing data of type τ' into data of type τ.
4. Distill a program equivalent to the given program which is defined on a well-partitioned *join*-list.

Using these four steps, we can automatically convert a given program into an equivalent program defined on well-partitioned data. Section 4.1 presents an overview of the distillation program transformation technique. Section 4.2 describes how we combine distillation and the automatic partitioning technique in order to convert a program into an equivalent one defined on well-partitioned data.

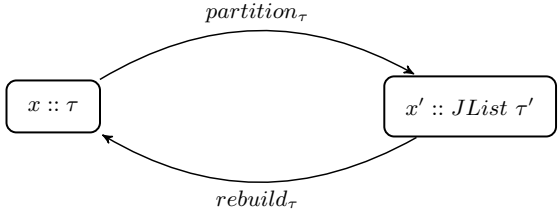


Fig. 3: Data Partitioning Functions

4.1 Distillation

Distillation [14, 16–18] is a powerful fold/unfold based program transformation technique which eliminates intermediate data-structures from higher-order functional programs. It is capable of obtaining a super-linear increase in efficiency and is significantly more powerful than the *positive-supercompilation* program transformation technique [15, 32, 33] which is only capable of obtaining a linear increase in efficiency [34].

Distillation essentially performs normal-order reduction according to the reduction rules defined in [14]. *Folding* is performed on encountering a renaming of a previously encountered expression, and *generalization* is performed to ensure the termination of the transformation process. The expressions compared prior to folding or generalization within the distillation transformation are the results of symbolic evaluation of the expressions, whereas in positive-supercompilation, the syntax of the expressions are compared. Generalization is performed upon encountering an expression which contains an embedding of a previously encountered expression. This is performed according to the *homeomorphic embedding relation*, which is used to detect divergence within term rewriting systems [8].

We do not give a full description of the distillation algorithm here; details can be found in [14]. The distillation algorithm is not required to understand

the remainder of this paper, it is sufficient to know that distillation can be used to eliminate the use of intermediate data structures in expressions.

4.2 Distilling Programs on Well-Partitioned Data

Given a sequential program, f, defined using an instantiated data-type, τ, we first define a conversion function which will convert data of type τ into a well partitioned *join*-list, $partition_\tau$, and one which will convert a *join*-list into data of type τ, $rebuild_\tau$, as depicted in Figure 3.

By applying distillation, denoted $\mathcal{D}$, to the composition of f and $rebuild_\tau$, $\mathcal{D}[\![f \circ rebuild_\tau]\!]$, we can automatically generate a function, f_{wp}, which is equivalent to f but is defined on a well-partitioned *join*-list containing data of type τ'. A high level overview of this process is presented in Figure 4.

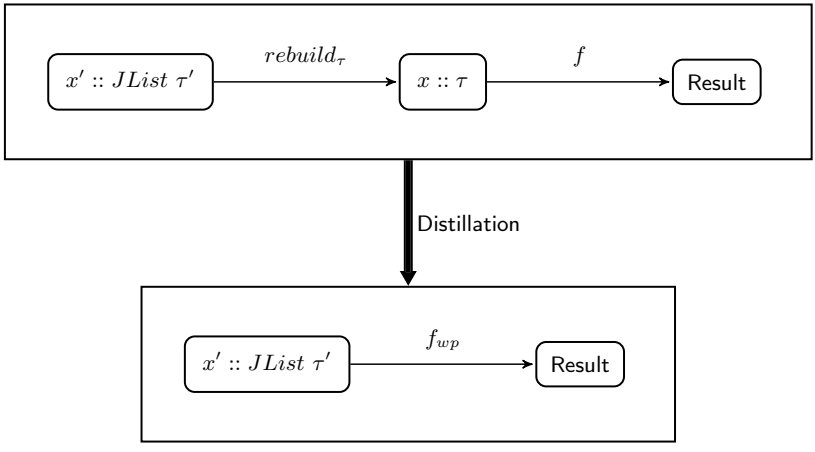


Fig. 4: Distillation of Programs on Well-Partitioned Data

Obviously, f_{wp} is defined on a well-partitioned *join*-list, whereas f is defined on data of type τ. We can generate the correct input for f_{wp} by applying $partition_\tau$ to the input of f and using the result of this as the input to f_{wp}.

5 Automatically Parallelizing Functional Programs

Given a program, once we have distilled an equivalent program defined on a well-partitioned *join*-list, this can be transformed into an equivalent explicitly parallelized program defined on a well-partitioned *join*-list. Our *novel* parallelization technique consists of two steps:

1. Ensure that expressions which are parallelizable are independent.

2. Explicitly parallelize the resulting parallelizable expressions.

Using these two steps, we can automatically convert a given program into an equivalent explicitly parallelized program that operates on well-partitioned data. Section 5.1 describes the process by which we ensure that expressions which are parallelizable are independent and Section 5.2 describes the process which explictly parallelizes a program defined on well-partitioned data.

5.1 Extracting Independent Parallelizable Expressions

Prior to explicitly parallelizing the output of distillation we must first make sure that any expressions that will be parallelized are independent of each other. Given an expression, e, we define its parallelizable expressions as the set of unique maximal sub-expressions of e which operate on a single *join*-list. That is, each parallelizable expression of e must be the largest sub-expression of e which operates on a *join*-list and is not a variable. Once we have identified all the parallelizable expressions within e, these are extracted into a **let** statement which is equivalent to e.

As an example, consider the function f, as defined below:

$$\begin{aligned} f = \lambda x.\ \lambda n.&\textbf{case}\ x\ \textbf{of} \\ &\mathit{Join}\ l\ r \rightarrow f\ r\ (f\ l\ n) \end{aligned}$$

Within f, both the *join*-lists l and r are evaluated, therefore we identify the parallelizable expressions in which they are evaluated, $f\ l\ n$ and $f\ r$ respectively. These are then extracted into a **let** statement equivalent to $f\ r\ (f\ l\ n)$ as follows:

$$\begin{aligned} f = \lambda x.\ \lambda n.&\textbf{case}\ x\ \textbf{of} \\ &\mathit{Join}\ l\ r \rightarrow \textbf{let}\ x_1 = f\ l\ n \\ &\qquad\qquad\quad\ \ x_2 = f\ r \\ &\qquad\qquad\ \ \textbf{in}\ \ x_2\ x_1 \end{aligned}$$

As the output of distillation contains no **let** statements, this process allows us to explicitly parallelize expressions that operate on well-partitioned *join*-lists, as any extracted expression within a **let** statement must be a parallelizable expression.

5.2 Explicit Parallelization of Functional Programs

Given a distilled program which operates on a well-partitioned *join*-list, f_{wp}, which has had its parallelizable expressions made independent, the final step of the automatic parallelization technique is to apply another transformation, $\mathcal{T}_p$, to f_{wp} in order to explicitly parallelize expressions which operate on well-partitioned *join*-lists. The result of this transformation is a function, f_{par}, which is equivalent to f but is defined on a well-partitioned *join*-list and has been explicitly parallelized. A high-level overview of this process is shown in Figure 5. The transformation rules for $\mathcal{T}_p$ are defined as shown in Fig. 6.

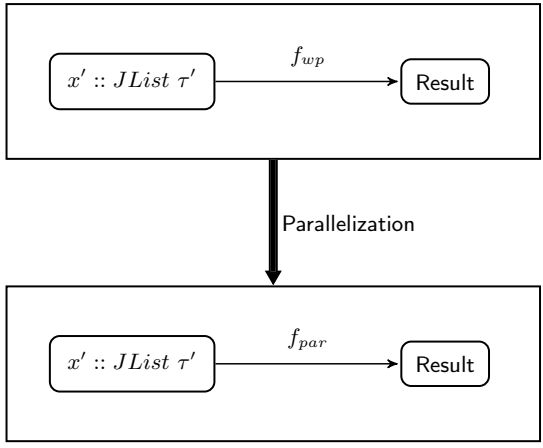


Fig. 5: Parallelization of Program defined on Well-Partitioned Data

The majority of the transformation rules should be self explanatory: variables, abstraction variables, constructor names are left unmodified. The bodies of constructors, applications, abstractions and function definitions are transformed according to the parallelization rules, as are the selectors and branch expressions of **case** statements.

The most interesting of the presented transformation rules is the one that deals with **let** statements. As each extracted expression within a **let** statement is a parallelizable expression its evaluation should be sparked in parallel. However, as these may contain further parallelizable expressions, we first apply the parallelization algorithm to the extracted expressions, as well as to the body of the **let** statement. Finally, we spark the evaluation of all but one of the parallelizable expressions in parallel with the full evaluation of the last parallelizable expression, x_n. This means that the evaluation of the parallelized version of e_0 will not begin until x_n has been fully evaluated, at which point, hopefully, the remaining parallelized expressions will have been evaluated in parallel.

As an example, consider again the definition of f shown previously, which has had its parallelized expressions extracted. Application of $\mathcal{T}_p$ to f results in the definition of f_{par} shown below:

$$
\begin{aligned}
f_{par} = \lambda x.\ \lambda n.&\mathbf{case}\ x\ \mathbf{of} \\
&Join\ l\ r \rightarrow\ \mathbf{let}\ x_1 = f_{par}\ l\ n \\
&\qquad\qquad\quad\ \ x_2 = f_{par}\ r \\
&\qquad\quad\ \mathbf{in}\ \ x_1\ `par`\ x_2\ `pseq`\ x_2\ x_1
\end{aligned}
$$

$$
\begin{array}{ll}
\mathcal{T}_p[\![x]\!] & = x \\
\mathcal{T}_p[\![c\ e_1\ \ldots\ e_n]\!] & = c\ \mathcal{T}_p[\![e_1]\!]\ \ldots\ \mathcal{T}_p[\![e_n]\!] \\
\mathcal{T}_p[\![f]\!] & = f \\
\mathcal{T}_p[\![\lambda x.e]\!] & = \lambda x.\mathcal{T}_p[\![e]\!] \\
\mathcal{T}_p[\![e_0\ e_1]\!] & = \mathcal{T}_p[\![e_0]\!]\ \mathcal{T}_p[\![e_1]\!] \\
\mathcal{T}_p[\![\textbf{case}\ x\ \textbf{of}\ p_1 \to e_1 \mid \ldots \mid p_k \to e_k]\!] & = \textbf{case}\ x\ \textbf{of}\ p_1 \to \mathcal{T}_p[\![e_1]\!] \mid \ldots \mid p_k \to \mathcal{T}_p[\![e_k]\!] \\
\mathcal{T}_p[\![e_0\ \textbf{where}\ f_1 = e_1\ \ldots\ f_n = e_n]\!] & = \mathcal{T}_p[\![e_0]\!]\ \textbf{where}\ f_1 = \mathcal{T}_p[\![e_1]\!]\ \ldots f_n = \mathcal{T}_p[\![e_n]\!] \\
\mathcal{T}_p[\![\textbf{let}\ x_1 = e_1\ \ldots\ x_n = e_n \textbf{in}\ e_0]\!] & \\
\quad = \left\{ \begin{array}{l} \textbf{let}\ x_1 = \mathcal{T}_p[\![e_1]\!] \\ \qquad \vdots \\ \quad x_n = \mathcal{T}_p[\![e_n]\!] \\ \textbf{in}\ \ x_1\ 'par'\ \ldots\ x_{n-1}\ 'par'\ x_n\ 'pseq'\ \mathcal{T}_p[\![e_0]\!] \end{array} \right. &
\end{array}
$$

Fig. 6: Transformation Rules for Parallelization

6 Automatic Parallelization of a Sample Program

This section presents an example of the application of the automatic parallelization technique to the program *sumList*, which calculates the sum of a *cons*-list of numbers, as shown below:

$$
\begin{array}{l}
sumList = \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad Nil \qquad\quad \to 0 \\
\qquad\qquad Cons\ x\ xs \to x + sumList\ xs
\end{array}
$$

The first step in applying the automatic parallelization technique to *sumList* is to derive the functions for converting data of type *List Int* to and from a well-partitioned *join*-list containing data of type $List'$ using our automatic partitioning technique, the results of which are shown below:

$$
\begin{array}{lcl}
data\ List' & ::= & Nil' \\
 & \mid & Cons'\ Int \\
 & & \\
partition_{(List\ Int)} & = & partition \circ flatten_{(List\ Int)} \\
 & & \\
flatten_{(List\ Int)} & = & \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
 & & \qquad Nil \qquad\quad \to [Nil'] \\
 & & \qquad Cons\ x_1\ x_2 \to [Cons'\ x_1] +\!\!+ flatten_{(List\ Int)}\ x_2 \\
 & & \\
rebuild_{(List\ Int)} & = & fst \circ unflatten_{(List\ Int)} \circ rebuild \\
 & & \\
unflatten_{(List\ Int)} & = & \lambda xs.\textbf{case}\ xs\ \textbf{of} \\
 & & \qquad (x : xs) \to \textbf{case}\ x\ \textbf{of} \\
 & & \qquad\qquad Nil' \qquad\quad \to (Nil,\ xs) \\
 & & \qquad\qquad Cons'\ x_1 \to \textbf{case}\ unflatten_{(List\ Int)}\ xs\ \textbf{of} \\
 & & \qquad\qquad\qquad (x_2,\ xs_2) \to (Cons\ x_1\ x_2, xs_2)
\end{array}
$$

After generating these conversion functions, we compose $sumList$ with $rebuild_{(List\ Int)}$ and distill this composition, $\mathcal{D}[\![sumList \circ rebuild_{(List\ Int)}]\!]$, to generate an efficient sequential program equivalent to $sumList$ defined on a partitioned *join*-list. The resulting function, $sumList_{wp}$ is shown below:

$$
\begin{aligned}
sumList_{wp} = \lambda x.&\textbf{case}\ x\ \textbf{of}\\
&Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of}\\
&\qquad\qquad Nil' \rightarrow 0\\
&Join\ l\ r \quad\ \rightarrow sumList'_{wp}\ l\ (sumList_{wp}\ r)
\end{aligned}
$$

$$
\begin{aligned}
sumList'_{wp} = \lambda x\ n.&\textbf{case}\ x\ \textbf{of}\\
&Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of}\\
&\qquad\qquad Nil' \quad\ \rightarrow n\\
&\qquad\qquad Cons'\ x \rightarrow x + n\\
&Join\ l\ r \quad\ \rightarrow sumList'_{wp}\ l\ (sumList'_{wp}\ r\ n)
\end{aligned}
$$

After defining $sumList_{wp}$, we extract its parallelizable expressions, resulting in the definition of $sumList_{wp}$, shown below:

$$
\begin{aligned}
sumList_{wp} = \lambda x.&\textbf{case}\ x\ \textbf{of}\\
&Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of}\\
&\qquad\qquad Nil' \rightarrow 0\\
&Join\ l\ r \quad\ \rightarrow \textbf{let}\ l' = sumList'_{wp}\ l\\
&\qquad\qquad\qquad\quad r' = sumList_{wp}\ r\\
&\qquad\qquad\quad \textbf{in}\ \ l'\ r'
\end{aligned}
$$

$$
\begin{aligned}
sumList'_{wp} = \lambda x\ n.&\textbf{case}\ x\ \textbf{of}\\
&Singleton\ x \rightarrow \textbf{case}\ x\ \textbf{of}\\
&\qquad\qquad Nil' \quad\ \rightarrow n\\
&\qquad\qquad Cons'\ x \rightarrow x + n\\
&Join\ l\ r \quad\ \rightarrow \textbf{let}\ l' = sumList'_{wp}\ l\\
&\qquad\qquad\qquad\quad r' = sumList'_{wp}\ r\ n\\
&\qquad\qquad\quad \textbf{in}\ \ l'\ r'
\end{aligned}
$$

Finally, after defining $sumList_{wp}$, and extracting its parallelizable expressions, we apply $\mathcal{T}_p$ to $sumList_{wp}$ in order to explicitly parallelize its operations on well-partitioned *join*-lists. The resulting definition of $sumList_{par}$ is shown below:

$$
\begin{aligned}
sumList_{par} = \lambda x.&\textbf{case } x \textbf{ of} \\
&Singleton\ x \rightarrow \textbf{case } x \textbf{ of} \\
&\qquad\qquad Nil' \rightarrow 0 \\
&Join\ l\ r \quad\ \rightarrow \textbf{let } l' = sumList'_{par}\ l \\
&\qquad\qquad\qquad\quad r' = sumList_{par}\ r \\
&\qquad\qquad\quad \textbf{in } l'\ `par`\ r'\ `pseq`\ l'\ r'
\end{aligned}
$$

$$
\begin{aligned}
sumList'_{par} = \lambda x\ n.&\textbf{case } x \textbf{ of} \\
&Singleton\ x \rightarrow \textbf{case } x \textbf{ of} \\
&\qquad\qquad Nil' \quad\ \rightarrow n \\
&\qquad\qquad Cons'\ x \rightarrow x + n \\
&Join\ l\ r \quad\ \rightarrow \textbf{let } l' = sumList'_{par}\ l \\
&\qquad\qquad\qquad\quad r' = sumList'_{par}\ r\ n \\
&\qquad\qquad\quad \textbf{in } l'\ `par`\ r'\ `pseq`\ l'\ r'
\end{aligned}
$$

By making distillation aware of the definition of the + operator, it can derive the necessary associativity that allows for each child of a *Join* to be evaluated in parallel. It is worth noting that in the case of the above program, when evaluating the left child of a *Join* we create a partial application which can be evaluated in parallel with the evaluation of the right child. This partial application is equivalent to $(\lambda r.l + r)$, where r is the result of the evaluation of the right operand.

As both children are roughly equal in size, each parallel process created will have a roughly equal amount of work to do. In contrast, with respect to the original *sumList* defined on *cons*-lists, if the processing of both x and $sumList\ xs$ are performed in parallel, one process will have one element of the list to evaluate, while the other will have the remainder of the list to evaluate, which is undesirable.

7 Related Work

There are many existing works that aim to automate the parallelization process, however many of these works simply assume or require that they are provided with data for which there is a straight-forward conversion to a well-partitioned form. For example, list-homomorphisms [2–4, 10, 26] and their derivative works [5, 12, 13, 19–22, 30, 31] require that they are supplied with data in the form of a *cons*-list for which there is a simple conversion to a well-partitioned *join*-list. These techniques also require the specification of associative/distributive operators to be used as part of the parallelization process, which places more work in the hands of the developer.

Chin et al.'s [6, 7, 35] work on parallelization via context-preservation also makes use of *join*-lists as part of its parallelization process. Given a program, this technique derives two programs in *pre-parallel* form, which are then generalized. The resulting generalized function may contain undefined functions which can be defined using an inductive derivation. While such an approach is indeed powerful and does allow such complex functions as those with accumulating parameters,

nested recursion and conditional statements, it also has its drawbacks. One such drawback is that it requires that associativity and distributivity be specified for primitive functions by the developer. The technique is therefore *semi-automatic* and a fully automatic technique would be more desirable. While [35] presents an informal overview of the technique, a more concrete version was specified by Chin. et. al. in [6, 7]. However, these more formal versions are still only semi-automatic and are defined for a first-order language and require that the associativity/distributivity of operators be specified. This technique is also only applicable to *list-paramorphisms* [27] and while this encompasses a large number of function definitions, it is unrealistic to expect developers to define functions in this form. Further restrictions also exist in the transformation technique as the *context-preservation property* must hold in order to ensure the function can be parallelized.

While these techniques are certainly powerful, requiring that programs are only developed in terms of *cons*-lists is an unrealistic burden to place upon the developer, as is requiring that the associativity/distributivity of operators be specified. An important limitation to these techniques is that they are only applicable to lists, excluding the large class of programs that are defined on trees. One approach to parallelizing trees is that of Morihata et. al.'s [28] redefinition of the third homomorphism theorem [11] which is generalized to apply to trees. This approach makes use of zippers [23] to model the path from the root of a tree to an arbitrary leaf. While this approach presents an interesting approach to partitioning the data contained within a binary tree, the partitioning technique is quite complicated and relies upon zippers. It also presents no concrete methodology for generating zippers from binary-trees and assumes that the developer has provided such a function. It also requires that the user specify two functions in upward and downward form [28] which is quite restrictive so it is not realistic to expect a developer to define their programs in such a manner.

8 Conclusion

In conclusion, this paper has presented a novel, fully automatic parallelization technique for functional programs defined on any data-type. By defining a technique by which a developer can automatically parallelize programs, the difficulties associated with the parallelization process can be removed from the developer, who can continue developing software within the 'comfortable' sequential side of development and have equivalent parallel software derived automatically as needed.

Where existing automated parallelization techniques are restrictive with respect to the form of their input programs and the types they are defined on, the presented parallelization technique holds no such restrictions due to its use of our automatic data partitioning technique which converts data of any type into a well-partitioned form. To the best of the authors knowledge this is the first automated parallelization technique that is applicable to programs defined on any data-type.

One potential problem with our automatic parallelization technique is that it may get down to such a fine level of granularity of divide-and-conquer parallelism, that it is merely sparking a large number of trivial processes in parallel. This is obviously undesirable due to being wasteful of resources and potentially having a negative impact on efficiency due to the overheads associated with sparking parallel processes.

A solution to this problem is to control the level of granularity via the use of thresholding to govern the sparking of new parallel processes. Such an approach will also give the developer a measure of control over the parallelism obtained in the output program. Thresholding can be used to prevent the parallelization of *join*-lists whose size falls below a certain point. Research is ongoing to determine an optimal thresholding strategy for our automatically parallelized programs.

While our research is focused on divide-and-conquer task parallelization, it is also worth noting the potential for the system to be used as part of a data-parallel approach. If $partition_{(T\ T_1 \ldots T_g)}$ is redefined to generate a flattened list representation of the input data-structure, this could then be partitioned into chunks and distributed across a data-parallel architecture, such as a GPU. This would require a redefinition of $\mathcal{T}_p$ in order to support such an approach, which would implement the data-parallelism in the resulting program ensuring that the same function is applied to each chunk of data in parallel. Research is currently underway to extend the presented work to support automatic partitioning to enable GPU parallelization [24].

Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 10/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre.

References

1. L. Augustsson. Compiling Pattern Matching. *Functional Programming Languages and Computer Architecture*, 1985.
2. R. Backhouse. An Exploration of the Bird-Meertens Formalism. Technical report, In STOP Summer School on Constructive Algorithmics, Abeland, 1989.
3. R. Bird. Constructive Functional Programming. *STOP Summer School on Constructive Algorithmics*, 1989.
4. R. S. Bird. An Introduction to the Theory of Lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42, New York, NY, USA, 1987. Springer-Verlag New York, Inc.
5. G. E. Blelloch. Scans as Primitive Operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.
6. W.-N. Chin, S.-C. Khoo, Z. Hu, and M. Takeichi. Deriving Parallel Codes via Invariants. In J. Palsberg, editor, *Static Analysis*, volume 1824 of *Lecture Notes in Computer Science*, pages 75–94. Springer Berlin Heidelberg, 2000.
7. W. N. Chin, A. Takano, Z. Hu, W. ngan Chin, A. Takano, and Z. Hu. Parallelization via Context Preservation. In *In IEEE Intl Conference on Computer Languages*, pages 153–162. IEEE CS Press, 1998.

8. N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. 1990.
9. M. Dever and G. W. Hamilton. Automatically Partitioning Data to Facilitate the Parallelization of Functional Programs. *Proceedings of the Eight International Andrei Ershov Memorial Conference*, July 2014.
10. M. Fokkinga. A gentle introduction to category theory — the calculational approach. In *Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics*, pages 1–72. University of Utrecht, September 1992.
11. J. Gibbons. The Third Homomorphism Theorem. *Journal of Functional Programming*, 6(4):657–665, 1996. Earlier version appeared in C. B. Jay, editor, *Computing: The Australian Theory Seminar*, Sydney, December 1994, p. 62–69.
12. S. Gorlatch. Systematic Efficient Parallelization of Scan and Other List Homomorphisms. In *In Annual European Conference on Parallel Processing, LNCS 1124*, pages 401–408. Springer-Verlag, 1996.
13. S. Gorlatch. Systematic Extraction and Implementation of Divide-and-Conquer Parallelism. In *Programming languages: Implementation, Logics and Programs, Lecture Notes in Computer Science 1140*, pages 274–288. Springer-Verlag, 1996.
14. G. Hamilton and N. Jones. Distillation and Labelled Transition Systems. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation*, pages 15–24, January 2012.
15. G. Hamilton and N. Jones. Proving the Correctness of Unfold/Fold Program Transformations using Bisimulation. *Lecture Notes in Computer Science*, 7162:153–169, 2012.
16. G. W. Hamilton. Distillation: Extracting the Essence of Programs. *Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation*, 2007.
17. G. W. Hamilton. Extracting the Essence of Distillation. *Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics*, 2009.
18. G. W. Hamilton and G. Mendel-Gleason. A Graph-Based Definition of Distillation. *Proceedings of the Second International Workshop on Metacomputation in Russia*, 2010.
19. Z. Hu, H. Iwasaki, and M. Takechi. Formal Derivation of Efficient Parallel Programs by Construction of List Homomorphisms. *ACM Trans. Program. Lang. Syst.*, 19(3):444–461, May 1997.
20. Z. Hu, M. Takeichi, and W.-N. Chin. Parallelization in Calculational Forms. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 316–328, New York, NY, USA, 1998. ACM.
21. Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating Efficient Parallel Programs. In *In 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 85–94, 1999.
22. Z. Hu, T. Yokoyama, and M. Takeichi. Program Optimizations and Transformations in Calculation Form. In *Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 144–168, Berlin, Heidelberg, 2006. Springer-Verlag.
23. G. Huet. The Zipper. *J. Funct. Program.*, 7(5):549–554, Sept. 1997.
24. V. Kannan and G. W. Hamilton. Distillation to Extract Data Parallel Computations. *Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation*, July 2014.

25. H.-W. Loidl, P. W. Trinder, K. Hammond, A. Al Zain, and C. A. Baker-Finch. Semi-Explicit Parallel Programming in a Purely Functional Style: GpH. In M. Alexander and B. Gardner, editors, *Process Algebra for Parallel and Distributed Processing: Algebraic Languages in Specification-Based Software Development*, pages 47–76. Chapman and Hall, Dec. 2008.
26. G. Malcolm. Homomorphisms and Promotability. In *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, pages 335–347, London, UK, 1989. Springer-Verlag.
27. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
28. A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. The Third Homomorphism Theorem on Trees: Downward & Upward lead to Divide-and-Conquer. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 177–185, New York, NY, USA, 2009. ACM.
29. D. Skillicorn. *Foundations of Parallel Programming*. Cambridge International Series on Parallel Computation. Cambridge University Press, 2005.
30. D. B. Skillicorn. Architecture-Independent Parallel Computation. *Computer*, 23:38–50, December 1990.
31. D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In *Software for Parallel Computation, volume 106 of NATO ASI Series F*, pages 120–133. Springer-Verlag, 1993.
32. M. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. *International Logic Programming Symposium*, pages 465–479, 1995.
33. M. Sørensen, R. Glück, and N. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 1(1), January 1993.
34. M. H. Sørensen. Turchin's Supercompiler Revisited - An Operational Theory of Positive Information Propagation, 1996.
35. Y. M. Teo, W.-N. Chin, and S. H. Tan. Deriving Efficient Parallel Programs for Complex Recurrences. In *Proceedings of the second international symposium on Parallel symbolic computation*, PASCO '97, pages 101–110, New York, NY, USA, 1997. ACM.
36. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.
37. P. Wadler. Efficient Compilation of Pattern Matching. In S. P. Jones, editor, *The Implementation of Functional Programming Languages.*, pages 78–103. Prentice-Hall, 1987.
38. P. Wadler. Deforestation: Transforming Programs to Eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

Inductive Prover Based on Equality Saturation for a Lazy Functional Language*
(Extended Version)

Sergei A. Grechanik

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
sergei.grechanik@gmail.com

Abstract. The present paper shows how the idea of equality saturation can be used to build an inductive prover for a non-total first-order lazy functional language. We adapt equality saturation approach to a functional language by using transformations borrowed from supercompilation. A special transformation called merging by bisimilarity is used to perform proof by induction of equivalence between nodes of the E-graph. Equalities proved this way are just added to the E-graph. We also experimentally compare our prover with HOSC, HipSpec and Zeno.

1 Introduction

Equality saturation [23] is a method of program transformation that uses a compact representation of multiple versions of the program being transformed. This representation is based on E-graphs (graphs whose nodes are joined into equivalence classes [7, 18]) and allows us to represent a set of equivalent programs, consuming exponentially less memory than representing it as a plain set. Equality saturation consists in enlarging this set of programs by applying certain axioms to the E-graph until there's no axiom to apply or the limit of axiom applications is reached. The axioms are applied non-destructively, i.e. they only add information to the E-graph (by adding nodes, edges and equivalences).

Equality saturation has several applications. It can be used for program optimization – in this case after the process of equality saturation is finished, a single program should be extracted from the E-graph. It can also be used for proving program equivalence (e.g. for translation validation [22]) – in this case program extraction is not needed.

In the original papers equality saturation is applied to imperative languages, namely Java bytecode and LLVM (although the E-graph-based intermediate representation used there, called E-PEG, is essentially functional). In this paper we describe how equality saturation can be applied to the task of proving equivalence

* Supported by Russian Foundation for Basic Research grant No. 12-01-00972-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

of functions written in a lazy functional language, which is important for proving algebraic properties like monad laws or some laws concerning natural numbers and lists. We do this mainly by borrowing transformations from supercompilation [21, 24]. Since many properties require proof by induction, we introduce a special transformation called merging by bisimilarity which essentially proves by induction that two terms are equivalent. This transformation may be applied repeatedly, which gives an effect of discovering and proving lemmas needed for the main goal.

Unlike tools such as HipSpec [5] and Zeno [20], we don't instantiate the induction scheme, but instead check the correctness of the proof graph similarly to Agda and Foetus [3, 4]. We also fully support infinite data structures and partial values, and we don't assume totality. As we'll show, proving properties that hold only in total setting is still possible with our tool by enabling some additional transformations, but it's not very efficient.

The main contributions of this paper are: 1) we apply the equality saturation approach to a lazy functional language; 2) we propose to merge classes of the E-graph even if they represent functions equal only up to argument permutation; 3) we articulate the merging by bisimilarity transformation.

The paper is organized as follows. In Section 2 we briefly describe equality saturation and how functional programs and sets of functional programs can be represented by E-graphs. Then in Section 3 we discuss basic transformations which we apply to the E-graph. Section 4 deals with the merging by bisimilarity transformation. Section 5 discusses the order of transformation application. In Section 6 we consider a simple example. In Section 7 we present experimental evaluation of our prover. Section 8 discusses related work and Section 9 concludes the paper.

The source code of our experimental prover can be found on GitHub [1].

2 Programs and E-graphs

An E-graph is a graph enriched with information about equivalence of its nodes by means of splitting them into equivalence classes. In our case, an E-graph essentially represents a set of (possibly recursive) terms and a set of equalities on them, closed under reflexivity, transitivity and symmetry. If we use the congruence closure algorithm [18], then the set of equalities will also be closed under congruence. The E-graph representation is very efficient and often used for solving the problem of term equivalence.

If we have some axioms about our terms, we can also apply them to the E-graph, thus deducing new equalities from the ones already present in E-graph (which in its turn may lead to more axiom application opportunities). This is what equality saturation basically is. So, the process of solving the problem of function/program equivalence using equality saturation is as follows:

1. Convert both function definitions to E-graphs and put both of them into a single E-graph.

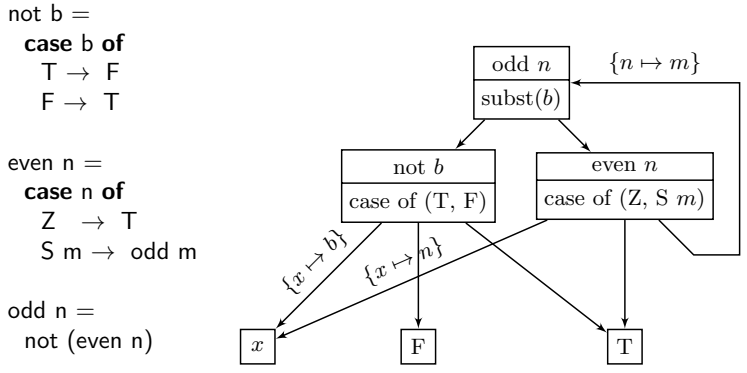


Fig. 1: A program and its graph representation

2. Transform the E-graph using some axioms (transformations) until the target terms are in the same equivalence class or no more axioms are applicable. This process is called saturation.

In pure equality saturation approach axioms are applied non-destructively and result only in adding new nodes and edges, and merging of equivalence classes, but in our prover we apply some axioms destructively, removing some nodes and edges. This makes the result of the saturation dependent on the order of axiom application, so we restrict it to breadth-first order (see Section 5 for more details). This deviation is essential for performance reasons.

In this paper we will use a lazy first-order untyped subset of Haskell (in our implementation higher-order functions are dealt with by defunctionalization). To illustrate how programs are mapped into graphs, let's consider the program in Figure 1a. This program can be naturally represented as a graph, as shown in Figure 1b. Each node represents a basic language construct (pattern matching, constructor, variable, or explicit substitution – we'll explain them in Section 2.1). If a node corresponds to some named function, its name is written in the top part of it. Some nodes are introduced to split complex expressions into basic constructs and don't correspond to any named functions. Recursion is simply represented by cycles. Some nodes are shared (in this example these are the variable x and the constructor T). Sharing is very important since it is one of the things that enable compactness of the representation.

Some of the edges are labeled with renamings. Actually, all edges are labeled with renamings, but identity renamings are not drawn. These renamings are very important – without them we would need a separate node for each variable, and we couldn't merge nodes representing the same function modulo renaming, which would increase space consumption (such functions often appear during transformation). Merging up to renaming will be discussed in Section 2.2.

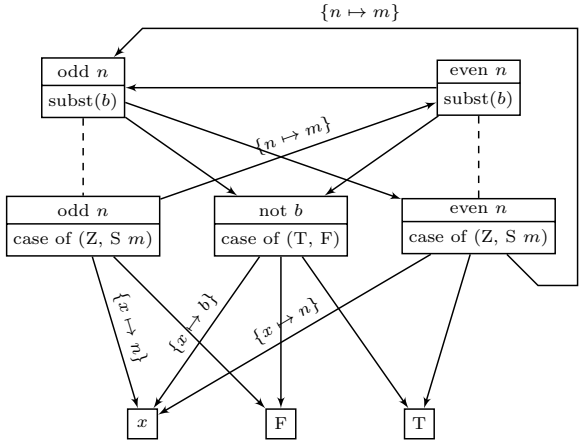


Fig. 2: E-graph representing functions even and odd

Note also that we use two methods of representing function calls. If all the arguments are *distinct* variables, then we can simply use a renaming (the function odd is called this way). If the arguments are more complex, then we use explicit substitution [2], which is very similar to function call but has more fine-grained reduction rules. We can use explicit substitutions even if the arguments are distinct variables, but it's more expensive than using renamings (and actually we have an axiom to transform such explicit substitutions to renamings). Note that we require an explicit substitution to bind *all* variables of the expression being substituted.

The same way graphs naturally correspond to programs, E-graphs naturally correspond to programs with multiple function definitions. Consider the following "nondeterministic" program:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
odd n = case n of { Z → F; S m → even m }

odd n = not (even n)
even n = not (odd n)
```

This program contains multiple definitions of the functions even and odd, but all the definitions are actually equivalent. This program can also be represented as a graph, but there will be multiple nodes corresponding to functions even and odd. If we add the information that nodes corresponding to the same function are in the same equivalence class, we get an E-graph. The E-graph corresponding to the above program is shown in Figure 2. Nodes of equivalent functions are connected with dashed lines, meaning that these nodes are in the same class of

equivalence. As can be seen, the drawing is messy and it's hard to understand what's going on there, so we'll mostly use textual form to describe E-graphs.

E-graphs are also useful for representing compactly sets of equivalent programs. Indeed, we can extract individual programs from an E-graph or a nondeterministic program by choosing a single node for each equivalence class, or in other words, a single definition for each function. However, we cannot pick the definitions arbitrarily. For example, the following program isn't equivalent to the one above:

```
not b = case b of { T → F; F → T }

odd n = not (even n)
even n = not (odd n)
```

This problem should be taken into account not only when performing program extraction, but also during certain complex transformations like merging by bisimilarity which we will discuss in Section 4.

2.1 Node labels

In this section we'll discuss how node labels correspond to language constructs.

First of all, each node of an E-graph is a member of some equivalence class. We will use symbols $f, g, h, \ldots$ to denote nodes as well as corresponding functions. Each node has a label $L(f)$ and a set of input variables $V(f)$ (in the implementation variables are numbered, but in this paper we treat them as named). $V(f)$ may decrease with graph evolution, and it should be kept up to date because we need $V(f)$ to perform some transformations (keeping it up to date is beyond the scope of this paper). Each edge of an E-graph is labeled with an injective renaming, its domain being the set of input variables of the edge's destination node. We will use the notation $f = L \to \theta_1 g_1, \ldots, \theta_n g_n$ to describe a node f with a label L and outgoing edges with renamings θ_i and destinations g_i. We will write $f \cong g$ to denote that f and g are from the same equivalence class.

There are only four kinds of node labels. We give a brief description for each of them and some code examples:

- $f = x$. (Variable / identity function). We use the convention that the identity function always takes the variable x, and if we need some other variable, we adjust it with a renaming. Code example: `f x = x`
- $f = \mathrm{subst}(x_1, \ldots, x_n) \to \xi h, \theta_1 g_1, \ldots, \theta_n g_n$. (Explicit substitution / function call / let expression). An explicit substitution substitutes values $\theta_i g_i$ for the variables x_i in ξh. We require it to bind *all* the variables of ξh. Explicit substitution nodes usually correspond to function calls:

  ```
  f x y = h (g1 x) (g2 y) (g3 x y)
  ```

  They may also correspond to non-recursive let expressions, or lambda abstractions immediately applied to the required number of arguments:

$$\begin{aligned}\mathsf{f\ x\ y} &= \textbf{let}\ \{\ \mathsf{u = g1\ x;\ v = g2\ y;\ w = g3\ x\ y}\ \}\ \textbf{in}\ \mathsf{h\ u\ v\ w}\\ &= (\lambda\ \mathsf{u\ v\ w\ .\ h\ u\ v\ w})\ (\mathsf{g1\ x})\ (\mathsf{g2\ y})\ (\mathsf{g3\ x\ y})\end{aligned}$$

But to describe E-graph transformations we will use the following non-standard (but hopefully more readable) postfix notation:

$$\mathsf{f\ x\ y} = \mathsf{h\ u\ v\ w}\ \{\ \mathsf{u = g1\ x,\ v = g2\ y,\ w = g3\ x\ y}\ \}$$

– $f = C \to \theta_1 g_1, \ldots, \theta_n g_n$. (Constructor). Code example:

$$\mathsf{f\ x\ y} = \mathsf{C\ (g1\ x)\ (g2\ y)\ (g3\ x\ y)}$$

– $f = \text{caseof}(C_1\overline{x_1}, \ldots, C_n\overline{x_n}) \to \xi h, \theta_1 g_1, \ldots, \theta_n g_n$. (Pattern matching). This label is parametrized with a list of patterns, each pattern is a constructor name and a list of variables. The corresponding case bodies ($\theta_i g_i$) don't have to use all the variables from the pattern. ξh represents the expression being scrutinized. Code example:

$$\begin{aligned}\mathsf{f\ x\ y} = \textbf{case}\ \mathsf{h\ x}\ \textbf{of}&\\ \mathsf{S\ n} &\to\ \mathsf{g1\ y\ n}\\ \mathsf{Z} &\to\ \mathsf{g2\ x}\end{aligned}$$

We will also need an operation of adjusting a node with a renaming. Consider a node $f = L \to \theta_1 g_1, \ldots, \theta_n g_n$ and a renaming ξ. Suppose, we want to create a function $f' = \xi f$ (f' is f with parameters renamed). We can do this by adjusting outgoing edges of f with ξ (unless $f = x$ in which case it doesn't have outgoing edges). We will use the following notation for this operation:

$$f' = \xi(L \to \theta_1 g_1, \ldots, \theta_n g_n)$$

The operation is defined as follows:

$$\begin{aligned}&\xi(C \to \theta_1 g_1, \ldots, \theta_n g_n) = C \to (\xi \circ \theta_1) g_1, \ldots, (\xi \circ \theta_n) g_n\\ &\xi(\text{subst}(\ldots) \to \zeta h, \theta_1 g_1, \ldots, \theta_n g_n) =\\ &\qquad \text{subst}(\ldots) \to \zeta h, (\xi \circ \theta_1) g_1, \ldots, (\xi \circ \theta_n) g_n\\ &\xi(\text{caseof}(\ldots) \to \zeta h, \theta_1 g_1, \ldots, \theta_n g_n) =\\ &\qquad \text{caseof}(\ldots) \to (\xi \circ \zeta) h, (\xi'_1 \circ \theta_1) g_1, \ldots, (\xi'_n \circ \theta_n) g_n\end{aligned}$$

In the last case each ξ'_i maps the variables bound by the ith pattern to themselves and works as ξ on all the other variables.

2.2 Merging

One of the basic operations of the E-graph is merging of equivalence classes. Usually it is done after applying axioms that result in adding new equalities between nodes. In the case of simple equalities like $f = g$ we should simply merge the corresponding equivalence classes. But we also want to merge functions which are equal only up to some renaming, so should take into account equalities

of the form $f = \theta g$ where θ is some non-identity renaming. In this case we should first adjust renamings on edges so that the equation becomes of the form $f = g$ and then proceed as usual.

Consider the equation $f = \theta g$. Let's assume that g is not a variable node (x) and it's not in the same equivalence class with a variable node (otherwise we can rewrite the equation as $g = \theta^{-1} f$, and if they both were equal to a variable node, then our E-graph would be self-contradictory). Now for each node h in the same equivalence class with g (including g) we should perform the following:

1. Adjust the outgoing edges of h with θ using previously described node adjustment operation.
2. For each edge incoming into h replace its renaming, say, ξ, with a renaming $\xi \circ \theta^{-1}$

After the adjustment the equation becomes $f = g$ and we can merge the equivalence classes.

Note that this procedure works if f and g aren't in the same equivalence classes. If they are, then the equation looks like $f = \theta f$ and should be modelled with an explicit substitution.

3 Axioms

3.1 Congruence

The most common cause of equivalence class merging is equivalence by congruence, that is if we know that $a = f(b)$, $c = f(d)$ and $b = d$, then we can infer that $a = c$. Note that usually this kind of merging is not explicitly formulated as an axiom, but we prefer to do it explicitly for uniformity. Also, in our case the axiom should take into account that we want to detect equivalences up to some renaming. Here is the axiom written as an inference rule, we will later refer to it as (cong):

$$\frac{f = L \to \theta_1 h_1, \ldots, \theta_n h_n \qquad \exists \xi : g = \xi(L \to \theta_1 k_1, \ldots, \theta_n k_n) \qquad \forall i \; h_i \cong k_i}{g = \xi f}$$

It says that if we have a node f and a node g that is equivalent to f adjusted with some renaming ξ, then we can add the equality $g = \xi f$ to the E-graph. This axiom is advantageous to apply as early as possible since it results in merging of equivalence classes, which reduces duplication and gives more opportunities for applying axioms. Also note that to make the search for the appropriate ξ faster, it is beneficial to represent nodes in normal form:

$$f = \zeta(L \to \theta_1 g_1, \ldots, \theta_n g_n)$$

Where θ_i are as close to identity renamings as possible, so to find ξ we should just compare the ζ's.

3.2 Injectivity

This axiom may be seen as something like "inverse congruence". If we know that $a = f(b)$, $c = f(d)$ and $a = c$ and f is injective, then $b = d$. Of course, we could achieve the same effect by adding the equalities $b = f^{-1}(a)$ and $d = f^{-1}(c)$ to the E-graph and then using congruence, but we prefer a separate axiom for performance reasons. We will call it (inj):

$$\frac{f = L \to \xi(\theta_1 h_1, \dots \theta_n h_n) \qquad g = L \to \xi(\zeta_1 k_1, \dots \zeta_n k_n) \qquad f \cong g \qquad L \text{ is inj}}{\forall i \,.\, h_i = \theta_i^{-1} \zeta_i k_i}$$

"L is inj" means that L is either a constructor, or a case-of that scrutinizes a variable (i.e. $\theta_1 = \zeta_1$ and $h_1 = k_1 = x$) such that none of the $\theta_2 h_2, \dots, \theta_n h_n$, $\zeta_2 k_2, \dots, \zeta_n k_n$ uses this variable (in other words, positive information is propagated). This axiom is also advantageous to apply as early as possible.

3.3 Semantics of explicit substitutions

In this and the next sections we will write axioms in a less strict but more human-readable form. A rewriting rule $E_1 \mapsto E_2$ means that if we have a node f_1 representing the expression E_1, then we can add an equality $f_1 = f_2$ to the E-graph where f_2 is the node representing E_2 (which should also be added to the E-graph unless it's already there). We use the compact postfix notation to express explicit substitutions. We use letters $e, f, g, h, \dots$ to represent nodes whose structure doesn't matter. We sometimes write them applied to variables they use ($f\ x\ y$), but if variables don't really matter, we omit them. Note that the presented rules can be generalized to the case when pattern matchings have arbitrary number of branches and functions take arbitrary number of arguments, we just use minimal illustrative examples for the sake of readability.

In Figure 3 four axioms of explicit substitutions [2] are shown. All of them basically describe how to evaluate a node if it is an explicit substitution (using call-by-name strategy). The answer is to push the substitution down (the last three rules) until we reach a variable where we can just perform the actual substitution (the first rule, (subst-id)). The appropriate rule depends on the node we choose as the leftmost child of our substitution node – there are four kinds of nodes, so there are four rules.

(subst-id)	$x\ \{x = g\} \ \mapsto\ g$
(subst-subst)	$f\ x\ \{x = g\ y\}\ \{y = h\} \ \mapsto\ f\ x\ \{x = g\ y\ \{y = h\}\}$
(subst-constr)	$C\ (f\ x)\ (g\ x)\ \{x = h\} \ \mapsto\ C\ (f\ x\ \{x = h\})\ (g\ x\ \{x = h\})$
(subst-case-of)	$(\textbf{case}\ f\ x\ \textbf{of}\ C\ y \to g\ x\ y)\ \{x = h\} \ \mapsto$ $\quad \textbf{case}\ f\ x\ \{x = h\}\ \textbf{of}\ C\ y \to g\ x\ y\ \{x = h,\ y = y\}$

Fig. 3: Axioms of explicit substitutions

(case-of-constr) $(\textbf{case}\ \ C\ e\ \ \textbf{of}\ C\ y \to f\ x\ y) \ \mapsto$
$\quad f\ x\ y\ \{x = x,\ y = e\}$

(case-of-case-of) $(\textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ C_1\ y \to g)\ \textbf{of}\ C_2\ z \to h) \ \mapsto$
$\quad \textbf{case}\ e\ \textbf{of}\ C_1\ y \to (\textbf{case}\ g\ \textbf{of}\ C_2\ z \to h)$

(case-of-id) $(\textbf{case}\ x\ \textbf{of}\ C\ y\ z \to f\ x\ y\ z) \ \mapsto$
$\quad \textbf{case}\ x\ \textbf{of}\ C\ y\ z \to f\ x\ y\ z\ \{x = (C\ y\ z),\ y = y,\ z = z\}$

(case-of-transpose) $\textbf{case}\ h\ \textbf{of}\ \{$
$\quad C_1\ x \to \textbf{case}\ z\ \textbf{of}\ D\ v \to f\ v\ x;$
$\quad C_2\ y \to \textbf{case}\ z\ \textbf{of}\ D\ v \to g\ v\ y;$
$\} \ \mapsto$
$\quad \textbf{case}\ z\ \textbf{of}\ D\ v \to$
$\quad\quad \textbf{case}\ h\ \textbf{of}\ \{$
$\quad\quad\quad C_1\ x \to f\ v\ x;$
$\quad\quad\quad C_2\ y \to g\ v\ y;$
$\quad\quad \}$

Fig. 4: Axioms of pattern matching

Usually substitution in the body of a function is performed as an indivisible operation, but this kind of transformation would be too global for an E-graph, so we use explicit substitutions to break it down.

There are two more rather technical but nonetheless important axioms concerning substitution. The first one is elimination of unused variable bindings:

(subst-unused) $\quad f\ x\ y\ \{x = g,\ y = h,\ z = k\} \ \mapsto\ f\ x\ y\ \{x = g,\ y = h\}$

When this axiom is applied *destructively* (i.e. the original node is removed), it considerably simplifies the E-graph. This axiom is the reason why we need the information about used variables in every node.

The second axiom is conversion from a substitution that substitutes variables for variables to a renaming:

(subst-to-renaming) $\quad f\ x\ y\ \{x = y,\ y = z\} \ \mapsto\ f\ y\ z$

This axiom requires the original substitution to be injective. Note also that application of this axiom results in merging of the equivalence classes corresponding to the node representing the substitution and the node f, so if they are already in the same class, this axiom is inapplicable. We also apply this axiom destructively.

3.4 Semantics of pattern matching

The axioms concerning pattern matching are shown in Figure 4. The first of them, (case-of-constr), is essentially a reduction rule: if the scrutinee is an expression starting with a constructor, then we just substitute appropriate subexpressions into the corresponding case branch.

The next two axioms came from supercompilation [21,24]. They tell us what to do when we get stuck during computation because of missing information (i.e. a variable). The axiom (case-of-case-of) says that if we have a pattern matching that scrutinizes the result of another pattern matching, then we can pull the inner pattern matching out. The axiom (case-of-id) is responsible for positive information propagation: if a case branch uses the variable being scrutinized, then it can be replaced with its reconstruction in terms of the pattern variables.

The last axiom, (case-of-transpose), says that we can swap two consecutive pattern matchings. This transformation is not performed by supercompilers and is actually rarely useful in a non-total language.

3.5 Totality

If we assume that our language is total, then we can use all the axioms mentioned above and also some more axioms that don't hold in the presence of bottoms. Although proving equivalence of total functions is not our main goal, our implementation has a totality mode which enables three additional axioms from Figure 5.

(case-of-constr-total)

```
case h of {
    C1 x → D (f x);
    C2 y → D (f y);
} ↦
    D (case h of {
        C1 x → f x;
        C2 y → f y;
    })
```

(case-of-transpose-total)

```
case h of {
    C1 x → case z of D v → f v x z;
    C2 y → g y;
} ↦
    case z of D v →
        case h of {
            C1 x → f v x z;
            C2 y → g y;
        }
```

(useless-case-of-total)

```
case h of {
    C1 x → f;
    C2 y → f;
} ↦
    f
```

Fig. 5: Additional axioms of pattern matching in total setting

The axiom (case-of-constr-total) lifts equal constructors from case branches. If E could be bottom, then it wouldn't be correct to do that (actually the axiom makes the function lazier). Note that constructors may have arbitrary arity.

The axiom (case-of-transpose-total) is a variation of the axiom (case-of-transpose). It may swap the pattern matchings even if inner pattern matching is not performed in some branches of the outer one.

The axiom (useless-case-of-total) removes an unnecessary pattern matching when all of its branches are equal (they can't use pattern variables (x and y in this case) though).

3.6 On correctness and completeness

Correctness of the mentioned axioms can be easily proved if we fix an appropriate semantics for the language.

Since the problem of function equivalence is undecidable, no finite set of axioms can be complete, but we can compare our set of axioms with other transformers. If we take all the axioms from Figure 3 and the axiom (case-of-constr) from Figure 4, we will be able to perform interpretation. If we also add axioms (case-of-case-of) and (case-of-id) from Figure 4, then we will be able to perform driving (interpretation with incomplete information, i.e. with free variables). Given infinite time, driving allows us to build a perfect tree for a function (which is something like an infinite tabular representation of a function). Perfect trees consist of constructors, variables and pattern matchings on variables with positive information fully propagated. Perfect trees aren't unique, some functions may have multiple perfect trees, and the (case-of-transpose) axiom is used to mitigate this problem (although not eliminate it completely). The totality axioms (Figure 5) equate even more perfect trees by rearranging their nodes.

Of course, we could add more axioms. For example, in the total case we could use an axiom to lift pattern matchings through explicit substitutions, not only other pattern matchings. Or we could add generalizations which are used in supercompilers. All of this would make our equality saturator more powerful but at the cost of lower performance. So this is all about balance. As for the generalization, in the case of equality saturation expressions are already in generalized state, and we can transform any subexpression of any expression. It's not a complete solution to the problem of generalization since it's only equal to peeling the outer function call from an expression, but it still allows us to solve many examples that can't be solved with driving alone.

Another issue is proof by induction or coinduction. In supercompilers coinduction is implicitly applied when we build a residual program. Higher level supercompilers and inductive provers are able to apply (co)induction several times, thus proving the lemmas needed to prove the target proposition. In our equality saturator (co)induction is implemented as a special transformation called merging by bisimilarity which is discussed in Section 4.

3.7 Axioms applied destructively

We apply some transformations destructively, i.e. remove the original nodes and edges that triggered the transformation. It is a deviation from pure equality saturation approach, but it is a necessary one. Currently the transformations we apply destructively are (subst-id), (subst-unused), (subst-to-renaming), and (case-of-constr). We have tried to switch on and off their destructivity. Turned out that non-destructive (case-of-constr) leads to a lot of failures on our test suite (due to timeouts), but helps to pass one of the tests that cannot be passed when it's destructive (which is expected: non-destructive transformations are strictly more powerful when there is no time limit). Non-destructive (subst-unused) has a similar effect: it helps to pass two different tests, but at the price of failing several other tests. At last, non-destructivity of (subst-id) and (subst-to-renaming) doesn't impede the ability of our tool to pass tests from our test suite but when either of them is applied non-destructively, our tool becomes about 15% slower. We also tried to make *all* the mentioned transformations non-destructive, which rendered our tool completely unusable because of combinatorial explosion of the E-graph, which substantiates the importance of at least some destructivity.

4 Merging by bisimilarity

The axiom of congruence can merge two functions into one equivalence class if they have the same tree representation. But if their definitions involve separate (but equal) cycles, then the congruence axiom becomes useless. Consider the following two functions:

```
f = S f
g = S g
```

If they aren't in the same equivalence class in the first place, none of the already mentioned axioms can help us equate them. Here we need some transformation that is aware of recursion. Note that in the original implementation of equality saturation called Peggy [23] there is such a transformation that merges θ-nodes, but it doesn't seem to be published anywhere and it is much less powerful than the one described here.

The general idea of this kind of transformation is to find two bisimilar subgraphs growing from the two given nodes from different equivalence classes and merge these equivalence classes if the subgraphs have been found. Note though that not every subgraph is suitable. Consider the following nondeterministic program:

```
f x = C;        g x = D
f x = f (f x); g x = g (g x)
```

The functions f and g are different but they both are idempotent, which is stated by the additional definitions, so we have two equal closed subgraphs "defining" the functions:

```
f x = f (f x)
g x = g (g x)
```

Of course, we cannot use subgraphs like these to decide whether two functions are equal, because they don't really define the functions, they just state that they have the property of idempotence. So we need a condition that guarantees that there is (semantically) only one function satisfying the subgraph.

In our implementation we employ the algorithm used in Agda and Foetus to check if a recursive function definition is structural or guarded [4]. These conditions are usually used in total languages to ensure termination and productivity, but we believe that they can be used to guarantee *uniqueness* of the function satisfying a definition in a non-total language with infinite and partial values, although a proof of this claim is left for future work. Informally speaking, in this case guarded recursion guarantees that there is data output between two consecutive recursive function calls, and structural recursion guarantees that there is data input between them (i.e. a pattern matching on a variable that hasn't been scrutinized before). It's not enough for function totality since the input data may be infinite, but it defines the behaviour of the function on each input, thus guaranteeing it to be unique.

Note that there is a subtle difference between subgraphs that may have multiple fixed points and subgraphs that have a single fixed point equal to bottom. Consider the following function "definition":

```
f x = f x
```

The least fixed point interpretation of this function is bottom. But there are other fixed points (actually, any one-argument function is a fixed point of this definition). Now consider the following function:

```
f x = eat infinity
infinity = S infinity
eat x = case x of { S y → eat y }
```

The definition of `infinity` is guardedly recursive, and the definition of `eat` is structurally recursive. The least fixed point interpretation of the function `f` is still bottom but now it is guaranteed to be the only interpretation.

Of course, this method of ensuring uniqueness may reject some subgraphs having a single fixed point, because the problem is undecidable in general. Note also that this is not the only possible method of ensuring uniqueness. For example, we could use ticks [19] as in two-level supercompilation [15]. Ticks are similar to constructors but have slightly different properties, in particular they cannot be detected by pattern matching. Tick transformations could be encoded as axioms for equality saturation.

4.1 Algorithm description

In this subsection we'll describe the algorithm that we use to figure out if two nodes have two bisimilar subgraphs growing from them and meeting the uniqueness condition. First of all, the problem of finding two bisimilar subgraphs is a

```
function MERGE-BY-BISIMILARITY(m, n)
    if BISIMILAR?(m, n, ∅) then MERGE(m, n)

function BISIMILAR?(m, n, history)
    if m ≅ n then return true
    else if ∃(m′, n′) ∈ history : m′ ≅ m ∧ n′ ≅ n then
        if the uniqueness condition is met then
            return true
        else
            return false
    else if m and n are incompatible then return false
    else
        for (m′, n′) : m′ ≅ m ∧ n′ ≅ n ∧ label(m′) = label(n′) do
            children_pairs = zip(children(m′), children(n′))
            if length(children(m′)) = length(children(n′))
                and ∀ (m″, n″) ∈ children_pairs
                        BISIMILAR?(m″, n″, {(m′, n′)} ∪ history)  then
                return true
        return false
```

Fig. 6: Merging by bisimilarity

variation of the subgraph bisimulation problem which is NP-complete [8]. In certain places we trade completeness for performance (so sometimes our algorithm fails to find the subgraphs when they exist), but merging by bisimilarity is still one of the biggest performance issues in our experimental implementation. The merging by bisimilarity algorithm that we use is outlined in Figure 6. It checks (using the function BISIMILAR?) if there are two bisimilar subgraphs meeting the uniqueness condition, and if there are, merges the equivalence classes of the nodes. Checking for bisimilarity essentially consists in simultaneous depth-first traversal of the E-graph from the given nodes. This process resembles supercompilation.

The function BISIMILAR? works as follows. If the two nodes are equal, then they are bisimilar and we return true. If we encounter a previously visited pair of nodes (up to $\cong$), we check if the uniqueness condition holds, and if it does, we return true (this case corresponds to folding in supercompilation), and otherwise we stop trying and return false (this case doesn't guarantee that there's no bisimulation, but we do it for efficiency). This case also ensures termination of the algorithm since the number of nodes in the E-graph is finite, and they are bound to repeat at some point. Note that some kinds of uniqueness conditions have to be checked after the whole bisimulation is known (and the guardedness and structurality checker is of this kind since it needs to know all the recursive call sites). In this case it is still advantageous to check some prerequisite condition while folding, which may be not enough to guarantee correctness, but enough to filter out obviously incorrect graphs.

If neither of the two previous cases is applicable, we check if the two nodes are at least compatible (again, for efficiency reasons, we could do without it in theory). That means that there are no nodes equal to them that have incompatible labels, like different constructors or a constructor and a pattern matching on a variable. If the nodes are compatible, we go on and check all pairs of nodes equivalent to the original ones. If there is a pair of nodes such that their children are bisimilar, then the original pair is bisimilar.

We can extract the actual bisimulation by collecting all the node pairs on which we return true (we will call this relation R). We can also extract the two bisimilar subgraphs (actually, E-subgraphs) by taking the corresponding elements of these pairs (either left or right) and outgoing edges for those nodes that occurred bisimilar to some other nodes because their children were bisimilar. Indeed, the roots of these two subgraphs are in R (up to $\cong$) since the function BISIMILAR? returned true, and each pair of nodes from R is either a pair of equivalent nodes (in which case their outgoing edges are not included in the subgraph) or a pair of nodes with equal labels such that their pairs of children are in R (up to $\cong$). This substantiates the name of this transformation. And again we emphasize that the existence of two bisimilar subgraphs proves that the nodes are equivalent only if they meet the uniqueness condition.

Note that in our description of the algorithm we ignored the question of renamings. We did it for the sake of brevity, and actually (since we want to merge nodes even if they are equal only up to some renaming) we should take them into account which complicates the real implementation a little bit.

5 On order of transformation

Our experimental implementation of an equivalence prover for a first-order lazy language based on equality saturation is written in Scala and can be found on GitHub [1].

In our implementation we deviate from pure equality saturation for practical reasons. Pure equality saturation approach requires all transformations to be monotone with respect to the ordering $\sqsubseteq$ on E-graphs where $g_1 \sqsubseteq g_2$ means that the set of equalities encoded by g_1 is a subset of the corresponding set for g_2. Moreover, it requires them to be applied non-destructively, i.e. $g \sqsubseteq t(g)$ for each transformation t (in other words, we cannot remove nodes and edges, and split equivalence classes). But in return we are granted with a nice property: if we reach the fully saturated state (no transformation can change the E-graph further), then the resulting E-graph will be the same no matter in what order we have applied the transformations.

Unfortunately, in reality this is not very practical. First of all, saturation can never be reached if our axioms are complex enough (or at least it will take too long). In particular, the axioms we described above can be applied indefinitely in most cases. To solve this problem we should limit the axiom application. We can do this either by sacrificing randomness of the order of axiom application and simply limiting the total number of applications, or by using some limiting

monotone preconditions (similar to whistles in supercompilers), e.g. limiting the depth of the nodes to which axioms may be applied.

Second, if we always apply axioms non-destructively, we may find ourselves with an E-graph littered with useless garbage. But applying axioms destructively makes the randomness of axiom applications questionable, to say the least. Of course, the system may preserve the nice property of ordering independence, but it may be much harder to prove.

All in all, more or less deterministic order of transformation seems very desirable in practice. In our implementation we use the following order:

1. Transform the programs into an E-graph.
2. Apply all possible non-destructive transformations except merging by bisimilarity, congruence and injectivity to the equations that are already in E-graph but not to the equations that are added by the transformations performed in this step. This can be done by postponing the effects of transformations: first, we collect the effects of applicable transformations (nodes and edges to add, and classes to merge), then we apply all these effects at once.
3. Perform E-graph simplification: apply congruence, injectivity and destructive transformations to the E-graph until the saturation w.r.t. these transformations is reached. It is quite safe since all these transformations are normalizing in nature (i.e. they simplify the E-graph).
4. Perform merging by bisimilarity over each pair of equivalence classes. Pairs of equivalence classes are sorted according to resemblance of their components, and then the merging by bisimilarity algorithm is applied to them sequentially. After each successful merge perform E-graph simplification exactly as in the previous step.
5. Repeat steps 2–5 until the goal is reached.

This way E-graph is being built in a breadth-first manner, generation by generation, each generation of nodes and edges results from applying transformations to the nodes and edges of the previous generations. An exception from this general rule is a set of small auxiliary (but very important) transformations consisting of congruence, injectivity and all the destructive transformations which are applied until the saturation because they always simplify the E-graph.

6 Example

In this section we'll discuss a simple example to illustrate how the transformations described earlier work in practice. Consider the following program:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
odd n = case n of { Z → F; S m → even m }

evenSlow n = case n of { Z → T; S m → oddSlow m }
oddSlow n = not (evenSlow n)
```

It defines functions that check if a natural number is odd or even. The functions even and odd are defined efficiently using tail recursion, but the functions evenSlow and oddSlow will need linear amount of memory during the execution. Still, they are semantically equivalent, so we want to prove that even = evenSlow.

We will follow the scheme from the previous section. The program above correspond to the initial state of the E-graph and constitutes the zeroth generation. Now we should apply all applicable transformations to it. The only application that produces something new after simplification is of the transformation (subst-case-of) to the nodes oddSlow n = not (evenSlow n) and not b = **case** b **of** {...}. Indeed, it produces

```
oddSlow n = case (n { n = evenSlow n }) of { T → F; F → T }
```

which is immediately simplified by destructive application of (subst-id) to

```
oddSlow n = case (evenSlow n) of { T → F; F → T }
```

Actually this sequence of transformation is just expansion of the function not. This new definition of oddSlow appears in the E-graph alongside with the old definition of oddSlow. The current state of the E-graph is the first generation.

Now it is possible to apply the transformation (case-of-case-of) to the nodes oddSlow n = **case** (evenSlow n) **of** {...} and evenSlow n = **case** n **of** {...} which after simplification with (case-of-constr) gives the following definition:

```
oddSlow n = case n of { Z → F; S m → evenSlow2 m }
evenSlow2 m = case (oddSlow m) of { T → F; F → T }
```

Here evenSlow2 is an auxiliary function which is actually equal to evenSlow, but we don't know that yet. The current state of the E-graph is the second generation.

Now we apply (case-of-case-of) to these last two definitions which give us the following:

```
evenSlow2 n = case n of { Z → T; S m → oddSlow2 m }
oddSlow2 m = case (evenSlow2 m) of { T → F; F → T }
```

Again, we had to introduce a new function oddSlow2 which will turn out to be equal to oddSlow.

We should also apply (case-of-case-of) to the same definition of evenSlow2 and a different definition of oddSlow, namely oddSlow n = **case** (evenSlow n) **of** {...}, which gives us

```
evenSlow2 m = case (evenSlow m) of { T → T; F → F }
```

Although from this definition it is quite obvious that evenSlow2 = evenSlow, it is of no use to us: since our internal representation is untyped, evenSlow may return something different from T and F, and the fact that it can't should be proved by induction. Instead, other definitions will be used to show by induction that this equivalence holds.

First of all, let's see what the E-graph currently looks like:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
odd n = case n of { Z → F; S m → even m }

evenSlow n = case n of { Z → T; S m → oddSlow m }

oddSlow n = not (evenSlow n)
oddSlow n = case (evenSlow n) of { T → F; F → T }
oddSlow n = case n of { Z → F; S m → evenSlow2 m }

evenSlow2 n = case n of { Z → T; S m → oddSlow2 m }
evenSlow2 m = case (oddSlow m) of { T → F; F → T }
evenSlow2 m = case (evenSlow m) of { T → T; F → F }

oddSlow2 m = case (evenSlow2 m) of { T → F; F → T }
```

Now we can extract two equal definitions for function pairs evenSlow, oddSlow, and evenSlow2, oddSlow2, the corresponding nodes are highlighted. Here is the definitions for the first pair:

```
evenSlow n = case n of { Z → T; S m → oddSlow m }
oddSlow n = case (evenSlow n) of { T → F; F → T }
```

The definitions for the second pair of functions is the same up to function names and names of bound variables. As it can be seen, all the recursive calls here are preformed on structurally smaller arguments, so there may be no more than one fixed point of each subgraph, and since the subgraphs are bisimilar, we come to a conclusion that evenSlow = evenSlow2 and oddSlow = oddSlow2. Let's add this information to the E-graph, thus performing merging by bisimilarity:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
odd n = case n of { Z → F; S m → even m }

evenSlow n = case n of { Z → T; S m → oddSlow m }
evenSlow n = case (oddSlow n) of { T → F; F → T }
evenSlow n = case (evenSlow n) of { T → T; F → F }

oddSlow n = not (evenSlow n)
oddSlow n = case (evenSlow n) of { T → F; F → T }
oddSlow n = case n of { Z → F; S m → evenSlow m }
```

Now we can perform another merging by bisimilarity to equate the functions even and evenSlow. The needed bisimilar subgraphs conists of the nodes highlighted in the above program. The resulting E-graph is the third (and last, since we've reached the goal) generation and looks like this:

```
not b = case b of { T → F; F → T }

even n = case n of { Z → T; S m → odd m }
even n = case (odd n) of { T → F; F → T }
even n = case (even n) of { T → T; F → F }

odd n = case n of { Z → F; S m → even m }
odd n = not (even n)
odd n = case (even n) of { T → F; F → T }
```

It is interesting to point out that although we proved the goal statement using two mergings by bisimilarity (i.e. we used a lemma), we could have managed with only one if we had waited till the fourth generation without using induction. So sometimes lemmas aren't really required but may lead to earlier proof completion. Still, it doesn't mean that the proof will be found faster, even more, usually our tool takes slightly less time if we restrict application of merging by bisimilarity to the nodes from the goal statement since it doesn't have to check all equivalence class pairs from the E-graph in this case – but this is achieved at the cost of failures on tasks that really require lemmas.

7 Experimental evaluation

We've used a set of simple equations to evaluate our prover and compare it to similar tools. We've split this set into four groups: a main group of relatively simple equalities (Table 1) which don't seem to need any special features, a group of equalities that require nontrivial generalizations (Table 2), a group of equalities that need strong induction (Table 3), and a group of equalities that require coinduction (Table 4). The tests can be found in our repository [1] under the directory `samples`. For some of the tests we gave human-readable equations in the second column – note though that real equations are often a bit more complex because we have to make sure that they hold in a non-total untyped language.

The tables show average wall-clock time in seconds that the tools we've tested spent on the tests. We used a time limit of 5 minutes, runs exceeding the time limit counted as failures. We ran our benchmark on an Intel(R) Core(TM) i7 930 @ 2.80 GHz machine with Ubuntu 12.04. The tools we used in our benchmarking were:

- **graphsc**. Graphsc is our experimental prover based on the methods described in this paper. Note that although it internally works only with first-order functions, and there are many equalities in our sets involving higher-order functions, it can still prove them, because we perform defunctionalization before conversion to E-graph.
- **hosc**. HOSC is a supercompiler designed for program analysis, including the problem of function equivalence [12] (but it's not perfectly specialized for this task). It uses the following technique: first supercompile left hand side

and right hand side separately and then syntactically compare the residual programs [14, 17]. The column labeled **hosc (hl)** corresponds to the higher-level version of HOSC [13, 15]. It can come up with lemmas necessary to prove the main goal and prove them using a single-level HOSC.

- **zeno**. Zeno [20] is an inductive prover for Haskell. Internally it is quite similar to supercompilers. Zeno assumes totality, so it is not fair to compare tools that don't (our tool and HOSC) to pure Zeno, so we used a trick to encode programs operating on data with bottoms as total programs by adding additional bottom constructor to each data type. The results of Zeno on the adjusted samples are shown in the column **zeno (p)**. The results of pure Zeno (assuming totality) are shown for reference in the column **zeno (t)**.
- **hipspec**. HipSpec [5] is an inductive prover for Haskell which can generate conjectures by testing (using QuickSpec [6]), prove them using an SMT-solver, and then use them as lemmas to prove the goal and other conjectures. Like Zeno, HipSpec assumes totality, so we use the same transformation to model partiality. The results are shown in columns **hipspec (p)** and **hipspec (t)**. Note also that the results of HipSpec are sensitive to the `Arbitrary` type class instances for data types. We generated these instances automatically and ran HipSpec with `--quick-check-size=10` to maximize the number of tests passed given these instances. We also used the maximal induction depth of 2 (`-d2`) to make HipSpec pass two tests requiring strong induction.

Since the test set is not representative, it is useless to compare the tools by the number of test they pass. Moreover, the tools seem to fall into different niches. Still, some conclusions may be drawn from the results.

First of all, HipSpec is a very powerful tool, in total mode it proves most of the equalities from the main set (Table 1) and all of the equalities that require complex generalizations (Table 2). However, it is very slow on some tests. It is also much less powerful on tests adjusted with bottoms. Indeed, partial equalities are often a bit trickier to prove than their total counterparts. It is also possible that this particular approach of reducing a partial problem to a total one and then using a total prover is not very efficient.

Zeno and HOSC are very fast which seems to be due to their depth-first nature. Zeno is also quite powerful and can successfully compete with the slower HipSpec, especially in the partial case. HOSC fails many test from the main set presumably due to the fact that it is not specialized enough for the task of proving equivalences. For example, the equivalence `idle-simple` is much easier to prove when transforming both sides simultaneously. Also HOSC can't prove `bool-eq` and `sadd-comm` because they need the transformation (case-of-transpose) which supercompilers usually lack. Interestingly, higher-level HOSC does prove some additional equalities, but not in the case of tests that really need lemmas (except `even-double-acc` from Table 2), which are the last four tests in the main set (they need lemmas in a sense that neither Graphsc, nor HipSpec can prove them without lemmas).

Name	Description	gra-phsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hip-spec (p)	hip-spec (t)
add-assoc	x + (y + z) = (x + y) + z	1.6	0.5	0.6	0.3	0.3	8.8	0.9
append-assoc	x ++ (y ++ z) = (x ++ y) ++ z	1.8	0.5	0.6	0.3	1.1	4.9	3.2
double-add	double (x+y) = double x + double y	2.2	0.6	0.6	0.3	0.3	21.5	2.0
even-double	even (double x) = true	1.7	0.6	0.6	0.3	0.3	82.5	97.9
ho/concat-concat	concat (concat xs) = concat (map concat xs)	3.2	0.6	0.7	0.4	0.4	80.0	47.0
ho/filter-append	filter p (xs ++ ys) = filter p xs ++ filter p ys	3.0	0.6	0.8	0.3	0.3	9.6	4.8
ho/map-append	map f (xs ++ ys) = map f xs ++ map f ys	2.1	0.6	0.7	0.3	0.3	9.8	4.5
ho/map-comp	map (f . g) xs = (map f . map g) xs	3.4	0.6	0.6	0.3	0.3	4.7	4.7
ho/map-concat	map f (concat x) = concat (map (map f) x)	2.8	0.6	0.7	0.3	0.3	91.4	47.9
ho/map-filter	filter p (map f xs) = map f (filter (p . f) xs)	3.6	0.7	0.7	0.3	0.3	6.3	5.8
idnat-idemp	idNat (idNat x) = idNat x	1.5	0.5	0.5	0.3	0.3	0.8	0.7
take-drop	drop n (take n x) = []	2.4	0.6	0.7	0.3	0.3	47.8	9.6
take-length	take (length x) x = x	2.3	0.6	0.6	0.3	0.3	6.8	7.9
length-concat	length (concat x) = sum (map length x)	2.8	0.7	0.8	0.3	0.3	fail	8.5
append-take-drop	take n x ++ drop n x = x	3.6	fail	1.1	0.5	0.3	113.0	11.9
deepseq-idemp	deepseq x (deepseq x y) = deepseq x y	1.8	fail	0.9	0.3	0.3	4.7	1.6
deepseq-s	deepseq x (S y) = deepseq x (S (deepseq x y))	2.1	fail	0.7	0.3	0.3	10.1	0.7
mul-assoc	(x * y) * z = x * (y * z)	11.6	0.8	fail	0.3	0.4	176.2	30.4
mul-distrib	(x*y) + (z*y) = (x + z)*y	3.9	0.7	fail	0.3	0.3	151.8	92.1
mul-double	x * double y = double (x*y)	5.1	0.6	fail	0.3	0.3	165.6	142.1
ho/fold-append	foldr f (foldr f a ys) xs = foldr f a (xs ++ ys)	2.1	0.6	0.7	fail	0.3	176.8	4.6
ho/church-id	unchurch (church x) = x	6.1	0.6	0.6	0.3	0.3	fail	fail
ho/church-pred		fail	0.7	0.8	fail	fail	fail	fail
ho/church-add		fail	0.7	0.7	0.3	0.3	fail	fail
idle-simple	idle x = idle (idle x)	1.4	fail	fail	0.3	0.3	0.8	0.7
bool-eq		1.3	fail	fail	0.3	0.3	1.1	0.8
sadd-comm		2.1	fail	fail	0.3	0.3	3.3	16.7
ho/filter-idemp	filter p (filter p xs) = filter p xs	fail	fail	fail	0.3	0.3	1.3	0.9
even-slow-fast	even x = evenSlow x	1.8	fail	fail	fail	fail	2.6	1.1
or-even-odd	even x || odd x = true	3.9	fail	fail	fail	0.3	128.9	1.0
dummy		1.6	fail	fail	0.3	0.3	2.4	0.8
idle	idle x = deepseq x 0	1.5	fail	fail	0.3	0.3	1.8	0.7
quad-idle		1.9	fail	fail	0.3	0.3	fail	0.7
exp-idle		3.4	fail	fail	0.3	fail	fail	1.7

Table 1: Comparison of different tools on the main test subset

Our tool, Graphsc, seems to be in the middle: it's slower than HOSC and Zeno (and it should be since it's breadth-first in nature) but rarely needs more than 10 seconds. It's interesting to analyze the failures of our tool. It fails three tests from the main set. The tests `ho/church-pred` and `ho/church-add` need deeper driving, our tool can pass them in the experimental supercompilation mode which change the order of transformation to resemble that of traditional supercompilers. Unfortunately, this mode is quite slow and leads to many other failures when enabled by default. The test `ho/filter-idemp` is interesting: it needs more information to be propagated, namely that the expression `p x` evaluates to `True`. Since this expression is not a variable, we don't propagate this information (and neither does HOSC, however there is an experimental mode for HOSC that does this and helps pass this test).

Name	Description	graphsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hipspec (p)	hipspec (t)
even-dbl-acc-lemma	even (doubleAcc x (S y)) = odd (doubleAcc x y)	fail	0.7	0.6	0.3	0.3	38.8	37.4
nrev-idemp-nat		fail	fail	fail	0.3	0.3	21.9	2.0
deepseq-add-comm		fail	fail	fail	fail	0.3	fail	2.1
even-double-acc	even (doubleAcc x 0) = true	fail	fail	0.8	fail	fail	fail	38.4
nrev-list	naiveReverse = reverse	fail	fail	fail	fail	fail	185.5	19.7
nrev-nat		fail	fail	fail	fail	fail	fail	1.1

Table 2: Comparison of the tools on the tests that require nontrivial generalization

Now let's look at the tests requiring nontrivial generalizations (Table 2). Here we call a generalization trivial if it's just peeling of the outer function call, e.g. `f (g a) (h b c)` trivially generalizes to `f x y` with `x = g a` and `y = h b c`. Our tool supports only trivial generalizations, and they are enough for a large number of examples. But in some cases more complex generalizations are needed, e.g. to prove the equality `even-dbl-acc-lemma` one need to generalize the expression `odd (doubleAcc x (S (S y)))` to `odd (doubleAcc x z)` with `z = S (S y)`. It's not super sophisticated, but the expression left after taking out the `S (S y)` is a composition of two functions, which makes this generalization nontrivial. Our tool is useless on these examples. Supercompilers like HOSC usually use most specific generalizations which helps in some cases. But the best tool to prove equalities like these is HipSpec (and still it doesn't work that well in the partial case).

In Table 3 the tests are shown that require strong induction, i.e. induction schemes that peel more than one constructor at a time. This is not a problem for Graphsc and HOSC since they don't explicitly instantiate induction schemes. But Zeno and HipSpec do. In the case of HipSpec the maximum induction depth can be increased, so we specified the depth of 2, which helped HipSpec to pass two of these tests at the price of increased running times for other tests.

Name	Description	graphsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hip-spec (p)	hip-spec (t)
add-assoc-bot		2.1	0.6	0.6	fail	fail	fail	fail
double-half	double (half x) + mod2 x = x	4.6	fail	1.2	fail	fail	81.7	6.6
length-intersperse	length (intersperse x xs) = length (intersperse y xs)	fail	0.6	0.7	fail	fail	1.6	0.9
kmp-eq		fail	1.2	1.7	fail	fail	fail	fail

Table 3: Comparison of the tools on the tests that require strong induction

Our tool doesn't pass the KMP-test because it requires deep driving (and again, our experimental supercompilation mode helps pass it). In the case of `length-intersperse` it has trouble with recognizing the goal as something worth proving because both sides are equal up to renaming. Currently it is not obvious how this (seemingly technical) problem can be solved.

Name	Description	graphsc	hosc	hosc (hl)	zeno (p)	zeno (t)	hip-spec (p)	hip-spec (t)
inf	fix S = fix S	1.2	0.4	0.5	fail	fail	fail	fail
shuffled-let		1.5	0.5	0.5	fail	fail	fail	fail
shifted-cycle	cycle [A,B] = A : cycle [B,A]	3.6	fail	fail	fail	fail	fail	fail
ho/map-iterate	map f (iterate f a) = iterate f (f a)	fail	0.6	0.6	fail	fail	fail	fail

Table 4: Comparison of the tools on the tests that require coinduction

The last test subset to discuss is the subset of tests requiring coinduction (Table 4). Coinduction is not currently supported by Zeno and HipSpec, although there are no obstacles to implement it in the future. The equality `ho/map-iterate` can't be proved by our tool because besides coinduction it needs a nontrivial generalization.

8 Related work

Our work is based on the method of equality saturation, originally proposed by Tate et al [23], which in turn is inspired by E-graph-based theorem provers like Simplify [7]. Their implementation, named Peggy, was designed to transform programs in low-level imperative languages (Java bytecode and LLVM), although internally Peggy uses a functional representation. In our work we transform lazy functional programs, so we don't have to deal with encoding imperative operations in functional representation, which makes everything much easier. Another difference is that in our representation nodes correspond to functions, not just first-order values, which allows more general recursion to be used, moreover we merge equivalence classes corresponding to functions equal up to parameter permutation, which considerably reduces the E-graph complexity. We also articulate the merging by bisimilarity transformation which plays a

very important role, making our tool essentially an inductive prover. Note that Peggy has a similar (but simpler) transformation that can merge θ-nodes, but it doesn't seem to be published anywhere.

Initially our work arose from analyzing differences between overgraph supercompilation [9] and equality saturation, overgraph supercompilation being a variety of multi-result supercompilation with a flavor of equality saturation. The present paper is loosely based on the preprint [10] which used a different terminology (hypergraph instead of E-graph, hence the name of our GitHub repository). We also used to consider the method to be a kind of supercompilation, but although it borrows a lot from supercompilation, it is much closer to equality saturation.

Supercompilation [24] is a program transformation technique that consists in building a process tree (perhaps implicitly) by applying driving and generalization to its leaves, and then folding the tree, essentially producing a finite program, equivalent to the original one. Although supercompilation is usually considered a source-to-source program transformation, it can be used to prove program equivalence by syntactically comparing the resulting programs, due to the normalizing effect of supercompilation.

Traditional supercompilers always return a single program, but for some tasks, like program analysis, it is beneficial to produce a set of programs for further processing. This leads to the idea of multi-result supercompilation, which was put forward by Klyuchnikov and Romanenko [16]. Since there are many points of decision making during the process of supercompilation (mainly when and how to generalize), a single-result supercompiler may be transformed into a multi-result one quite easily by taking multiple paths in each such point. The mentioned motivation behind multi-result supercompilation is essentially the same as that behind equality saturation.

Another important enhancement of traditional supercompilation is higher-level supercompilation. Higher-level supercompilation is a broad term denoting systems that use supercompilation as a primitive operation, in particular supercompilers that can invent lemmas, prove them with another (lower-level) supercompiler, and use them in the process of supercompilation. Examples of higher-level supercompilation are distillation, proposed by Hamilton [11], and two-level supercompilation, proposed by Klyuchnikov and Romanenko [13,15].

Zeno [20] is an inductive prover for Haskell which works quite similarly to multi-result supercompilation. Indeed, Zeno performs case analysis and applies induction (both correspond to driving in supercompilation) until it heuristically decides to generalize or apply a lemma (in supercompilation this heuristic is called a whistle). That is, both methods are depth-first in nature unlike the equality saturation approach, which explores possible program transformations in breadth-first manner.

HipSpec [5] is another inductive prover for Haskell. It uses theory exploration to discover lemmas. For this purpose it invokes QuickSpec [6], which generates all terms up to some depth, splits them into equivalence classes by random testing, and then transforms these classes into a set of conjectures. After that

these conjectures are proved one by one and then used as lemmas to prove other conjectures and the main goal. To prove conjectures HipSpec uses external SMT-solvers. This bottom-up approach is contrasted to the top-down approach of most inductive provers, including Zeno and supercompilers, which invent lemmas when the main proof gets stuck. HipSpec discovers lemmas speculatively which is good for finding useful generalizations but may take much more time.

As to our tool, we do something similar to the bottom-up approach, but instead of using arbitrary terms, we use the terms represented by equivalence classes of the E-graph (i.e. generated by transforming initial term) and then try to prove them equal pairwise, discarding unfruitful pairs by comparing perfect tree prefixes that have been built in the E-graph so far, instead of testing. Since we use only terms from the E-graph, we can't discover complex generalizations this way, although we can still find useful auxiliary lemmas sometimes (but usually for quite artificial examples).

Both Zeno and HipSpec instantiate induction schemes while performing proof by induction. We use a different technique, consisting in checking the correctness of a proof graph, similarly to productivity and termination checking in languages like Agda. This approach has some advantages, for example we don't have to know the induction depth in advance. Supercompilers usually don't even check the correctness because for single-level supercompilation it is ensured automatically. It is not the case for higher-level supercompilation, and for example, HOSC checks that every lemma used is an improvement lemma in the terminology of Sand's theory [19].

9 Conclusion and future work

In this paper we have shown how an inductive prover for a non-total first-order lazy functional language can be constructed on top of the ideas of equality saturation. The key ingredient is merging by bisimilarity which enables proof by induction. Another feature that we consider extremely important is the ability to merge equivalence classes even if they represent functions equal only up to some renaming. This idea can be extended, for example if we had ticks, we could merge classes representing functions which differ by a finite number of ticks, but we haven't investigated into it yet.

Of course our prover has some deficiencies:

– Our prover lacks proper generalizations. This is a huge issue since many real-world examples require them. We have an experimental flag that enables arbitrary generalizations, but it usually leads to combinatorial explosion of the E-graph. There are two plausible ways to fix this issue. The first one is to use some heuristics to find generalizations from failed proof attempts, like it's done in supercompilers and many inductive provers. The other one is to rely on some external generalization and lemma discovery tools. In this case a mechanism of applying externally specified lemmas and generalizations may be very useful. In the case of E-graphs it is usually done with E-matching,

and we have an experimental implementation, although it doesn't work very well yet.
- Although it is possible to prove some propositions that hold only in total setting by adding some transformations, our prover is not very effective on this task. It may not seem to be a big problem if we only work with non-total languages like Haskell, but actually even in this case the ability to work with total values is important since such values may appear even in partial setting, e.g. when using the function `deepseq`.
- Our internal representation is untyped, and for this reason we cannot prove some natural equalities.
- We don't support higher-order functions internally and need to perform defunctionalization if the input program contains them. This issue is especially important if we want to produce a residual program.
- Our prover is limited to propositions about function equivalence, and it is not obvious how to add support for implications.

Besides mitigating the above problems, another possibility for future work is exploring other applications. Equality saturation is a program transformation technique which is not limited to proving function equivalence. Initially it was successfully applied to imperative program optimization, so some results in the functional field are to be expected. Even merging by bisimilarity may be of some use since it is known that using lemmas may lead to superlinear performance improvement. Another possible area is program analysis.

Acknowledgements

The author would like to express his gratitude to Sergei Romanenko, Andrei Klimov, Ilya Klyuchnikov, and other participants of the Refal seminar at Keldysh Institute.

References

1. Graphsc source code and the test suite. `https://github.com/sergei-grechanik/supercompilation-hypergraph`.
2. M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991. Summary in *ACM Symposium on Principles of Programming Languages (POPL), San Francisco, California*, 1990.
3. A. Abel. Foetus – termination checker for simple functional programs, July 16 1998.
4. A. Abel and T. Altenkrich. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002.
5. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In M. P. Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2013.

6. K. Claessen, N. Smallbone, and J. Hughes. Quickspec: Guessing formal specifications using testing. In G. Fraser and A. Gargantini, editors, *Tests and Proofs, 4th International Conference, TAP 2010, Málaga, Spain, July 1-2, 2010. Proceedings*, volume 6143 of *Lecture Notes in Computer Science*, pages 6–21. Springer, 2010.
7. Detlefs, Nelson, and Saxe. Simplify: A theorem prover for program checking. *JACM: Journal of the ACM*, 52, 2005.
8. A. Dovier and C. Piazza. The subgraph bisimulation problem. *IEEE Transactions on Knowledge & Data Engineering*, 15(4):1055–6, 2003. Publisher: IEEE, USA.
9. S. A. Grechanik. Overgraph representation for multi-result supercompilation. In A. Klimov and S. Romanenko, editors, *Proceedings of the Third International Valentin Turchin Workshop on Metacomputation*, pages 48–65, Pereslavl-Zalessky, Russia, July 2012. Pereslavl-Zalessky: Ailamazyan University of Pereslavl.
10. S. A. Grechanik. Supercompilation by hypergraph transformation. Preprint 26, Keldysh Institute of Applied Mathematics, 2013.
URL: `http://library.keldysh.ru/preprint.asp?id=2013-26&lg=e`.
11. 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.
12. I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.
13. I. Klyuchnikov. Towards effective two-level supercompilation. Preprint 81, Keldysh Institute of Applied Mathematics, 2010. URL: `http://library.keldysh.ru/preprint.asp?id=2010-81&lg=e`.
14. 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.
15. I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.
16. I. G. Klyuchnikov and S. A. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In E. Clarke, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011*, volume 7162 of *Lecture Notes in Computer Science*, pages 210–226. Springer, 2012.
17. A. Lisitsa and M. Webster. Supercompilation for equivalence testing in metamorphic computer viruses detection. In *Proceedings of the First International Workshop on Metacomputation in Russia*, 2008.
18. Nelson and Oppen. Fast decision procedures based on congruence closure. *JACM: Journal of the ACM*, 27, 1980.
19. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.
20. W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In *TACAS*, Lecture Notes in Computer Science, March 2012.
21. M. Sørensen, R. Glück, and N. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1993.
22. M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 737–742. Springer, 2011.

23. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.
24. V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

Staged Multi-Result Supercompilation: Filtering by Transformation*

Sergei A. Grechanik, Ilya G. Klyuchnikov, Sergei A. Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Abstract. When applying supercompilation to problem-solving, multi-result supercompilation enables us to find the best solutions by generating a set of possible residual graphs of configurations that are then filtered according to some criteria. Unfortunately, the search space may be rather large. However, we show that the search can be drastically reduced by decomposing multi-result supercompilation into two stages. The first stage produces a compact representation for the set of residual graphs by delaying some graph-building operation. These operations are performed at the second stage, when the representation is interpreted, to actually produce the set of graphs. The main idea of our approach is that, instead of filtering a collection of graphs, we can analyze and clean its compact representation. In some cases of practical importance (such as selecting graphs of minimal size and removing graphs containing unsafe configurations) cleaning can be performed in linear time.

1 Introduction

When applying supercompilation [9,19,21,33,35,37–39,42–46] to problem-solving [10,12,14,15,22,28–32], multi-result supercompilation enables us to find the best solutions by generating a set of possible residual graphs of configurations that are then filtered according to some criteria [13, 23–25].

Unfortunately, the search space may be rather large [16, 17]. However, we show that the search can be drastically reduced by decomposing multi-result supercompilation into two stages [40, 41]. The first stage produces a compact representation for the set of residual graphs by delaying some graph-building operation. These operations are performed at the second stage, when the representation is interpreted, to actually produce the set of graphs. The main idea of our approach is that, instead of filtering a collection of graphs, we can analyze and clean its compact representation. In some cases of practical importance (such as selecting graphs of minimal size and removing graphs containing unsafe configurations) cleaning can be performed in linear time.

* Supported by Russian Foundation for Basic Research grant No. 12-01-00972-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

2 Filtering before Producing... How?

2.1 Multi-Result Supercompilation and Filtering

A popular approach to problem solving is *trial and error*: (1) *generate* alternatives, (2) *evaluate* alternatives, (3) *select* the best alternatives.

Thus, when trying to apply supercompilation to problem solving we naturally come to the idea of *multi-result* supercompilation: instead of trying to guess, which residual graph of configurations is "the best" one, a multi-result supercompiler produces a collection of residual graphs.

Suppose we have a multi-result supercompiler `mrsc` and a filter `filter`. Combining them, we get a problem-solver

```
solver = filter ∘ mrsc
```

where `mrsc` is a general-purpose tool (at least to some extent), while `filter` incorporates some knowledge about the problem domain. A good feature of this design is its modularity and the clear separation of concerns: in the ideal case, `mrsc` knows nothing about the problem domain, while `filter` knows nothing about supercompilation.

2.2 Fusion of Supercompilation and Filtering

However, the main problem with multi-result supercompilation is that it can produce millions of residual graphs! Hence, it seems to be a good idea to suppress the generation of the majority of residual graphs "on the fly", in the process of supercompilation. This can be achieved if the criteria `filter` is based upon are "monotonic": if some parts of a partially constructed residual graph are "bad", then the completed residual graph is also certain to be a "bad" one[1].

We can exploit monotonicity by fusing `filter` and `mrsc` into a monolithic program

```
solver′ = fuse filter mrsc
```

where `fuse` is an automatic tool (based, for example, on supercompilation), or just a postgraduate who has been taught (by his scientific adviser) to perform fusion by hand. :-)

An evident drawback of this approach is its non-modularity. Every time `filter` is modified, the fusion of `mrsc` and `filter` has to be repeated.

2.3 Staged Supercompilation: Multiple Results Seen as a Residual Program

Here we put forward an alternative approach that:

[1] Note the subtle difference between "monotonic" and "finitary" [39]. "Monotonic" means that a *bad* situation cannot *become* good, while "finitary" means that a *good* situation cannot forever *remain* good.

1. Completely separates supercompilation from filtering.
2. Enables filtering of partially constructed residual graphs.

Thus the technique is modular, and yet reduces the search space and consumed computational resources.

Our "recipe" is as follows. (1) Replace "small-step" supercompilation with "big-step" supercompilation. (2) Decompose supercompilation into two stages. (3) Consider the result of the first stage as a "program" to be interpreted by the second stage. (4) Transform the "program" to reduce the number of graphs to be produced.

Small-step ⇒ big-step Supercompilation can be formulated either in "small-step" or in "big-step" style. Small-step supercompilation proceeds by rewriting a graph of configurations. Big-step supercompilation is specified/implemented in compositional style: the construction of a graph amounts to constructing its subgraphs, followed by synthesizing the whole graph from its previously constructed parts. Multi-result supercompilation was formulated in small-step style [23, 24]. First of all, given a small-step multi-result supercompiler `mrsc`, we can refactor it, to produce a *big-step* supercompiler `naive-mrsc` (see details in Section 3).

Identifying Cartesian products Now, examining the text of `naive-mrsc` (presented in Section 3), we can see that, at some places, `naive-mrsc` calculates "Cartesian products": if a graph g is to be constructed from k subgraphs $g_1, \ldots, g_k$, `naive-mrsc` computes k sets of graphs $gs_1, \ldots, gs_k$ and then considers all possible $g_i \in gs_i$ for $i = 1, \ldots, k$ and constructs corresponding versions of the graph g.

Staging: delaying Cartesian products At this point the process of supercompilation can be decomposed into two stages

$$\texttt{naive-mrsc} \stackrel{\circ}{=} \langle\!\langle \_ \rangle\!\rangle \circ \texttt{lazy-mrsc}$$

where $\langle\!\langle \_ \rangle\!\rangle$ is a unary function, and $f \stackrel{\circ}{=} g$ means that $f\ x = g\ x$ for all x.

At the first stage, `lazy-mrsc` generates a "lazy graph", which, essentially, is a "program" to be "executed" by $\langle\!\langle \_ \rangle\!\rangle$. Unlike `naive-mrsc`, `lazy-mrsc` does not calculate Cartesian products immediately: instead, it outputs requests for $\langle\!\langle \_ \rangle\!\rangle$ to calculate them at the second stage.

Fusing filtering with the generation of graphs Suppose, `l` is a lazy graph produced by `lazy-mrsc`. By evaluating $\langle\!\langle$ `l` $\rangle\!\rangle$, we can generate the same bag of graphs, as would have been produced by the original `naive-mrsc`.

However, usually, we are not interested in the whole bag $\langle\!\langle$ `l` $\rangle\!\rangle$. The goal is to find "the best" or "most interesting" graphs. Hence, there should be developed some techniques of extracting useful information from a lazy graph l without evaluating $\langle\!\langle$ `l` $\rangle\!\rangle$ directly.

This can be formulated in the following form. Suppose that a function `filter` filters bags of graphs, removing "bad" graphs, so that

$$\texttt{filter}\ \langle\!\langle\ \texttt{l}\ \rangle\!\rangle$$

generates the bag of "good" graphs. Let `clean` be a transformer of lazy graphs such that

$$\texttt{filter} \circ \langle\!\langle \_ \rangle\!\rangle \stackrel{\circ}{=} \langle\!\langle \_ \rangle\!\rangle \circ \texttt{clean}$$

which means that $\texttt{filter}\ \langle\!\langle\ \texttt{l}\ \rangle\!\rangle$ and $\langle\!\langle\ \texttt{clean l}\ \rangle\!\rangle$ always return the same collection of graphs.

In general, a lazy graph transformer `clean` is said to be a *cleaner* if for any lazy graph `l`

$$\langle\!\langle\ \texttt{clean l}\ \rangle\!\rangle \subseteq \langle\!\langle\ \texttt{l}\ \rangle\!\rangle$$

The nice property of cleaners is that they are *composable*: given $\texttt{clean}_1$ and $\texttt{clean}_2$, $\texttt{clean}_2 \circ \texttt{clean}_1$ is also a cleaner.

2.4 Typical Cleaners

Typical tasks are finding graphs of minimal size and removing graphs that contain "bad" configurations. It is easy to implement corresponding cleaners in such a way that the lazy graph is traversed only once, in a linear time.

2.5 What are the Advantages?

We get the following scheme:

$$\texttt{filter} \circ \texttt{naive-mrsc} \stackrel{\circ}{=}$$
$$\texttt{filter} \circ \langle\!\langle \_ \rangle\!\rangle \circ \texttt{lazy-mrsc} \stackrel{\circ}{=} \langle\!\langle \_ \rangle\!\rangle \circ \texttt{clean} \circ \texttt{lazy-mrsc}$$

We can see that:

- The construction is modular: `lazy-mrsc` and $\langle\!\langle \_ \rangle\!\rangle$ do not have to know anything about filtering, while `clean` does not have to know anything about `lazy-mrsc` and $\langle\!\langle \_ \rangle\!\rangle$.
- Cleaners are composable: we can decompose a sophisticated cleaner into a composition of simpler cleaners.
- In many cases (of practical importance) cleaners can be implemented in such a way that the best graphs can be extracted from a lazy graph in linear time.

2.6 Codata and Corecursion: Decomposing `lazy-mrsc`

By using codata and corecursion, we can decompose `lazy-mrsc` into two stages

```
lazy-mrsc ≙ prune-cograph ∘ build-cograph
```

where `build-cograph` constructs a (potentially) infinite tree, while `prune-cograph` traverses this tree and turns it into a lazy graph (which is finite).

The point is that `build-cograph` performs driving and rebuilding configurations, while `prune-cograph` uses whistle to turn an infinite tree to a finite graph. Thus `build-cograph` knows nothing about the whistle, while `prune-cograph` knows nothing about driving and rebuilding. This further improves the modularity of multi-result supercompilation.

2.7 Cleaning before Whistling

Now it turnes out that some cleaners can be pushed over `prune-cograph`!

Suppose `clean` is a lazy graph cleaner and `clean∞` a cograph cleaner, such that

```
clean ∘ prune-cograph ≙ prune-cograph ∘ clean∞
```

then

```
clean ∘ lazy-mrsc ≙
    clean ∘ prune-cograph ∘ build-cograph ≙
    prune-cograph ∘ clean∞ ∘ build-cograph
```

The good thing is that `build-cograph` and `clean∞` work in a lazy way, generating subtrees by demand. Hence, evaluating

```
⟪ prune-cograph ∘ (clean∞ (build-cograph c))  ⟫
```

is likely to be less time and space consuming than directly evaluating

```
⟪ clean (lazy-mrsc c) ⟫
```

3 A Model of Big-Step Multi-Result Supercompilation

We have formulated and implemented in Agda [2] an idealized model of big-step multi-result supercompilation [1]. This model is rather abstract, and yet it can be instantiated to produce runnable supercompilers. By the way of example, the abstract model has been instantiated to produce a multi-result supercompiler for counter systems [16].

3.1 Graphs of Configurations

Given an initial configuration c, a supercompiler produces a list of "residual" graphs of configurations: $g_1, \ldots, g_k$.

Graphs of configurations are supposed to represent "residual programs" and are defined in Agda (see `Graphs.agda`) [2] in the following way:

```
data Graph (C : Set) : Set where
  back  : ∀ (c : C) → Graph C
  forth : ∀ (c : C) (gs : List (Graph C)) → Graph C
```

Technically, a `Graph C` is a tree, with `back` nodes being references to parent nodes.

A graph's nodes contain configurations. Here we abstract away from the concrete structure of configurations. In this model the arrows in the graph carry no information, because, this information can be kept in nodes. (Hence, this information is supposed to be encoded inside "configurations".)

To simplify the machinery, back-nodes in this model of supercompilation do not contain explicit references to parent nodes. Hence, `back c` means that `c` is foldable to a parent configuration (perhaps, to several ones).

- Back-nodes are produced by folding a configuration to another configuration in the history.
- Forth-nodes are produced by
  - decomposing a configuration into a number of other configurations (e.g. by driving or taking apart a let-expression), or
  - by rewriting a configuration by another one (e.g. by generalization, introducing a let-expression or applying a lemma during two-level supercompilation).

3.2 "Worlds" of Supercompilation

The knowledge about the input language a supercompiler deals with is represented by a "world of supercompilation", which is a record that specifies the following[2].

- `Conf` is the type of "configurations". Note that configurations are not required to be just expressions with free variables! In general, they may represent sets of states in any form/language and as well may contain any *additional* information.
- `_⊑_` is a "foldability relation". `c ⊑ c′` means that `c` is "foldable" to `c′`. (In such cases `c′` is usually said to be "more general" than `c`.)
- `_⊑?_` is a decision procedure for `_⊑_`. This procedure is necessary for implementing algorithms of supercompilation.

[2] Note that in Agda a function name containing one or more underscores can be used as a "mixfix" operator. Thus `a + b` is equivalent to `_+_ a b`, `if a then b else c` to `if_then_else_ a b c` and `[ c ]` to `[_] c`.

– `_⇉` is a function that gives a number of possible decompositions of a configuration. Let `c` be a configuration and `cs` a list of configurations such that `cs ∈ c ⇉`. Then `c` can be "reduced to" (or "decomposed into") configurations `cs`.
  Suppose that "driving" is deterministic and, given a configuration `c`, produces a list of configurations `c ⇊`. Suppose that "rebuilding" (generalization, application of lemmas) is non-deterministic and `c ↷` is the list of configurations that can be produced by rebuilding. Then (in this special case) `_⇉` can be implemented as follows:

```
c ⇉ = [ c ⇊ ] ++ map [_] (c ↷)
```

– `whistle` is a "bar whistle" [6] that is used to ensure termination of functional supercompilation (see details in Section 3.6) .

Thus we have the following definition in Agda[3]:

```
record ScWorld : Set₁ where

  field
    Conf : Set
    _⊑_ : (c c′ : Conf) → Set
    _⊑?_ : (c c′ : Conf) → Dec (c ⊑ c′)
    _⇉ : (c : Conf) → List (List Conf)
    whistle : BarWhistle Conf

  open BarWhistle whistle public

  History : Set
  History = List Conf

  Foldable : ∀ (h : History) (c : Conf) → Set
  Foldable h c = Any (_⊑_ c) h

  foldable? : ∀ (h : History) (c : Conf) → Dec (Foldable h c)
  foldable? h c = Any.any (_⊑?_ c) h
```

Note that, in addition to (abstract) fields, there are a few concrete type and function definitions[4].

– `History` is a list of configuration that have been produced in order to reach the current configuration.
– `Foldable h c` means that `c` is foldable to a configuration in the history `h`.
– `foldable? h c` decides whether `Foldable h c`.

[3] Record declarations in Agda are analogous to "abstract classes" in functional languages and "structures" in Standard ML, abstract members being declared as "fields".

[4] The construct `open BarWhistle whistle public` brings the members of `whistle` into scope, so that they become accessible directly.

3.3 Graphs with labeled edges

If we need labeled edges in the graph of configurations, the labels can be hidden inside configurations. (Recall that "configurations" do not have to be just symbolic expressions, as they can contain any additional information.)

Here is the definition in Agda of worlds of supercompilation with labeled edges:

```
record ScWorldWithLabels : Set₁ where
  field
    Conf : Set    -- configurations
    Label : Set   -- edge labels
    _⊑_ : (c c′ : Conf) → Set            -- c is foldable to c′
    _⊑?_ : (c c′ : Conf) → Dec (c ⊑ c′)  -- ⊑ is decidable
    -- Driving/splitting/rebuilding a configuration:
    _⇉ : (c : Conf) → List (List (Label × Conf))
    -- a bar whistle
    whistle : BarWhistle Conf
```

There is defined (in `BigStepSc.agda`) a function

```
injectLabelsInScWorld : ScWorldWithLabels → ScWorld
```

that injects a world with labeled edges into a world without labels (by hiding labels inside configurations).

3.4 A Relational Specification of Big-Step Non-Deterministic Supercompilation

In `BigStepSc.agda` there is given a relational definition of non-deterministic supercompilation [24] in terms of two relations

```
infix 4 _⊢NDSC_↪_ _⊢NDSC*_↪_

data _⊢NDSC_↪_ : ∀ (h : History) (c : Conf)
    (g : Graph Conf) → Set
_⊢NDSC*_↪_ : ∀ (h : History) (cs : List Conf)
    (gs : List (Graph Conf)) → Set
```

which are defined with respect to a world of supercompilation.

Let `h` be a history, `c` a configuration and `g` a graph. Then `h ⊢NDSC c ↪ g` means that `g` can be produced from `h` and `c` by non-deterministic supercompilation.

Let `h` be a history, `cs` a list of configurations, `gs` a list of graphs, and `length cs = length gs`. Then `h ⊢NDSC* cs ↪ gs` means that each `g ∈ gs` can be produced from the history `h` and the corresponding `c ∈ cs` by non-deterministic supercompilation. Or, in Agda:

```
h ⊢NDSC* cs ↪ gs = Pointwise.Rel (_⊢NDSC_↪_ h) cs gs
```

`⊢NDSC_↪_` is defined by two rules

```
data _⊢NDSC_↪_ where
  ndsc-fold  : ∀ {h : History} {c}
    (f : Foldable h c) →
    h ⊢NDSC c ↪ back c
  ndsc-build : ∀ {h : History} {c}
    {cs : List (Conf)} {gs : List (Graph Conf)}
    (¬f : ¬ Foldable h c)
    (i : cs ∈ c ⇉)
    (s : (c :: h) ⊢NDSC* cs ↪ gs) →
    h ⊢NDSC c ↪ forth c gs
```

The rule `ndsc-fold` says that if `c` is foldable to a configuration in `h` there can be produced the graph `back c` (consisting of a single back-node).

The rule `ndsc-build` says that there can be produced a node `forth c gs` if the following conditions are satisfied.

- `c` *is not* foldable to a configuration in the history `h`.
- `c ⇉` contains a list of configurations `cs`, such that `(c :: h) ⊢NDSC* cs ↪ gs`.

Speaking more operationally, the supercompiler first decides how to decompose `c` into a list of configurations `cs` by selecting a `cs ∈ c ⇉`. Then, for each configuration in `cs` the supercompiler produces a graph, to obtain a list of graphs `gs`, and builds the graph `c ↪ forth c gs`.

3.5 A Relational Specification of Big-Step Multi-Result Supercompilation

The main difference between multi-result and non-deterministic supercompilation is that multi-result uses a *whistle* (see `Whistles.agda`) in order to ensure the finiteness of the collection of residual graphs [24].

In `BigStepSc.agda` there is given a relational definition of multi-result supercompilation in terms of two relations

```
infix 4 _⊢MRSC_↪_ _⊢MRSC*_↪_

data _⊢MRSC_↪_ : ∀ (h : History) (c : Conf)
    (g : Graph Conf) → Set
_⊢MRSC*_↪_ : ∀ (h : History) (cs : List Conf)
    (gs : List (Graph Conf)) → Set
```

Again, `_⊢MRSC*_↪_` is a "point-wise" version of `_⊢MRSC_↪_`:

```
h ⊢MRSC* cs ↪ gs = Pointwise.Rel (_⊢MRSC_↪_ h) cs gs
```

`_⊢MRSC_↪_` is defined by two rules

```
data _⊢MRSC_↪_ where
  mrsc-fold  : ∀ {h : History} {c}
    (f : Foldable h c) →
    h ⊢MRSC c ↪ back c
  mrsc-build : ∀ {h : History} {c}
    {cs : List Conf} {gs : List (Graph Conf)}
    (¬f : ¬ Foldable h c)
    (¬w : ¬ ↯ h) →
    (i : cs ∈ c ⇉)
    (s : (c :: h) ⊢MRSC* cs ↪ gs) →
    h ⊢MRSC c ↪ forth c gs
```

We can see that `_⊢NDSC_↪_` and `_⊢MRSC_↪_` differ only in that there is an additional condition `¬ ↯ h` in the rule `mrsc-build`.

The predicate `↯` is provided by the whistle, `↯ h` meaning that the history `h` is "dangerous". Unlike the rule `ndsc-build`, the rule `mrsc-build` is only applicable when `¬ ↯ h`, i.e. the history `h` is not dangerous.

Multi-result supercompilation is a special case of non-deterministic supercompilation, in the sense that any graph produced by multi-result supercompilation can also be produced by non-deterministic supercompilation:

```
MRSC→NDSC : ∀ {h : History} {c g} →
  h ⊢MRSC c ↪ g → h ⊢NDSC c ↪ g
```

A proof of this theorem can be found in `BigStepScTheorems.agda`.

3.6 Bar Whistles

Now we are going to give an alternative definition of multi-result supercompilation in form of a total function `naive-mrsc`. The termination of `naive-mrsc` is guaranteed by a "whistle".

In our model of big-step supercompilation whistles are assumed to be "inductive bars" [6] and are defined in Agda in the following way.

First of all, `BarWhistles.agda` contains the following declaration of `Bar D h`:

```
data Bar {A : Set} (D : List A → Set) :
         (h : List A) → Set where
  now   : {h : List A} (bz : D h) → Bar D h
  later : {h : List A} (bs : ∀ c → Bar D (c :: h)) → Bar D h
```

At the first glance, this declaration looks as a puzzle. But, actually, it is not as mysterious as it may seem.

We consider sequences of elements (of some type `A`), and a predicate `D`. If `D h` holds for a sequence `h`, `h` is said to be "dangerous".

`Bar D h` means that either (1) `h` is dangerous, i.e. `D h` is valid right now (the rule `now`), or (2) `Bar D (c :: h)` is valid for *all* possible `c :: h` (the rule `later`). Hence, for any continuation `c :: h` the sequence will *eventually* become dangerous.

The subtle point is that if `Bar D []` is valid, it implies that *any* sequence will eventually become dangerous.

A *bar whistle* is a record (see `BarWhistles.agda`)

```
record BarWhistle (A : Set) : Set₁ where
  field
    ↯   : (h : List A) → Set
    ↯:: : (c : A) (h : List A) → ↯ h → ↯ (c :: h)
    ↯?  : (h : List A) → Dec (↯ h)

    bar[] : Bar ↯ []
```

where

- `↯` is a predicate on sequences, `↯ h` meaning that the sequence `h` is dangerous.
- `↯::` postulates that if `↯ h` then `↯ (c :: h)` for all possible `c :: h`. In other words, if `h` is dangerous, so are all continuations of `h`.
- `↯?` says that `↯` is decidable.
- `bar[]` says that any sequence eventually becomes dangerous. (In Coquand's terms, `Bar ↯` is required to be "an inductive bar".)

3.7 A Function for Computing Cartesian Products

The functional specification of big-step multi-result supercompilation considered in the following section is based on the function `cartesian`:

```
cartesian2 : ∀ {A : Set} → List A → List (List A) → List (List A)
cartesian2 [] yss = []
cartesian2 (x :: xs) yss = map (_::_ x) yss ++ cartesian2 xs yss

cartesian : ∀ {A : Set} (xss : List (List A)) → List (List A)
cartesian [] = [ [] ]
cartesian (xs :: xss) = cartesian2 xs (cartesian xss)
```

`cartesian` takes as input a list of lists `xss`. Each list `xs ∈ xss` represents the set of possible values of the correspondent component.

Namely, suppose that `xss` has the form $xs_1, xs_2, \ldots, xs_k$ Then `cartesian` returns a list containing all possible lists of the form $x_1 :: x_2 :: \ldots :: x_k :: []$ where $x_i \in xs_i$. In Agda, this property of `cartesian` is formulated as follows:

```
∈*↔∈cartesian :
  ∀ {A : Set} {xs : List A} {yss : List (List A)} →
    Pointwise.Rel _∈_ xs yss ↔ xs ∈ cartesian yss
```

A proof of the theorem `∈*↔∈cartesian` can be found in `Util.agda`.

3.8 A Functional Specification of Big-Step Multi-Result Supercompilation

A functional specification of big-step multi-result supercompilation is given in the form of a total function (in `BigStepSc.agda`) that takes the initial configuration `c` and returns a list of residual graphs:

```
naive-mrsc : (c : Conf) → List (Graph Conf)
naive-mrsc′ : ∀ (h : History) (b : Bar ↯ h) (c : Conf) →
                List (Graph Conf)

naive-mrsc c = naive-mrsc′ [] bar[] c
```

`naive-mrsc` is defined in terms of a more general function `naive-mrsc′`, which takes more arguments: a history `h`, a proof `b` of the fact `Bar ↯ h`, and a configuration `c`.

Note that `naive-mrsc` calls `naive-mrsc′` with the empty history and has to supply a proof of the fact `Bar ↯ []`. But this proof is supplied by the whistle!

```
naive-mrsc′ h b c with foldable? h c
... | yes f = [ back c ]
... | no ¬f with ↯? h
... | yes w = []
... | no ¬w with b
... | now bz with ¬w bz
... | ()
naive-mrsc′ h b c | no ¬f | no ¬w | later bs =
  map (forth c)
      (concat (map (cartesian ∘ map
                        (naive-mrsc′ (c :: h) (bs c))) (c ⇉)))
```

The definition of `naive-mrsc′` is straightforward[5].

- If `c` is foldable to the history `h`, a back-node is generated and the function terminates.
- Otherwise, if `↯ h` (i.e. the history `h` is dangerous), the function terminates producing no graphs.
- Otherwise, `h` is not dangerous, and the configuration `c` can be decomposed. (Also there are some manipulations with the parameter `b` that will be explained later.)
- Thus `c ⇉` returns a list of lists of configurations. The function considers each `cs ∈ c ⇉`, and, for each `c′ ∈ cs` recursively calls itself in the following way:

  `naive-mrsc′ (c :: h) (bs c) c′`

[5] In Agda, `with e` means that the value of `e` has to be matched against the patterns preceded with `... |`. `()` is a pattern denoting a logically impossible case, for which reason `()` is not followed by a right hand side.

producing a list of residual graphs `gs′`. So, `cs` is transformed into gss, a list of lists of graphs. Note that
`length cs = length gss`.
- Then the function computes cartesian product `cartesian gss`, to produce a list of lists of graphs. Then the results corresponding to each `cs` $\in$ `c` $\rightrightarrows$ are concatenated by `concat`.
- At this moment the function has obtained a list of lists of graphs, and calls `map (forth c)` to turn each graph list into a forth-node.

The function `naive-mrsc` is correct (sound and complete) with respect to the relation `_⊢MRSC_↪_`:

```
⊢MRSC↪⇔naive-mrsc :
  {c : Conf} {g : Graph Conf} →
    [] ⊢MRSC c ↪ g ⇔ g ∈ naive-mrsc c
```

A proof of this theorem can be found in `BigStepScTheorems.agda`.

3.9 Why Does `naive-mrsc′` Always Terminate?

The problem with `naive-mrsc′` is that in the recursive call

```
naive-mrsc′ (c :: h) (bs c) c′
```

the history grows (`h` becomes `c :: h`), and the configuration is replaced with another configuration of unknown size (`c` becomes `c′`). Hence, these parameters do not become "structurally smaller".

But Agda's termination checker still accepts this recursive call, because the second parameter does become smaller (`later bs` becomes `bs c`). Note that the termination checker considers `bs` and `bs c` to be of the same "size". Since `bs` is smaller than `later bs` (a constructor is removed), and `bs` and `bs c` are of the same size, `bs c` is "smaller" than `later bs`.

Thus the purpose of the parameter `b` is to persuade the termination checker that the function terminates. If `lazy-mrsc` is reimplemented in a language in which the totality of functions is not checked, the parameter `b` is not required and can be removed.

4 Staging Big-Step Multi-Result Supercompilation

As was said above, we can decompose the process of supercompilation into two stages

$$\texttt{naive-mrsc} \doteq \langle\!\langle \_ \rangle\!\rangle \circ \texttt{lazy-mrsc}$$

At the first stage, `lazy-mrsc` generates a "lazy graph", which, essentially, is a "program" to be "executed" by $\langle\!\langle \_ \rangle\!\rangle$.

4.1 Lazy Graphs of Configurations

A `LazyGraph C` represents a finite set of graphs of configurations (whose type is `Graph C`).

```
data LazyGraph (C : Set) : Set where
  ∅     : LazyGraph C
  stop  : (c : C) → LazyGraph C
  build : (c : C) (lss : List (List (LazyGraph C))) → LazyGraph C
```

A lazy graph is a tree whose nodes are "commands" to be executed by the interpreter $\langle\!\langle \_ \rangle\!\rangle$.

The exact semantics of lazy graphs is given by the function $\langle\!\langle \_ \rangle\!\rangle$, which calls auxiliary functions $\langle\!\langle \_ \rangle\!\rangle*$ and $\langle\!\langle \_ \rangle\!\rangle\rightrightarrows$ (see `Graphs.agda`).

```
⟪_⟫ : {C : Set} (l : LazyGraph C) → List (Graph C)
⟪_⟫* : {C : Set} (ls : List (LazyGraph C)) →
    List (List (Graph C))
⟪_⟫⇉ : {C : Set} (lss : List (List (LazyGraph C))) →
    List (List (Graph C))
```

Here is the definition of the main function $\langle\!\langle \_ \rangle\!\rangle$:

```
⟪ ∅ ⟫ = []
⟪ stop c ⟫ = [ back c ]
⟪ build c lss ⟫ = map (forth c) ⟪ lss ⟫⇉
```

It can be seen that `∅` means "generate no graphs", `stop` means "generate a back-node and stop".

The most interesting case is a build-node `build c lss`, where `c` is a configuration and `lss` a list of lists of lazy graphs. Recall that, in general, a configuration can be decomposed into a list of configurations in several different ways. Thus, each `ls` $\in$ `lss` corresponds to a decomposition of `c` into a number of configurations $c_1, \ldots c_k$. By supercompiling each c_i we get a collection of graphs that can be represnted by a lazy graph ls_i.

The function $\langle\!\langle \_ \rangle\!\rangle*$ considers each lazy graph in a list of lazy graphs `ls`, and turns it into a list of graphs:

```
⟪ [] ⟫* = []
⟪ l :: ls ⟫* = ⟪ l ⟫ :: ⟪ ls ⟫*
```

The function $\langle\!\langle \_ \rangle\!\rangle\rightrightarrows$ considers all possible decompositions of a configuration, and for each decomposition computes all possible combinations of subgraphs:

```
⟪ [] ⟫⇉ = []
⟪ ls :: lss ⟫⇉ = cartesian ⟪ ls ⟫* ++ ⟪ lss ⟫⇉
```

There arises a natural question: why ⟪_⟫* is defined by explicit recursion, while it does exactly the same job as would do `map` ⟪_⟫? The answer is that Agda's termination checker does not accept `map` ⟪_⟫, because it cannot see that the argument in the recursive calls to ⟪_⟫ becomes structurally smaller. For the same reason ⟪_⟫⇉ is also defined by explicit recursion.

4.2 A Functional Specification of Lazy Multi-Result Supercompilation

Given a configuration `c`, the function `lazy-mrsc` produces a lazy graph.

```
lazy-mrsc : (c : Conf) → LazyGraph Conf
```

`lazy-mrsc` is defined in terms of a more general function `lazy-mrsc′`

```
lazy-mrsc′ : ∀ (h : History) (b : Bar ↯ h)
    (c : Conf) → LazyGraph Conf
lazy-mrsc c = lazy-mrsc′ [] bar[] c
```

The general structure of `lazy-mrsc′` is very similar (see Section 3) to that of `naive-mrsc′`, but, unlike `naive-mrsc`, it does not build Cartesian products immediately.

```
lazy-mrsc′ h b c with foldable? h c
... | yes f = stop c
... | no ¬f with ↯? h
... | yes w = ∅
... | no ¬w with b
... | now bz with ¬w bz
... | ()
lazy-mrsc′ h b c | no ¬f | no ¬w | later bs =
  build c (map (map (lazy-mrsc′ (c :: h) (bs c))) (c ⇉))
```

Let us compare the most interesting parts of `naive-mrsc` and `lazy-mrsc`:

```
map (forth c)
    (concat (map (cartesian ∘
                    map (naive-mrsc′ (c :: h) (bs c))) (c ⇉)))
...
build c (map (map (lazy-mrsc′ (c :: h) (bs c))) (c ⇉))
```

Note that `cartesian` disappears from `lazy-mrsc`.

4.3 Correctness of `lazy-mrsc` and ⟪_⟫

`lazy-mrsc` and ⟪_⟫ are correct with respect to `naive-mrsc`. In Agda this is formulated as follows:

```
naive≡lazy : (c : Conf) → naive-mrsc c ≡ ⟪ lazy-mrsc c ⟫
```

In other words, for any initial configuraion `c`, ⟪ `lazy-mrsc c` ⟫ returns the same list of graphs (the same configurations in the same order!) as would return `naive-mrsc c`.

A formal proof of `naive≡lazy` can be found in `BigStepScTheorems.agda`.

5 Cleaning Lazy Graphs

As was said in Section 2.3, we can replace filtering of graphs with cleaning of lazy graphs

```
filter ∘ naive-mrsc ≗ ⟪_⟫ ∘ clean ∘ lazy-mrsc
```

In `Graphs.agda` there are defined a number of filters and corresponding cleaners.

5.1 Filter `fl-bad-conf` and Cleaner `cl-bad-conf`

Configurations represent states of a computation process. Some of these states may be "bad" with respect to the problem that is to be solved by means of supercompilation.

Given a predicate `bad` that returns `true` for "bad" configurations, `fl-bad-conf bad gs` removes from `gs` the graphs that contain at least one "bad" configuration.

The cleaner `cl-bad-conf` corresponds to the filter `fl-bad-conf`. `cl-bad-conf` exploits the fact that "badness" is monotonic, in the sense that a single "bad" configuration spoils the whole graph.

```
fl-bad-conf : {C : Set} (bad : C → Bool) (gs : List (Graph C)) →
  List (Graph C)

cl-bad-conf : {C : Set} (bad : C → Bool) (l : LazyGraph C) →
  LazyGraph C
```

`cl-bad-conf` is correct with respect to `fl-bad-conf`:

```
cl-bad-conf-correct : {C : Set} (bad : C → Bool) →
  ⟪_⟫ ∘ cl-bad-conf bad ≗ fl-bad-conf bad ∘ ⟪_⟫
```

A formal proof of this theorem is given in `GraphsTheorems.agda`.

It is instructive to take a look at the implementation of `cl-bad-conf` in `Graphs.agda`, to get the general idea of how cleaners are really implemented:

```
cl-bad-conf : {C : Set} (bad : C → Bool) (l : LazyGraph C) →
  LazyGraph C

cl-bad-conf⇉ : {C : Set} (bad : C → Bool)
  (lss : List (List (LazyGraph C))) → List (List (LazyGraph C))
```

```
cl-bad-conf* : {C : Set} (bad : C → Bool)
  (ls : List (LazyGraph C)) → List (LazyGraph C)

cl-bad-conf bad ∅ = ∅
cl-bad-conf bad (stop c) =
  if bad c then ∅ else (stop c)
cl-bad-conf bad (build c lss) =
  if bad c then ∅ else (build c (cl-bad-conf⇉ bad lss))

cl-bad-conf⇉ bad [] = []
cl-bad-conf⇉ bad (ls :: lss) =
  cl-bad-conf* bad ls :: (cl-bad-conf⇉ bad lss)

cl-bad-conf* bad [] = []
cl-bad-conf* bad (l :: ls) =
  cl-bad-conf bad l :: cl-bad-conf* bad ls
```

5.2 Cleaner `cl-empty`

`cl-empty` is a cleaner that removes subtrees of a lazy graph that represent empty sets of graphs[6].

```
cl-empty : {C : Set} (l : LazyGraph C) → LazyGraph C
```

`cl-bad-conf` is correct with respect to $\langle\!\langle \_ \rangle\!\rangle$:

```
cl-empty-correct : ∀ {C : Set} (l : LazyGraph C) →
  ⟪ cl-empty l ⟫ ≡ ⟪ l ⟫
```

A formal proof of this theorem is given in `GraphsTheorems.agda`.

5.3 Cleaner `cl-min-size`

The function `cl-min-size`

```
cl-min-size : ∀ {C : Set} (l : LazyGraph C) → ℕ × LazyGraph C
```

takes as input a lazy graph `l` and returns either `(0 , ∅)`, if `l` contains no graphs, or a pair `(k , l′)`, where `l′` is a lazy graph, representing a single graph `g′` of minimal size `k`[7].

More formally,

- $\langle\!\langle$ `l′` $\rangle\!\rangle$ = `[ g′ ]`.
- `graph-size g′ = k`

[6] Empty sets of graphs may appear when multi-result supercompilation gets into a blind alley: the whistle blows, but neither folding nor rebuilding is possible.

[7] This cleaner is useful in cases where we use supercompilation for problem solving and want to find a solution of minimum size.

– `k` $\leq$ `graph-size g` for all `g` $\in$ $\langle\!\langle$ `l` $\rangle\!\rangle$.

The main idea behind `cl-min-size` is that, if we have a node `build c lss`, then we can clean each `ls` $\in$ `lss`, to produce `lss′`, a cleaned version of `lss`.

Let us consider an `ls` $\in$ `lss`. We can clean with `cl-min-size` each `l` $\in$ `ls` to obtain `ls′` a new list of lazy graphs . If $\emptyset$ $\in$ `ls′`, we replace the node `build c lss` with $\emptyset$. The reason is that computing the Cartesian product for `ls′` would produce an empty set of results. Otherwise, we replace `build c lss` with `build c lss′`.

The details of how `cl-min-size` is implemented can be found in `Graphs.agda`.

A good thing about `cl-min-size` is it cleans any lazy graph `l` in linear time with respect to the size of `l`.

6 Codata and Corecursion: Cleaning before Whistling

By using codata and corecursion, we can decompose `lazy-mrsc` into two stages

`lazy-mrsc` $\triangleq$ `prune-cograph` $\circ$ `build-cograph`

where `build-cograph` constructs a (potentially) infinite tree, while `prune-cograph` traverses this tree and turns it into a lazy graph (which is finite).

6.1 Lazy Cographs of Configurations

A `LazyCograph C` represents a (potentially) infinite set of graphs of configurations whose type is `Graph C` (see `Cographs.agda`).

```
data LazyCograph (C : Set) : Set where
  ∅     : LazyCograph C
  stop  : (c : C) → LazyCograph C
  build : (c : C)
      (lss : ∞(List (List (LazyCograph C)))) → LazyCograph C
```

Note that `LazyCograph C` differs from `LazyGraph C` the evaluation of `lss` in build-nodes is delayed.

6.2 Building Lazy Cographs

Lazy cographs are produced by the function `build-cograph`

```
build-cograph : (c : Conf) → LazyCograph Conf
```

which can be derived from the function `lazy-mrsc` by removing the machinery related to whistles.

`build-cograph` is defined in terms of a more general function `build-cographs′`.

```
build-cograph′ : (h : History) (c : Conf) → LazyCograph Conf
build-cograph c = build-cograph′ [] c
```

The definition of `build-cograph′` uses auxiliary functions `build-cograph⇉` and `build-cograph*`, while the definition of `lazy-mrsc` just calls `map` at corresponding places. This is necessary in order for `build-cograph′` to pass Agda's "productivity" check.

```
build-cograph⇉ : (h : History) (c : Conf)
    (css : List (List Conf)) → List (List (LazyCograph Conf))

build-cograph* : (h : History)
    (cs : List Conf) → List (LazyCograph Conf)

build-cograph′ h c with foldable? h c
... | yes f = stop c
... | no ¬f =
  build c (♯ build-cograph⇉ h c (c ⇉))

build-cograph⇉ h c [] = []
build-cograph⇉ h c (cs :: css) =
  build-cograph* (c :: h) cs :: build-cograph⇉ h c css

build-cograph* h [] = []
build-cograph* h (c :: cs) =
  build-cograph′ h c :: build-cograph* h cs
```

6.3 Pruning Lazy Cographs

A lazy cograph can be pruned by means of the function `prune-cograph` to obtain a finite lazy graph.

```
prune-cograph : (l : LazyCograph Conf) → LazyGraph Conf
```

which can be derived from the function `lazy-mrsc` by removing the machinery related to generation of nodes (since it only consumes nodes that have been generated by `build-cograph`).

`prune-cograph` is defined in terms of a more general function `prune-cograph′`:

```
prune-cograph l = prune-cograph′ [] bar[] l
```

The definition of `prune-cograph′` uses the auxiliary function `prune-cograph*`.

```
prune-cograph* : (h : History) (b : Bar ↯ h)
  (ls : List (LazyCograph Conf)) → List (LazyGraph Conf)

prune-cograph′ h b ∅ = ∅
prune-cograph′ h b (stop c) = stop c
prune-cograph′ h b (build c lss) with  ↯? h
... | yes w = ∅
... | no ¬w with b
```

```
... | now bz with ¬w bz
... | ()
prune-cograph′ h b (build c lss) | no ¬w | later bs =
  build c (map (prune-cograph* (c :: h) (bs c)) (♭ lss))

prune-cograph* h b [] = []
prune-cograph* h b (l :: ls) =
  prune-cograph′ h b l :: (prune-cograph* h b ls)
```

Note that, when processing a node `build c lss`, the evaluation of `lss` has to be explicitly forced by ♭.

`prune-cograph` and `build-cograph` are correct with respect to `lazy-mrsc`:

```
prune∘build-correct :
   prune-cograph ∘ build-cograph ≗ lazy-mrsc
```

A proof of this theorem can be found in `Cographs.agda`.

6.4 Promoting some Cleaners over the Whistle

Suppose `clean∞` is a cograph cleaner such that

```
clean ∘ prune-cograph ≗ prune-cograph ∘ clean∞
```

then

```
clean ∘ lazy-mrsc ≗
  clean ∘ prune-cograph ∘ build-cograph ≗
  prune-cograph ∘ clean∞ ∘ build-cograph
```

The good thing about `build-cograph` and `clean∞` is that they work in a lazy way, generating subtrees by demand. Hence, evaluating

```
⟪ prune-cograph ∘ (clean∞ (build-cograph c))  ⟫
```

may be less time and space consuming than evaluating

```
⟪ clean (lazy-mrsc c) ⟫
```

In `Cographs.agda` there is defined a cograph cleaner `cl-bad-conf∞` that takes a lazy cograph and prunes subrees containing bad configurations, returning a lazy subgraph (which can be infinite):

```
cl-bad-conf∞ : {C : Set} (bad : C → Bool) (l : LazyCograph C) →
  LazyCograph C
```

`cl-bad-conf∞` is correct with respect to `cl-bad-conf`:

```
  cl-bad-conf∞-correct : (bad : Conf → Bool) →
    cl-bad-conf bad ∘ prune-cograph ≗
      prune-cograph ∘ cl-bad-conf∞ bad
```

A proof of this theorem can be found in `Cographs.agda`.

7 Related Work

The idea that supercompilation can produce a compact representation of a collection of residual graphs is due to Grechanik [7,8]. In particular, the data structure 'LazyGraph C' we use for representing the results of the first phase of the staged multi-result supercompiler can be considered as a representation of "overtrees", which was informally described in [7].

Big-step supercompilation was studied and implemented by Bolingbroke and Peyton Jones [4]. Our approach differs in that we are interested in applying supercompilation to problem solving. Thus

- We consider multi-result supercompilation, rather than single-result supercompilation.
- Our big-step supercompilation constructs graphs of configurations in an explicit way, because the graphs are going to be filtered and/or analyzed at a later stage.
- Bolingbroke and Peyton Jones considered big-step supercompilation in functional form, while we have studied both a relational specification of big-step supercompilation and the functional one and have proved the correctness of the functional specification with respect to the relational one.

A relational specification of single-result supercompilation was suggested by Klimov [11], who argued that supercompilation relations can be used for simplifying proofs of correctness of supercompilers. Later, Klyuchnikov [20] used a supercompilation relation for proving the correctness of a small-step single-result supercompiler for a higher-order functional language. In the present work we consider a supercompilation relation for a *big-step multi-result* supercompilation.

We have developed an abstract model of big-step multi-result supercompilation in the language Agda and have proved a number of properties of this model. This model, in some respects, differs from the other models of supercompilation.

The MRSC Toolkit by Klyuchnikov and Romanenko [23] abstracts away some aspects of supercompilation, such as the structure of configurations and the details of the subject language. However, the MRSC Toolkit is a framework for implementing small-step supercompilers, while our model in Agda [2] formalizes big-step supercompilation. Besides, the MRSC Toolkit is implemented in Scala, for which reason it currently provides no means for neither formulating nor proving theorems about supercompilers implemented with the MRSC Toolkit.

Krustev was the first to formally verify a simple supercompiler by means of a proof assistant [26]. Unlike the MRSC Toolkit and our model of supercompilation, Krustev deals with a specific supercompiler for a concrete subject language. (Note, however, that also the subject language is simple, it is still Turing complete.)

In another paper Krustev presents a framework for building formally verifiable supercompilers [27]. It is similar to the MRSC in that it abstracts away some details of supercompilation, such as the subject language and the structure of

configurations, providing, unlike the MRSC Toolkit, means for formulating and proving theorems about supercompilers.

However, in both cases Krustev deals with single-result supercompilation, while the primary goal of our model of supercompilation is to formalize and investigate some subtle aspects of multi-result supercompilation.

8 Conclusions

When using supercompilation for problem solving, it seems natural to produce a collection of residual graphs of configurations by multi-result supercompilation and then to filter this collection according to some criteria. Unfortunately, this may lead to combinatorial explosion.

We have suggested the following solution.

- Instead of generating and filtering a collection of residual graphs of configurations, we can produce a compact representation for the collection of graphs (a "lazy graph"), and then analyze this representation.
- This compact representation can be derived from a (big-step) multi-result supercompiler in a systematic way by (manually) staging this supercompiler to represent it as a composition of two stages. At the first stage, some graph-building operations are delayed to be later performed at the second stage.
- The result produced by the first stage is a "lazy graph", which is, essentially, a program to be interpreted at the second stage, to actually generate a collection of residual graphs.
- The key point of our approach is that a number of problems can be solved by directly analyzing the lazy graphs, rather than by actually generating and analyzing the collections of graphs they represent.
- In some cases of practical importance, the analysis of a lazy graph can be performed in linear time.

Acknowledgements

The authors express their gratitude to the participants of the Refal seminar at Keldysh Institute for useful comments and fruitful discussions.

References

1. Staged multi-result supercompilation: filtering before producing, 2013. `https://github.com/sergei-romanenko/staged-mrsc-agda`.
2. The Agda Wiki, 2013. `http://wiki.portal.chalmers.se/agda/`.
3. D. Bjørner, M. Broy, and I. V. Pottosin, editors. *Perspectives of Systems Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25–28, 1996*, volume 1181 of *Lecture Notes in Computer Science*. Springer, 1996.

4. M. Bolingbroke and S. Peyton Jones. Supercompilation by evaluation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 135–146, New York, NY, USA, 2010. ACM.
5. E. Clarke, I. Virbitskaite, and A. Voronkov, editors. *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011*, volume 7162 of *Lecture Notes in Computer Science*. Springer, 2012.
6. T. Coquand. About Brouwer's fan theorem. *Revue internationale de philosophie*, 230:483–489, 2003.
7. S. A. Grechanik. Overgraph representation for multi-result supercompilation. In Klimov and Romanenko [18], pages 48–65.
8. S. A. Grechanik. Supercompilation by hypergraph transformation. *Keldysh Institute Preprints*, (26), 2013.
   URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2013-26`.
9. A. V. Klimov. An approach to supercompilation for object-oriented languages: the Java Supercompiler case study. In Nemytykh [34], pages 43–53.
10. A. V. Klimov. JVer project: Verification of Java programs by the Java Supercompiler. `http://pat.keldysh.ru/jver/`, 2008.
11. A. V. Klimov. A program specialization relation based on supercompilation and its properties. In Nemytykh [34], pages 54–77.
12. A. V. Klimov. A Java Supercompiler and its application to verification of cache-coherence protocols. In Pnueli et al. [36], pages 185–192.
13. A. V. Klimov. Multi-result supercompilation in action: Solving coverability problem for monotonic counter systems by gradual specialization. In *International Workshop on Program Understanding, PU 2011, Novososedovo, Russia, July 2–5, 2011*, pages 25–32. Ershov Institute of Informatics Systems, Novosibirsk, 2011.
14. A. V. Klimov. Yet another algorithm for solving coverability problem for monotonic counter systems. In V. Nepomnyaschy and V. Sokolov, editors, *PSSV*, pages 59–67. Yaroslavl State University, 2011.
15. A. V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In Clarke et al. [5], pages 193–209.
16. A. V. Klimov, I. G. Klyuchnikov, and S. A. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation. *Keldysh Institute Preprints*, (19), 2012.
   URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2012-19`.
17. A. V. Klimov, I. G. Klyuchnikov, and S. A. Romanenko. Implementing a domain-specific multi-result supercompiler by means of the MRSC toolkit. *Keldysh Institute Preprints*, (24), 2012.
   URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2012-24`.
18. A. V. Klimov and S. A. Romanenko, editors. *Third International Valentin Turchin Workshop on Metacomputation, Pereslavl-Zalessky, Russia, July 5–9, 2012*. Ailamazyan University of Pereslavl, Pereslavl-Zalessky, 2012.
19. I. G. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. *Keldysh Institute Preprints*, (63), 2009.
   URL: `http://library.keldysh.ru/preprint.asp?id=2009-63`.
20. I. G. Klyuchnikov. Supercompiler HOSC: proof of correctness. *Keldysh Institute Preprints*, (31), 2010.
   URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2010-31`.
21. I. G. Klyuchnikov and S. A. Romanenko. SPSC: a simple supercompiler in scala. In *PU'09 (International Workshop on Program Understanding)*, 2009.

22. I. G. Klyuchnikov and S. A. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In Pnueli et al. [36], pages 193–205.
23. I. G. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. *Keldysh Institute Preprints*, (77), 2011. URL: `http://library.keldysh.ru/preprint.asp?lg=e&id=2011-77`.
24. I. G. Klyuchnikov and S. A. Romanenko. Formalizing and implementing multi-result supercompilation. In Klimov and Romanenko [18], pages 142–164.
25. I. G. Klyuchnikov and S. A. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In Clarke et al. [5], pages 210–226.
26. D. N. Krustev. A simple supercompiler formally verified in Coq. In A. P. Nemytykh, editor, *META*, pages 102–127. Ailamazyan University of Pereslavl, Pereslavl-Zalessky, 2010.
27. D. N. Krustev. Towards a framework for building formally verified supercompilers in Coq. In H.-W. Loidl and R. Peña, editors, *Trends in Functional Programming*, volume 7829 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2012.
28. A. P. Lisitsa and A. P. Nemytykh. SCP4: Verification of protocols. `http://refal.botik.ru/protocols/`.
29. A. P. Lisitsa and A. P. Nemytykh. Verification of MESI cache coherence protocol. `http://www.csc.liv.ac.uk/~alexei/VeriSuper/node5.html`.
30. A. P. Lisitsa and A. P. Nemytykh. Towards verification via supercompilation. In *COMPSAC*, pages 9–10. IEEE Computer Society, 2005.
31. A. P. Lisitsa and A. P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.
32. A. P. Lisitsa and A. P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.
33. A. P. Nemytykh. The supercompiler SCP4: General structure. In M. Broy and A. V. Zamulin, editors, *PSI*, volume 2890 of *Lecture Notes in Computer Science*, pages 162–170. Springer, 2003.
34. A. P. Nemytykh, editor. *First International Workshop on Metacomputation in Russia, Pereslavl-Zalessky, Russia, July 2–5, 2008*. Ailamazyan University of Pereslavl, Pereslavl-Zalessky, 2008.
35. A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: Experiments with a supercompiler. In Bjørner et al. [3], pages 249–260.
36. A. Pnueli, I. Virbitskaite, and A. Voronkov, editors. *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Akademgorodok, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*. Springer, 2010.
37. M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Dept. of Computer Science, University of Copenhagen, 1994.
38. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
39. M. H. B. Sørensen. Convergence of program transformers in the metric space of trees. *Science of Computer Programming*, 37(1-3):163–205, May 2000.
40. W. Taha. A gentle introduction to multi-stage programming. In C. Lengauer, D. S. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 30–50. Springer, 2003.

41. W. Taha. A gentle introduction to multi-stage programming, part II. In R. Lämmel, J. Visser, and J. Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 260–290. Springer, 2007.
42. V. F. Turchin. A supercompiler system based on the language Refal. *ACM SIGPLAN Not.*, 14(2):46–54, 1979.
43. V. F. Turchin. The language Refal: The theory of compilation and metasystem analysis. Technical Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.
44. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.
45. V. F. Turchin. Supercompilation: Techniques and results. In Bjørner et al. [3], pages 227–248.
46. V. F. Turchin, R. M. Nirenberg, and D. V. Turchin. Experiments with a supercompiler. In *LFP '82: Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, August 15-18, 1982, Pittsburgh, PA, USA*, pages 47–55. ACM, 1982.

Supercompiling with Staging

Jun Inoue

INRIA Paris-Rocquencourt
DI École Normale Supérieure, Paris
Jun.Inoue@inria.fr

Abstract. Supercompilation is a powerful program optimization framework which Sørensen et al. showed to subsume, and exceed, partial evaluation and deforestation. Its main strength is that it optimizes a conditional branch by assuming the branch's guard tested true, and that it can propagate this information to data that are not directly examined in the guard. We show that both of these features can be mimicked in multi-stage programming, a code generation framework, by modifying metadata attached to generated code in-place. This allows for explicit, programmer-controlled supercompilation with well-defined semantics as to where, how, and whether a program is optimized. Our results show that staging can go beyond partial evaluation, with which it originated, and is also useful for writing libraries in high-level style where failing to optimize away the overheads is unacceptable.

Keywords: Supercompilation, Multi-Stage Programming, Functional Programming

1 Introduction

Supercompilation is a powerful metacomputation framework known to subsume other systems like deforestation and partial evaluation [13]. A key benefit of such frameworks is to enable the use of abstractions without runtime penalties. For example, functional programs often split a loop into a function that produces a stream of items and another function that performs work on each item. This abstraction with streams greatly improves modularity, at the cost of more allocation and time spent inspecting the stream. Supercompilation can eliminate the stream, resolving the tension between abstraction and performance.

Supercompilation is usually studied as a fully automatic optimization, but this approach has pros and cons. In exchange for the convenience of automation, programmers lose control over when and how optimization happens, and it can be difficult to tell whether supercompilation will eliminate a specific abstraction used in a given source program. This can be problematic if failing to optimize is unacceptable, as in embedded systems and high-performance computing.

Multi-stage programming (MSP) [17] has evolved as a tool to solve similar problems with automation in the context of partial evaluation (PE). For instance, the MSP language MetaOCaml can optimize the power function as follows.

```
let rec power n x = if n = 1 then x else x * power (n-1) x
let rec genpow n x = if n = 1 then x else .<.~x * .~(genpow (n-1) x)>.
let rec stpow n = !. .<fun x → .~(genpow n .<x>.)>.
```

`power` computes x^n, while `stpow` generates loop-unrolled `power` for concrete values of `n` using MetaOCaml's three *staging constructs*. Brackets `.<`e`>.` delay the expression e. An escape `.~`e must occur inside brackets and instructs e to be evaluated without delay. The result must be of the form `.<`e'`>.`, and e' replaces `.~`e. Run `!.`e compiles and runs the delayed expression returned by e. These constructs are like LISP's quasiquote, unquote, and eval but are hygienic, i.e. preserves static scope [1]. In this example, `genpow n .<x>.` generates the code `.<x*x*...x>.` with `n` copies of `x`, while `stpow` places that inside a binder to get `.<fun x → x*x*...x>.` and compiles it by `!.` It is evident from the source code that `genpow` completely unrolls the recursion seen in `power` and produces code containing only `*`, because that's the only operation occurring inside brackets.

In this paper, we bring this kind of explicit programmer control to supercompilation through MSP techniques that mimic positive supercompilation [12,13]. Most importantly, we express the introduction and propagation of assumptions under conditionals that Sørensen et al. [13] identified as the key improvements that supercompilation makes over PE and deforestation. For example, in

```
let rec contrived zs =
  let f xs ys = match xs with [] → length xs + length ys
                            | w::ws → length xs
  in f zs (1::zs)
and length ws = match ws with [] → 0
                            | _::ws → 1 + length ws
```

positive supercompilation optimizes the first branch of the `match` by *assuming* `xs = []` and simplifying the branch body, which gives `0 + length ys`. Moreover, noting `ys` shares a substructure with `xs`, it *propagates* the assumption `xs = []` to `ys = [1]`, optimizing the whole branch to just `1`. By pattern-matching on `xs`, we learn something about `ys`, and the supercompiler tracks this knowledge.

In our technique of *delimited assumptions*, we manipulate not raw code values like `.<x>.` in the power example, but a wrapper that attaches information about the value `x` must have. We update this metadata when we generate a branch of a conditional to reflect any new assumptions. We modify the metadata in place to propagate the change to all copies of the data. This modification is undone by a dynamic wind when that branch is left, giving the assumption a (dynamic) scope delimited by the conditional. We show in this paper how this technique, combined with auxiliary techniques for ensuring termination of the generator, can achieve a great deal of the effects of supercompilation.

1.1 Contributions

We will use the `contrived` function above as a running example to illustrate the main techniques. Specifically, after reviewing in more detail how the positive supercompiler works (Section 2):

- We introduce the technique of delimited assumptions, which combines partially static data [11] with judicious uses of mutation to achieve the introduction and propagation of assumptions explained above (Section 3).
- We show memoization techniques for ensuring termination of the generator, explaining unique challenges posed by supercompilation (Section 4). Briefly, conditionals are converted to functions and invoked whenever a conditional of the same form is encountered, where the criterion for sameness must be modeled after α-invariant folding.
- We show that the techniques in this paper are sufficient to specialize a naïve string search algorithm to the Knuth-Morris-Pratt (KMP) algorithm [8], which is a staple test case for supercompilation (Section 5). This example motivates a technique called delimited aliasing which ensures static information is properly retained during memoization.

A heavily commented MetaOCaml source file containing all nontrivial code of this paper is available from the author's homepage. However, note that some parts of the code were shortened or omitted due to space limitations.

2 Background: Supercompilation

In this section we briefly review Sørensen et al.'s positive supercompiler [13]. We use the `contrived` function from the introduction as a running example. When asked to optimize the `contrived` function, the supercompiler starts a process called *driving* on the body of the function, reducing it as much as possible:

```
let f xs ys = match xs with [] → length xs + length ys
                          | w::ws → length xs
in f zs (1::zs)
⇓
match zs with [] → length zs + length (1::zs)
            | w::ws → length zs
```

where ⇓ denotes reduction – note that an open term is being reduced, with `zs` free. Now the code is at a pattern-match that cannot be resolved statically. In that case, driving replaces the scrutinee in each branch with the corresponding pattern:

```
match zs with [] → length [] + length (1::[])
            | w::ws → length (w::ws)
```

Note that `zs` is replaced by `[]` in the first branch but by `w::ws` in the second.

This substitution implements the *introduction of assumptions* mentioned in the introduction: the supercompiler rewrites each branch with the knowledge that the scrutinee must have a particular form in order for that branch to be entered. Furthermore, both calls to `length` in the first branch benefit by introducing the assumption `zs = []`. In the original source program, the scrutinee was `xs`, whereas the second call's argument was `ys`; the β substitution during

the reduction step (shown as ⇓ above) has exposed the sharing of the substructure `zs` in these two variables, so that the assumption introduced on (what used to be) `xs` *propagates* to (what used to be) `ys`. Put another way:

- Assumptions are introduced by replacing the scrutinee with patterns.
- Assumptions are propagated by sharing substructures.

The main idea behind delimited assumptions is that we can imitate both of these mechanisms by mutating metadata on a delayed variable.

After assumptions are introduced and propagated, the rewritten branch bodies are driven separately; however, blindly doing so can lead to non-termination. For example, driving the second branch by unrolling `length` gives

```
length (w::ws)
⇓
match w::ws with [] → 0
           | _::ws' → 1 + length ws'
⇓
1 + length ws
```

Note the `match` statement can be resolved statically, so no assumptions are introduced. The supercompiler at this point drives each argument of `+` separately. The left operand is in normal form, so it turns to the right operand, `length ws`.

```
length ws
⇓
match ws with [] → 0
         | _::ws' → 1 + length ws'
```

But the second branch is in a form already encountered before, so this unrolling can go on forever.

To avoid infinite unrolling, the positive supercompiler *lambda-lifts* and memoizes each statically unresolvable `match`. After introducing assumptions, but before driving each branch, the supercompiler places the whole `match` expression in a new top-level function whose parameters are the free variables of the expression.

```
let rec newfun xs =
  match xs with [] → 0
           | _::ws' → 1 + length ws'
```

When the supercompiler encounters the same `match` while driving the branches of `newfun`, where two terms are the "same" iff lambda-lifting them gives α-equivalent functions, then it emits a call to `newfun` instead of driving the same term again. For example, driving the `length ws'` in the second branch of the `match` in `newfun` replaces it by `newfun ws'`.

Put together, the supercompiler compiles the `contrived` function into

```
let rec contrived zs =
  match zs with [] → 0 + (1 + 0)
           | w::ws → 1 + length ws
and length ws = match ws with [] → 0
                       | _::ws → 1 + length ws
```

```
type ('s,'d) sd =
   { mutable dynamic : 'd code;
     mutable static : 's option; }
type ('s,'d) ps_cell =
 | Nil
 | Cons of ('s,'d) sd * ('s,'d) psl
and ('s,'d) psl =
 (('s,'d) ps_cell, 'd list) sd
(*unknown : 'd code → ('s,'d) sd*)
let unknown x =
   { dynamic = x; static = None }
(*forget : ('a,'b) sd → 'b code*)
let forget x = x.dynamic

(* assuming_eq : ('a, 'b) sd → 'a
   → (unit → 'c) → 'c *)
let assuming_eq x v thunk =
  let saved = x.static in
  try x.static <- Some v;
      let ret = thunk () in
      x.static <- saved; ret
  with e → x.static <- saved;
            raise e

(*dfun : (('a, 'b) sd → 'c code)
      → ('b → 'c) code*)
let dfun f =
 .<fun x → .~(f (unknown .<x>.))>.

(* match_ls :
    (('a,'b) ps_cell,'b list) sd
    → (unit → 'c code)
    → (('a,'b) sd → ('a,'b) psl
       → 'c code)
    → 'c code
 *)
let match_ls ls for_nil for_cons =
  match ls.static with
  | Some Nil → for_nil ()
  | Some (Cons (x,xs)) →
     for_cons x xs
  | None →
    .<match .~(forget ls) with
      | [] → .~(assuming_eq ls Nil
                  for_nil)
      | x::xs →
       .~(let x = unknown .<x>.
          and xs = unknown .<xs>.
          in assuming_eq
               ls (Cons (x,xs))
               (fun () →
                  for_cons x xs))>.
```

Fig. 1: Data types and functions implementing delimited assumptions.

In general, driving stops when the term under consideration reaches either a normal form or a memoized form. This heuristic is called *α-invariant folding*. Stronger termination heuristics are possible and implemented usually as *generalization*, but we will not deal with that aspect in this paper.

3 Delimited Assumptions

Driving follows the execution of its input program with three mechanisms: reduction of open terms, introduction of assumptions, and propagation of assumptions. As seen in the `power` example from the introduction, reduction of open terms is handled very naturally with MSP, as delayed variables can be manipulated like values and injected into generated code. Effectively, inserting brackets and escapes to force evaluation under binders corresponds to implementing the reduction part of driving. The trickier part is the handling of assumptions.

Figure 1 shows types and functions used to handle assumptions with MSP. Whereas the `power` example directly manipulated raw code values of the form `.<x>.`, the delimited assumption technique uses *static-dynamic values*, of type

`sd`. Here, "dynamic" means delayed by brackets, and "static" means not delayed. The `sd` type carries a dynamic value `.<x>.` and a static description of `x`'s dynamic value (i.e. of the value `x` will have when the generated code is run). The type of `x`'s value is `'d`, and the type of the static description is `'s`. Static-dynamic values are created by `unknown`, which attaches void static information to a dynamic value, and cast back to a dynamic value with `forget`, which discards static information. An example is seen in `dfun`, which generates a dynamic `fun`, wraps the parameter in `unknown`, and passes that to a callback to generate the body.

Static knowledge is often partial. For example, we might know that a dynamic list `xs` must be a cons cell `x::xs'` but not the value of `x` or whether `xs'` is also a cons cell. We need to mix in `sd` throughout data structures to represent such partial knowledge, which for the list type gives the partially static list type, `psl`. The `ps_cell` type encodes one cell worth of static information: empty or not, and if nonempty, the static-dynamic representations of the head and tail. The `psl` type is a static-dynamic type whose dynamic component is a `list` and whose static component is `ps_cell`.

The static information is manipulated during a call to `match_ls`, which looks deliberately like a `match` on a list:

```
match_ls xs (fun () → .<"empty">.) (fun x xs' → .<"nonempty">)
```

Conceptually, this function is a dynamic `match` whose branches are generated by the two callbacks, but it avoids generating a `match` at all if the static information on `xs` tells us the outcome, e.g. whether the list is empty or a cons cell. This optimization is implemented in the first half of `match_ls` – if static information is available, `match_ls` calls only one of the callbacks. However, if static information is unavailable, `match_ls` generates a dynamic `match`, then wraps pattern variables (if any) in `sd` and invokes the callbacks. Moreover, the scrutinee's static information is destructively updated to reflect which branch was taken: to `Nil` in the `[]` branch, and to `Cons` in the `x::xs` branch. This update is undone when the callback returns, so the assumption's lifetime is delimited by the `match` branch in which it was introduced – hence the name *delimited assumption*. This modification and restoration of static information is done in `assuming_eq`.

Note that the update by `assuming_eq` is done by mutation. By destructively updating static information, all copies of the data see the update. For example, the `contrived` function in the introduction can be staged as follows.

```
let rec gen_contrived () = dfun (fun zs →
  let f xs ys = match_ls xs
                (fun () → .<.~(gen_len xs) + .~(gen_len ys)>.)
                (fun _ ws → .<(* discussed later *)>.)
  in f zs (cons (known 1 .<1>.) zs)))
and gen_len ws = match_ls ws (fun () → .<0>.)
                             (fun _ ws → .<1 + .~(gen_len ws)>.)
```

Basically, we just replaced `fun` by `dfun` and `match` by `match_ls`. The `dfun` wraps the generated parameter in void static information, so `zs.static = None` and `zs.dynamic = .<v_zs>.` for some (dynamically bound) variable. The `cons` oper-

ator is just `::` for partially static lists, and `known` creates `sd` with the specified static information (definitions omitted), so when `f` is entered, we have[1]

```
zs = { dynamic = .<v_zs>.; static = None }
xs == zs (* NB: physical equality *)
ys = { dynamic = .<1::v_zs>.; static = Cons (1, zs) }
```

representing the fact that we have no knowledge of the dynamic value of `zs` while we do know `xs = zs` and `ys = 1::zs`. Most importantly, `ys` shares the `zs` node with `xs`, so that any changes to `zs` are visible from both `xs` and `ys`. When the `match_ls` in `f` introduces the assumption `xs = []` by modifying `xs`, that change also happens on `zs` (because they're physically equal), and this change is visible from `ys`. After introducing the assumption, the data look like

```
zs = { dynamic = .<zs'>.; static = Some Nil }
xs == zs (* NB: physical equality *)
ys = { dynamic = .<1::zs'>.; static = Cons (1, zs) }
```

representing the updated, local knowledge `zs = []` and `xs = []` and `ys = [1]`, as desired. Subsequent `match_ls` on `ys` can avoid generating any dynamic `match` using this static information.

Overall, the generated code is

```
.<fun zs → match zs with [] → 0 + (1 + 0)
                       | x::xs → (* discussed later *)>.
```

Both calls to `length` have been completely optimized away. This would not have happened if the assumption about `xs` didn't propagate to `ys`.

Thus, the techniques in this section suffice to imitate driving, including open-term reduction, introduction of assumptions, and propagation to all copies. We should note that not all open-term reductions are easily simulated this way. For example, in the `f` function above, `(+)` is hard-coded inside brackets, so it's not optimized away, whereas an automated supercompiler might reduce it as well. For this example, if we really need to optimize that addition, we can still do so by making the returned integer partially static. Such a workaround may or may not be so obvious in general; however, experience with more traditional, PE-like uses of MSP suggests that this is not a significant issue.

4 Ensuring Termination

The previous section deliberately ignored a part of `contrived` that involves a termination issue. In this section, we explain how to simulate α-invariant folding to ensure termination. The most obvious way to fill in the expression marked *(*discussed later *)* in the staged code above is to put `gen_len xs` there, following the structure of the original, unstaged code. Alas, this call never finishes. The input `xs` is not completely statically known, so `gen_len` eventually runs out of

[1] Pedantically, the first argument of the `Cons` in `ys.static` should be another `sd`, but we simply write the static representation `1` for the sake of conciseness.

```
(* State monad. *)
type ('a,'st) monad = 'st → ('a * 'st)
(* memoize : 'key
   → ('a code, ('key,'a code) table) monad
   → ('a code → ('b code, ('key,'a code) table) monad)
   → ('b code, ('key,'a code) table) monad *)
let memoize key fcn call =
  bind get (fun table →
  match lookup key table with
  | Some f → call f
  | None → bind get (fun table →
            ret .<let rec f = .~(run_monad fcn (add key .<f>. table))
                  in .~(run_monad (call .<f>.) table)>.))

(* Fix the table type for brevity. *)
type 'a table_monad =
  ('a, ((int, int) psl, (int list → int) code) table) monad
(* gen_contrived : unit → (int list → int) code table_monad *)
let rec gen_contrived () = dfun (fun zs →
  let f xs ys = match_ls xs
                (fun () → gen_len xs +! gen_len ys)
                (fun _ ws → gen_len xs)
  in f zs (cons (known 1 .<1>.) zs))
(* gen_len : (int,int) psl → (int code) table_monad *)
and gen_len ws =
  memoize (freeze ws)
   (dfun (fun ws' → alias ws (forget ws') (fun () →
            match_ls ws
            (fun () → ret .<0>.)
            (fun _ ws → ret .<1>. +! gen_len ws))))
   (fun f → return .<.~f .~(forget ws)>.)
```

Fig. 2: Staged `contrived` function with memoization.

static information to act on. This means the `match_ls` in `gen_len` generates a dynamic `match`, whose cons-branch is generated by creating a fresh `ws`, again with no static information. This is then passed recursively to `gen_len`, which repeats the same process.

This situation is analogous to driving without folding. With `match_ls`, we are forcing the evaluation of branch bodies of statically unresolvable pattern-matches by making deeper and deeper assumptions about the input list, but there is no bound on the depth of this assumption. This leads to non-termination, because unlike the driving process described in Section 2, the code shown here doesn't generate a (recursive) function that can be reused later when an identical `match_ls` is reached. Generating and memoizing those functions is an integral

part of the positive supercompiler's termination heuristic, and we need to simulate this in MSP as well.

Figure 2 shows a terminating generator which memoizes the pattern-match in `gen_len`, keyed with the scrutinee (since that's the only free variable in the `match` statement). Following Swadi et al. [15], we thread the memo table by a state monad; `ret`, `bind`, and `get` are the usual state monad operations, and `match_ls` and `dfun` are updated to work inside the monad. Similarly, `(+!)` generates a dynamic `(+)` inside the monad. Other than that, the only change is the addition of a call to `memoize`, which takes a `key`, a monadic action `fcn` that generates a function, and `call` which maps a dynamic function to some code invoking that function. If `key` is not in the table, `memoize` dynamically binds the function returned by `fcn` and generates a call to it with `call`. The `fcn` is run on a state extended with the mapping `key` $\mapsto$ `.<f>.`, where `f` is the newly generated function. If `memoize` is invoked again with the same key while `fcn` generates the body of `f`, then only `call` is invoked, without generating a new function. Thus, the code in Figure 2 terminates and generates

```
.<fun zs → match zs with [] → 0 + (1 + 0)
                       | w::ws → 1 +
                                 (let rec len ws =
                                    match ws with [] → 0
                                                | _::ws' → 1 + len ws'
                                  in len ws)>.
```

This memoization scheme has several subtleties, two of which are explained here, while the last one is explained in the next section using the more sophisticated KMP example. The first subtlety is that memoization keys must be deep-copied before inserting into the table, because subsequent introduction of assumptions can change their static information. The `freeze` function in Figure 2 performs this deep copy. The second subtlety is that key comparison cannot be simple equality. For example, if `gen_len` is called on the partially static datum

```
xs = { dynamic = .<v_xs>.; static = None }
```

for some dynamic variable `v_xs` bound on the caller's side, then a new entry is created in the memo table with `xs` as the key (assuming it's not already there). However, the second branch of `match_ls` calls `gen_len` on

```
ws = { dynamic = .<v_ws>.; static = None }
```

where `v_ws` is the symbol freshly generated by `match_ls`. If these keys were compared with `(=)`, then the lookup would fail, resulting in non-termination.

This shows that key comparison should ignore differences in names of dynamic variables. However, it should *not* ignore differences in sharing. Although not an issue for `gen_len`, if a function with two arguments `xs` and `ys` introduces assumptions on `xs` and then pattern-match on `ys`, then a memo entry created when `xs` and `ys` are physically equal must not be used at a call site where they are not equal. In general, the keys must be compared under DAG isomorphism – they are equal iff they have the same shape (same number of cons cells with

the same heads, a.k.a. car's, linked together in the same manner), but not the same names on the leaves where static information is `None`.

This keying discipline is not so mysterious if we consider the connection to α-invariant folding in positive supercompilation. A static-dynamic datum with void static information is like a variable in the object term of supercompilation, whereas a static-dynamic datum with, say `Cons(1,xs)` as static information is like an open object term `1::xs` in supercompilation. The function generated during memoization is the lambda-lifting of the `match` that is memoized, and the memo keys are the collection of all partially static data manipulated inside that `match`. Hence, if a `match` statement on a particular source location is executed multiple times, each execution instance is uniquely identified by the key. The lambda-lifted function `f` is reusable precisely when supercompiling a term whose lambda-lifting is α-equivalent to `f`, which is necessary and sufficient for the lookup key to be graph-isomorphic to the key found in the table.

5 Case Study: KMP

In this section, we show that our MSP techniques suffice to pass the "KMP test" for supercompilation [13]. In this test case, we explain the final subtlety in implementing α-invariant folding with memoization, which motivates one final technique which we call delimited aliasing.

Figure 3a shows a function `search` that tests if a pattern string `p` occurs in a subject string `s`.[2] It checks if `p` is a prefix of `s` by character-wise comparison, and upon a mismatch, drops the head of `s` and starts over. If `p`,`s` have lengths m, n, respectively, this takes $O(mn)$ comparisons. The objective is, given a concrete pattern, to generate the efficient KMP algorithm in Figure 3c which performs only $O(m + n)$ comparisons (not counting generation cost).

Specializing `search` to a fixed pattern `"aab"` with PE gives more or less Figure 3b, where the `[]`-cases of `match`es are omitted due to space limitations. The `match`es on the pattern are statically resolved, but the subject is still rewound to the beginning upon a mismatch, resulting in $O(mn)$ comparisons. We can do better. If the third character mismatched, the subject must start with `"aa"`, so we know the first comparison of the next round will return `true`. We can therefore skip that comparison. Eliminating such redundant comparisons gives the KMP algorithm in Figure 3c. Note the failing branch of the comparison in `kmp_b` jumps to `kmp_ab` instead of `kmp_aab`.

This optimization happens by noting static information learned about `os` due to pattern matches and comparisons on `ss`. It's by following the `match ss` and `if s = 'a'` that we learn (or assume) that the subject starts with `"aa"`, and `os` is never inspected; nonetheless, this information should propagate to `os` and be used to skip (or statically perform) redundant comparisons. This is just what positive supercompilation does, as do our MSP techniques. Figure 3d demonstrates a staged version of the matcher. It is fairly straightforward, with

[2] Strings are represented as `char list` rather than `string`, but for brevity we write literals `"like this"` where convenient.

```
let rec search p s = loop p s p s
and loop pp ss op os =
  match pp with
  | [] → true
  | p::pp' →
    match ss with
    | [] → false
    | s::ss' →
      if s = p
      then loop pp' ss' op os
      else next op os
and next op = function
  | [] → false
  | s::ss → loop op ss op ss
(* Mnemonics for variable names:
   p, pp −− Pattern to search for
   s, ss −− Subject to search over
   op −− Original Pattern
   os −− Original String *)
```

(a) Generic version.

```
let rec naive_aab ss = aab ss ss
and aab ss os =
  match ss with
  | x::xs →
      if x = 'a' then ab xs os
      else next os
and ab ss os =
  match ss with
  | x::xs →
      if x = 'a' then b xs os
      else next os
and b ss os =
  match ss with
  | x::xs →
      if x = 'b' then true
      else next os
and next = function
  | _::xs → aab xs xs
```

(b) Naïvely specialized to "aab".

```
let rec kmp_aab = function
  | x::xs →
    if x = 'a' then kmp_ab xs
    else kmp_aab xs
and kmp_ab = function
  | x::xs →
    if x = 'a' then kmp_b xs
    else kmp_aab xs
and kmp_b = function
  | x::xs →
    if x = 'b' then true
    else if x = 'a' then kmp_b xs
         else kmp_ab xs
```

(c) Hand-written KMP for "aab" (split in two columns).

```
let rec loop pp ss op os =
  match pp with
  | p::pp' →
    memoize (freeze (pp,ss,op,os))
      (dfun (fun ss' → alias ss (forget ss')) (fun () →
              match_ls ss (fun () → ret .<false>.)
                          (fun s ss →
                            ifeq s (known p .<p>.)
                            (fun () → loop pp' ss op os)
                            (fun () → next op os)))))
      (fun f → ret .<.~f .~(forget ss)>.)
and next op os () = match_ls os (fun () → ret .<false>.)
                                (fun s ss → loop op ss op ss ())
```

(d) Staged string search with memoization (one column).

Fig. 3: String matcher. Suffixes in specializations indicate remaining pattern.

`match` replaced by `match_ls` and `if s = c` replaced by `ifeq`, a combinator similar to `match_ls` but generating equality tests with constants.

The one aspect in which this example differs significantly from `contrived` is the use of the combinator

```
alias : ('a,'b) sd → 'b code → (unit → ('c,'d) monad) → ('c,'d) monad
```

which is almost the same as `assuming_eq` but updates the dynamic value instead of the static information. For example, if we reach the `memoize` in Figure 3d when

```
pp = "aab"
op = "aab"
ss = { dynamic = .<v_ss>.; static = None }
os = { dynamic = .<v_os>.; static = Some ('a', ss) }
```

then `memoize` calls back the generator of the memoized body (i.e. the part that starts out with `dfun`), and the `dfun` creates a new static-dynamic value

```
ss' = { dynamic = .<v_ss'>.; static = None }
```

Then `alias ss (forget ss')` modifies the dynamic variable associated to `ss` to make it an alias for `ss'`, hence

```
ss = { dynamic = .<v_ss'>.; static = None }
```

All other static-dynamic values remain unchanged. Just like `assuming_eq`, this mutation is undone when the thunk (the last argument to `alias`) returns.

The reason we need this is because, by making `v_ss` an argument to the function generated by `memoize`, we're effectively renaming the dynamic variable `v_ss`. The whole point of generating a function is to have its body process the parameter `v_ss'` instead of `v_ss`, so that this body becomes reusable. However, in the case of KMP, the body must also process `os`, which would still refer to `v_ss` instead of `v_ss'`; mutating the `ss` structure ensures that both `os` and `ss` are updated to point to `v_ss'`.

This scheme once again corresponds to α-invariant folding, where free variables are captured and consistently renamed. Mutating the dynamic variables on leaf nodes of static-dynamic values corresponds to renaming the dynamic variable associated with that value across the board.

It should be noted that to be faithful to the α-invariant folding heuristic, `alias` should only be used on leaf nodes, whose static information is `None`. This ensures maximum retention of static information, because `alias`'ed nodes must have void static information (since the new dynamic variables have no static information). Thus, a combinator would be helpful that traverses static-dynamic data and collects such nodes, generating a function with as many arguments as needed. This will be fairly tricky to type in (Meta)OCaml, since we need to traverse arbitrary data structures while managing a heterogeneous collection of dynamic variables to eliminate duplicates. We leave the pursuit of such a combinator for another occasion.

With these mechanisms in hand, the generator in Figure 3d produces more or less the KMP code in Figure 3c, with two minor differences. Firstly, as posi-

tive supercompilation tracks equalities but not disequalities, we have redundant comparisons of the form

```
if x = 'a' then ...
else if x = 'a' then ... else ...
```

This can be eliminated by maintaining richer static information. For example, a dynamic value can be tagged with the set of values it can have, rather than a single value. The other difference is that the generated code nests `let rec`s like

```
let rec f1 =
  let rec f2 = bar
  in baz
in foo
```

instead of having a single, flat `let rec`. Hence, only functions generated in the direct ancestors of a `memoize` call can be reused, which is both a good safety precaution and a limitation. Reusing functions from a different conditional branch runs the risk of invoking code that relies on assumptions valid only in that branch, but if used properly it can reduce generated code size. Current MetaOCaml provides no way to generate `let rec` with a variable number of bindings, but a new primitive allowing that is expected in a future release.[3] It would be interesting to see if they enable notable improvements.

6 Related Work

Supercompilation was devised by Turchin for Refal [19] and later adapted to more standard functional languages by Glück and Klimov [3]. Sørensen et al. placed this on the same theoretical footing as PE, deforestation, and generalized partial computation (GPC), and showed that supercompilation subsumes PE and deforestation [13]. We have drawn heavily from this work: [13] effectively identified all the key ingredients for supercompilation, in terms that are transferable to MSP. Supercompilation has been extended by distillation [4], but it remains unclear what the differences are, in terms that can be mapped to MSP.

GPC [2,18] is an extension of PE that uses a theorem prover to manage static information. While the use of a theorem prover makes it harder to predict how it performs on any given task, we remark that the delimited assumption technique can be used to simulate GPC as well, by simply taking the static information to be variables in the theorem prover. Compared to GPC, however, our techniques perform the very stylized information propagation of supercompilation, which behaves more predictably than if the bookkeeping is delegated to a black-box solver. In this way, our techniques might be useful to lighten the load on the prover.

MSP was originally a notation for PE [10] but was later developed into a programming language feature by Taha and Sheard [17]. Its main advantages are the existence of a well-behaved metatheory [5] and type systems that make strong

[3] Private communication with the maintainer.

guarantees about generated code [6, 16, 20]. MetaOCaml statically prevents the construction of ill-formed or ill-typed code values, with the one exception that effects can cause scope extrusion, where a dynamic variable is floated out of its scope. Much effort has been expended on catching this problem early, resulting in static type systems [6,20] and dynamic checks [7], the latter of which MetaOCaml already implements. The present paper adds to the motivation for these efforts by offering a new, important use for effectful MSP.

Partially static data types were known in PE circles starting perhaps with Mogensen [9], but Sheard and Diatchki [11] seem to be the first to use it as a staging technique. However, they duplicated constructors instead of pairing dynamic values with optional static information, which made their code generally more verbose than ours. The pairing technique itself appears in earlier PE works, for example [14]. The observation that mutating components of these pairs can simulate supercompilation appears to be new.

7 Conclusion

We showed that MSP can achieve a good deal of the effects of positive supercompilation. The central idea is to update the static portion of partially static data structures upon entering a dynamic conditional, and to do this with mutation. This arrangement ensures that the assumption is propagated to all copies of the data, allowing smart handling of nonlinear code. As an auxiliary technique, a fairly nonstandard memoization scheme may be required to ensure termination, namely comparing partially static data with graph isomorphism. Taken together, these techniques can specialize a naïve string matcher to a KMP matcher.

The techniques in this paper should be thought of as low-level groundwork for realizing supercompilation by staging. It is fairly technical and we can't expect most MetaOCaml programmers to be apply this easily, without making mistakes. A well-designed combinator library should be able to alleviate this problem. An important goal for such a library is to offer a richer `memoize` combinator that collects leaf nodes from its key and generates a function with as many parameters as are needed, performing delimited aliasing as well. This would make the techniques much more straightforward to understand.

Finally, this paper's purpose is to demonstrate techniques that are useful in expressing supercompilation-like optimizations in MSP, and not to lay down a formal analysis. We did not attempt to define precisely what class of programs can be supercompiled, but as mentioned earlier, not all driving trees are naturally expressed with MSP. It would be interesting to see what kinds of driving trees are beyond MSP in its current form (if any).

Acknowledgment. We thank Oleg Kiselyov for his insightful comments and encouragement to publish this work.

References

1. Dybvig, R.K.: Writing hygienic macros in scheme with syntax-case. Tech. Rep. TR356, Indiana University Computer Science Department (1992)
2. Futamura, Y.: Program evaluation and generalized partial computation. In: FGCS. pp. 685–692 (1988)
3. Glück, R., Klimov, A.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Static Analysis, Lecture Notes in Computer Science, vol. 724, pp. 112–123. Springer Berlin Heidelberg (1993)
4. Hamilton, G.W.: Distillation: Extracting the essence of programs. In: PEPM. pp. 61–70. ACM, New York, NY, USA (2007)
5. Inoue, J., Taha, W.: Reasoning about multi-stage programs. In: ESOP. pp. 357–376 (2012)
6. Kameyama, Y., Kiselyov, O., Shan, C.c.: Combinators for impure yet hygienic code generation. In: PEPM. pp. 3–14 (2014)
7. Kiselyov, O.: The design and implementation of BER MetaOCaml: System description. In: FLOPS, to appear (2014)
8. Knuth, D.E., Morris, J., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal of Computing 6(2), 323–350 (1977)
9. Mogensen, T.Æ.: Efficient self-interpretations in lambda calculus. Journal of Functional Programming 2(3), 345–363 (1992)
10. Nielson, F., Nielson, H.R.: Two-level functional languages. Cambridge University Press (1992)
11. Sheard, T., Diatchki, I.S.: Staging algebraic datatypes. Unpublished manuscript, `http://web.cecs.pdx.edu/~sheard/papers/stagedData.ps`
12. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School. pp. 246–270 (1999)
13. Sørensen, M.H., Glück, R., Jones, N.D.: Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In: ESOP. pp. 485–500 (1994)
14. Sperber, M.: Self-applicable online partial evaluation. In: Partial Evaluation, LNCS, vol. 1110, pp. 465–480. Springer (1996)
15. Swadi, K., Taha, W., Kiselyov, O., Pašalić, E.: A monadic approach for avoiding code duplication when staging memoized functions. In: PEPM. pp. 160–169. ACM (2006)
16. Taha, W., Nielsen, M.F.: Environment classifiers. In: POPL. pp. 26–37. ACM (2003)
17. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: PEPM. pp. 203–217. ACM (1997)
18. Takano, A.: Generalized partial computation using disunification to solve constraints. In: CTRS. pp. 424–428 (1993)
19. Turchin, V.: A supercompiler system based on the language Refal. SIGPLAN Notices 14(2), 46–54 (1979)
20. Westbrook, E., Ricken, M., Inoue, J., Yao, Y., Abdelatif, T., Taha, W.: Mint: Java multi-stage programming using weak separability. In: PLDI (2010)

Towards Understanding Superlinear Speedup by Distillation

Neil D. Jones

Computer Science Department
University of Copenhagen
2100 Copenhagen, Denmark
neil@diku.dk

G.W. Hamilton

School of Computing
Dublin City University
Dublin 9, Ireland
hamilton@computing.dcu.ie

Abstract. Distillation is a fully automatic program transformation that can yield superlinear program speedups. Bisimulation is a key to the proof that distillation is correct, i.e., preserves semantics. Bisimulation normally requires explicit definition of equivalent states. However distillation can produce complexity reductions and thus fewer residual states. This often makes 1-1 relations between program states of original and transformed programs hard or impossible to see. The correctness proof of distillation, since based on observational equivalence, is insensitive to program running times, and does not help to explain how superlinear speedups can occur. This paper's approach to better understanding cause-and-effect in distillation is to simplify distillation as much as possible, while maintaining its capacity for superlinear speedups. We show *how distillation can give superlinear speedups* on some "old chestnut" programs well-known from the early program transformation literature: naive reverse, factorial sum, Fibonacci, and palindrome detection. We describe current work on such questions, partly theoretical and partly computer experiments. Furthermore, we show using complexity-theoretic tools that a sizable class of exponential-time programs can be converted into second-order polynomial-time equivalents. The idea is to trade time for space, in effect replacing `cons` or a Turing machine tape by first-order functions as arguments in a `cons`-free program. Finally, we conjecture that distillation can realise these superlinear speedup transformations in general.

1 Introduction

Distillation, supercompilation, and partial evaluation are automatic program transformations (see [11–13, 21–23]). The main goal of all three is to transform a program into an improved program. Partial evaluation has been fairly well automated [13]. A breakthrough occurred when the *Futamura projections* (Futamura, Ershov [7, 8]) were realised in practice: generating a compiler from an interpreter by self-applying a partial evaluator (see [13] for details and history). Furthermore, optimal specialisation has been achieved: partial evaluation can remove all interpretation overhead when specialising an interpreter to its program input.

In some respects supercompilation, deforestation and distillation (Turchin, Sørensen, Wadler, Hamilton [10–13,21,23,24]) can make deeper transformations on program control structure. A well-known example is that deforestation can transform a multipass program into a single pass algorithm [23,24], a feat beyond the reach of current partial evaluators.

1.1 Goal: extend automatic superlinear program speedup

Program optimisations by hand (Burstall-Darlington and many others [1, 3]) sometimes yield superlinear program speedups. Transformation can make substantial improvements, for instance changing a program running in time $O(n^2)$ or even $O(2^n)$ into one running in time $O(n)$. Familiar examples include naive programs for list reversal, sum of factorials, and the Fibonacci function. A goal for many years now has been how to obtain such effects *by well-automated methods.*

Classical compiler optimisations are a model of automation, though the program speedups they give are limited. Many have been proven correct using bisimulation, e.g., [17] by Lacey et al. This has led to some practical automation of compiler correctness proofs, e.g., [18] by Lerner et al, and successors.

However it has been proven (see [13,21]) that partial evaluation, deforestation and supercompilation (as well as most classical compiler optimisations) are all limited to *at most program speedups by linear constant factors.* One reason for such limited optimisation speedups is that the bisimulations of [17] all involve *one-to-one relations* between the control points of the original program and the compiler-optimised program.

In contrast, distillation [10, 11]) can yield superlinear asymptotic speedups: this refinement of supercompilation can sometimes transform a program into a semantically equivalent but asymptotically faster equivalent.

1.2 Bisimulation and program transformation.

Correctness of transformation can be proven using bisimulation [11, 12, 17] to relate computations by the original and the transformed programs. A question:

> How can a program running in time $O(n^2)$ (or even time $O(2^n)$) be bisimilar to a program running in time $O(n)$?

This puzzling question was the starting point of this work. It was clear at once that *1-1 relations between program control points would not suffice to explain the phenomenon.* A challenge to overcome: the system structure and techniques used in distillation as in [10–12] are complex, and hard to reason about globally.

This paper's approach to better understanding cause-and-effect in distillation is to *simplify distillation as much as possible*, while maintaining its capacity for superlinear speedups. We will describe current work on such questions, partly theoretical and partly computer experiments.

2 A language, observational equivalence, and labeled transition systems

Our approach is to simplify the general distillation techniques of [10–12], so its essence can be seen in a more limited context, to see what is happening abstractly. A clearer understanding of cause-and-effect could show how automatically to achieve superlinear speedup on a wider range of programs.

A longer-term goal is to apply distillation techniques to intermediate program representations in a compiler, where programs are call-by-value and imperative, i.e., tail-recursive. Related: Debois applies partial evaluation to realise some optimisations of intermediate program representations in a compiler [6]; however, superlinear speedup is beyond the current state of the art.

2.1 Source language syntax

Data: let Σ be an uninterpreted signature for constructors, and let T_Σ be the set of all well-formed trees over Σ, finite or infinite. Our examples use as constructors 0-ary 0, unary 1+ (successor), and binary constructors $+, *, ::$. The net effect of a program is to compute a (mathematical, partial) function $f : (T_\Sigma)^n \rightharpoonup T_\Sigma$.

Programs are first-order, built from variables x, constructors c, function calls, and **case**. Calls and constructor applications must have all their arguments, i.e., full arities. Semantics $[\![prog]\!] : (T_\Sigma)^n \rightharpoonup T_\Sigma$ is call-by-value, omitted for brevity and because of familiarity.

$$
\begin{array}{lll}
prog & ::= e \textbf{ where } \Delta & \\
\Delta & ::= f_1\ x_1 \ldots x_n = e_1 \quad \ldots \quad f_m\ x_1 \ldots x_p = e_m & \text{Function definitions} \\
e & ::= x \mid c\ e_1 \ldots e_k \mid call \mid case & \text{Expression} \\
call & ::= f\ e_1 \ldots e_n & \text{Function call} \\
case & ::= \textbf{case } e \textbf{ of } p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k & \text{Case expression} \\
p & ::= c\ x_1 \ldots x_k & \text{Case pattern}
\end{array}
$$

Free variables are allowed only in the e part of program e **where** Δ. All other variables must be bound, either by function parameters or in case patterns.

Definition 1. *Denote by* $time_p(x) \in \mathbb{N} \cup \{\infty\}$ *the running time of program* p *on input* x*, e.g., the number of steps used in computing* $[\![p]\!](x)$.

Goal: automatically transform program p into program p' such that $[\![p]\!] = [\![p']\!]$, but $time_{p'} < time_p$ asymptotically, i.e., in the limit as input size grows.

2.2 Observational equivalence and labeled transition systems

Distillation transforms a program p_1 into an observationally equivalent program p_2. (Two central references: Milner and Gordon [9, 19].) Observational equivalence implies semantic equivalence, i.e., $p_1 \simeq p_2$ implies $[\![p_1]\!] = [\![p_2]\!]$.

For definitions of context $C[]$ and evaluation $\Downarrow$ appropriate to call-by-value:

Definition 2 (Observational Equivalence). Programs p_1, p_2 are *observationally equivalent*, written $p_1 \simeq p_2$, if and only if they have the same termination behaviour in all closing contexts, i.e., $p_1 \simeq p_2$ iff $\forall C[] \,.\, C[p_1] \Downarrow$ iff $C[p_2] \Downarrow$.

A limitation of observational equivalence Unfortunately (from this paper's perspective), observational equivalence $p \simeq p'$ tells us *nothing whatsoever* about the comparative running times of the programs involved. In each instance of our "old chestnut" programs, the original program is observationally equivalent to its optimised version. Our goal: to clarify the relation between program running times before and after distillation.

Definition 3 (Labeled transition systems). *A labeled transition system (LTS for short) is a tuple* $t = (\mathcal{S}, s_0, \rightarrow, Act)$ *where* $\mathcal{S}$ *is a set of* states. $s_0 \in \mathcal{S}$ *is the* root state, $\mathbf{0}$ *is the end-of-action state, and Act is a set of actions* α. *The* transition relation *is* $\rightarrow \subseteq \mathcal{S} \times Act \times \mathcal{S}$. Notation: *as usual we write a transition* (s, α, s') *in* $\rightarrow$ *as* $s \xrightarrow{\alpha} s'$.

Definition 4 (Simulation). Binary relation $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ is a *simulation* of LTS $t_1 = (\mathcal{S}_1, s_0^1, \rightarrow_1, Act)$ by LTS $t_2 = (\mathcal{S}_2, s_0^2, \rightarrow_2, Act)$ if $(s_0^1, s_0^2) \in \mathcal{R}$, and for every pair $(s_1, s_2) \in \mathcal{R}$ and $\alpha \in Act, s_1' \in \mathcal{S}_1$:

$$\text{if } s_1 \xrightarrow{\alpha} s_1' \text{ then } \exists s_2' \in \mathcal{S}_2 \,.\, s_2 \xrightarrow{\alpha} s_2' \wedge (s_1', s_2') \in \mathcal{R}$$

Note: t_1 and t_2 must have the same action sets.

Definition 5 (Bisimulation). A *bisimulation* $\sim$ is a binary relation $\mathcal{R}$ such that both $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are simulations.

Using an LTS as a program's abstract syntax. Represent a variable x by a transition $s \xrightarrow{x} \mathbf{0}$; represent $c\ e_1 \ldots e_k$ where c is a constructor by transitions $s \xrightarrow{c} \mathbf{0}, s \xrightarrow{\#1} s_1, \ldots, s \xrightarrow{\#k} s_k$, where s_i is the root of the LTS representation of expression e_i; represent **case** e_0 **of** $p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k$ by transitions $s \xrightarrow{\mathbf{case}} s_0, s \xrightarrow{p_1} s_1, \ldots, s \xrightarrow{p_k} s_k$; and represent a function call $f\ e_1 \ldots e_n$ by transitions $s \xrightarrow{\mathbf{call}} s_0, s \xrightarrow{x_1} s_1, \ldots, s \xrightarrow{x_n} s_n$ where Δ contains function definition $f\ x_1 \ldots x_n = e_0$.

2.3 Example: "naive reverse" program representation as an LTS

```
nr input where

nr xs = case xs of
            nil       => nil
          | (:: y ys) => (ap (nr ys) (:: y nil))

ap us vs = case us of
               nil        => vs
             | (:: u us1) => (:: u (ap us1 vs))
```

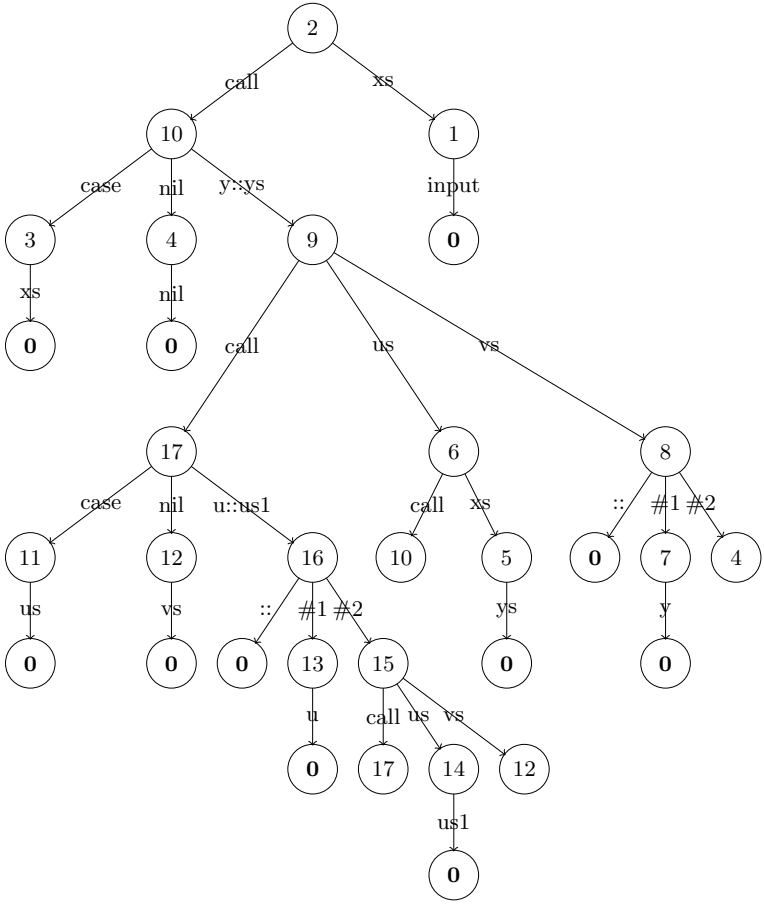


Fig. 1. Labelled Transition System for "naive reverse" program

The LTS representation of the naive reverse program is diagrammed in Fig. 1. This may be easier to follow than an unstructured set of transitions such as

$$\{\ 2 \xrightarrow{\textbf{call}} 10,\ 2 \xrightarrow{\texttt{xs}} 1,\ 1 \xrightarrow{\texttt{input}} \mathbf{0},\ 10 \xrightarrow{\textbf{case}} 3,\ 10 \xrightarrow{\texttt{nil}} 4,\ 10 \xrightarrow{\texttt{::(y,ys)}} 9,\ 3 \xrightarrow{\texttt{xs}} \mathbf{0},\ 4 \xrightarrow{\texttt{nil}} \mathbf{0},\ \ldots\}$$

Short form of the LTS for naive reverse (root state 2, `nr` *code start* `10`*, and* `ap` *code start* `17`*).* We abbreviate the LTS by omitting end-of-action state **0** and variable transitions to **0**, and bundling together transitions from a single state.

```
(2  -> (call 10 (input)))       ;  root = 2: call nr(input)
(10 -> (case xs ((nil).4) ((:: y ys).9))) ; start "nr"
(4  -> (nil))
```

```
(9  -> (call 17 (6 8))              ; call ap(nr(ys),...)
(6  -> (call 10 (ys)))              ; call nr(ys)
(8  -> (:: y 4))
(17 -> (case us ((nil).vs) ((:: u us1).16)))) ; start "ap"
(16 -> (:: u 15)
(15 -> (call 17 (us1 vs))           ; call ap(ws,vs)
```

An example of optimisation: The program above runs in time $O(n^2)$. It can, as is well known, be improved to run in time $O(n)$. Distillation does this automatically, yielding the following LTS with root state 3 and `rev` code start `10`. Effect: the nested loop in `nr` has been replaced by an accumulating parameter.

```
      ;  Reverse with an accumulating parameter
(3  -> (call 10 (us 2)))
(2  -> (nil))
(10 -> (case xs ((nil) . acc) ((:: x xs1) . 9)))
(9  -> (call 10 (xs1 8)))
(8  -> (:: x acc))
```

The distilled version in source language format.

```
rev us nil where

rev xs acc = case xs of
                  nil        => acc
                | (:: x xs1) => rev xs1 (:: x acc)
```

Are these programs bisimilar? There is no obvious bisimilarity relation between runtime states of `nr` and `rev`, e.g., because of different loop structures and numbers of variables. In the next section we will see that the *result of driving* a distilled program is always bisimilar to the *result of driving* the original program.

3 Distillation: a simplified version

We now describe (parts of) a cut-down version of distillation. Following the pattern of Turchin, Sørensen and Hamilton, the first step is driving.

3.1 Driving

Distillation and supercompilation of program $p = e$ **where** Δ both begin with an operation called *driving*. The result is an LTS $\mathcal{D}[\![p]\!]$, usually infinite, *with no function calls* and *with no free variables* other than those of p.

If p is closed, then driving will evaluate it completely, yielding as result an LTS for the value $[\![p]\!]$. Furthermore, given an LTS for a program p *with* free variables, the driver will:

- compute as much as can be seen to be computable;
- expand *all* function calls and
- yield as output a call-free LTS $\mathcal{D}[\![p]\!]$ equivalent to program p. (The output may be infinite if the input program has loops.)

$\mathcal{D}[\![p]\!]$ will consist entirely of constructors, variables and **case** expressions whose tests could not be resolved at driving time. This is a (usually infinite) LTS to compute the function $[\![p]\!]$ (of values of p's free variables). Another perspective: $\mathcal{D}[\![p]\!]$ is essentially a "glorified decision tree" to compute $[\![p]\!]$ without calls. Input is tested and decomposed by **case**, and output is built by constructors.

Although $\mathcal{D}[\![p]\!]$ may be infinite it is executable, given initial values of any free variables. This can be realised in a lazy language, where only a finite portion of the LTS is looked at in any terminating run.

Correctness of distillation: Theorem 3.10 in [11] shows that for any p, p',

$$\mathcal{D}[\![p]\!] \sim \mathcal{D}[\![p']\!] \textit{ implies } p \simeq p'$$

Bottom line: if two programs p, p' have bisimilar driven versions $\mathcal{D}[\![p]\!]$ and $\mathcal{D}[\![p']\!]$, then the programs are observationally equivalent.

3.2 A driver for the call-by-value language

The driving algorithm transforms a program into a call-free output LTS (possibly infinite). It is essentially an extended semantics: an expression evaluator that also allows free variables in the input (transitions to **0** are generated in the output LTS for these variables); and **case** edges that are applied to a non-constructor value (for each, a residual output LTS **case** transition is generated).

Relations to the drivers of [10–12]: We do not use silent transitions at all, and so do not need weak bisimulation. Our LTS states have no internal structure, i.e., they are not expressions as in [10–12], and have no syntactic information about the program from which they were generated, beyond function parameters and case pattern variables. (Embedding, generalisation, well-quasi-orders etc. are not discussed here, as this paper's points can be made without them.)

Another difference: the following constructs its output LTS "one state at a time": it explicitly allocates new states for constructors and for **case** expressions with unresolvable tests.[1]

One effect is an "instrumentation". For instance if p is closed, then the driven output LTS $\mathcal{D}[\![p]\!]$ will have one state for every constructor operation performed while computing $[\![p]\!]$, so $\mathcal{D}[\![\_]\!]$ yields *some intensional information* about its program argument's running time (in spite of Theorem 3.10 0f [11]).

Our language is call-by-value, so environments map variables into states, rather than into expressions as in [10, 11]. Types used in the driver:

[1] To avoid non-termination of the program transformer itself, we assume the input does not contain nonproductive loops such as $f\ 0$ **where** $f\ x = f\ x$.

$\mathcal{D} : Expression \to LTS$
$\mathcal{D}' : Expression \to LTS \to Environment \to FcnEnv \to LTS$
$\theta \in Environment = Variable \rightharpoonup State$
$\Delta \in FcnEnv = FunctionName \rightharpoonup Variable^* \rightharpoonup Expression$

Variable t ranges over LTS's, and s ranges over states. For brevity, function environment argument Δ in the definition of $\mathcal{D}'$ is elided since it is never changed.

1. $\mathcal{D}[\![e \textbf{ where } \Delta]\!] = \mathcal{D}'[\![e]\!]\ \varnothing\ \{\}\ \Delta$
2. $\mathcal{D}'[\![x]\!]\ t\ \theta\ = \begin{cases} t \text{ with root } \theta x & \textbf{if } x \in dom(\theta) \textbf{ else} \\ t \cup \{s \xrightarrow{x} \mathbf{0}\} & \textbf{where } s \text{ is a new root state} \end{cases}$
3. $\mathcal{D}'[\![c\ e_1 \ldots e_k]\!]\ t_0\ \theta\ = \textbf{let } t_1 = \mathcal{D}'[\![e_1]\!]\ t_0\ \theta, \ldots, t_k = \mathcal{D}'[\![e_k]\!]\ t_{k-1}\ \theta \textbf{ in}$
   $t_k \cup \{s \xrightarrow{c} \mathbf{0}, s \xrightarrow{\#1} root(t_1), \ldots, s \xrightarrow{\#k} root(t_k)\} \textbf{ where } s \text{ is a new root state}$
4. $\mathcal{D}'[\![f\ e_1 \ldots e_n]\!]\ t_0\ \theta\ = \textbf{let } t_1 = \mathcal{D}'[\![e_1]\!]\ t_0\ \theta, \ldots, t_k = \mathcal{D}'[\![e_k]\!]\ t_{k-1}\ \theta \textbf{ in}$
   $\mathcal{D}'[\![e^f]\!]\ t_n\ \{x_1 \mapsto root(t_1), \ldots, x_n \mapsto root(t_n)\}\ \textbf{where } \Delta\ f\ x_1 \ldots x_n = e^f$
5. $\mathcal{D}'[\![\textbf{case } e_0 \textbf{ of } p_1 \Rightarrow e_1 | \ldots | p_n \Rightarrow e_n]\!]\ t\ \theta\ = \textbf{let } t_0 = \mathcal{D}'[\![e_0]\!]\ t\ \theta\ \textbf{ in}$
   if $t_0 \ni s_0 \xrightarrow{c} \mathbf{0}, s_0 \xrightarrow{\#1} s_1, \ldots, s_0 \xrightarrow{\#k} s_k$ **and** $p_i = c\ x_1 \ldots x_k$
   then
   $\mathcal{D}'[\![e_i]\!]\ t_0\ (\theta \cup \{x_1 \mapsto s_1, \ldots, x_k \mapsto s_k\})$
   else
   let $t_1 = \mathcal{D}'[\![e_1]\!]\ t_0\ \theta, \ldots, t_n = \mathcal{D}'[\![e_n]\!]\ t_{n-1}\ \theta$ **in**
   $t_n \cup \{s \xrightarrow{\textbf{case}} root(t_0), s \xrightarrow{p_1} root(t_1), \ldots, s \xrightarrow{p_n} root(t_n)\}$
   where s is a new root state

3.3 Distillation's sequence of transformations

As presented in [11, 12], further analyses and transformations (homeomorphic embedding, generalisation, folding, etc.) on an infinite $\mathcal{D}[\![p]\!]$ will yield a finite transformed program p'. Furthermore, these transformations preserve the property of bisimilarity with $\mathcal{D}[\![p]\!]$.

The following may help visualise the various stages involved in distillation:

p	$\longrightarrow$	LTS^{in}	$\longrightarrow$	LTS^{driven}	$\longrightarrow$	LTS^{out}	$\longrightarrow$	p'
source program	**[parse]**	(finite, with calls)	**[drive]**	(infinite, no calls)	**[distill]**	(finite, with calls)	**[unparse]**	transformed program

Function $\mathcal{D}$ is **[drive]** $\circ$ **[parse]**.

4 Some speedup principles and examples

4.1 On sources of speedup by distillation

Speedups can be obtained for all our well-known "old chestnut" programs p as follows (where p_s is p applied to known values s of its free variables):

1. Drive: construct $\text{LTS}^{driven} = \mathcal{D}[\![p_s]\!]$ from p_s. (This is finite if p_s terminates.)
2. Remove "dead code" from LTS^{driven}: these are any states that are unreachable from its root.
3. Merge any bisimilar states in LTS^{driven}.

Step 2 must be done after constructing LTS^{driven}. Step 3 can be done either after or during driving: elide adding a new state and transitions $s \xrightarrow{a_1} s_1, \ldots, s \xrightarrow{a_n} s_n$ to LTS^{driven} if an already-existing LTS state has the same transitions.

Two points: first, in traditional compiler construction, dead code elimination is very familiar; whereas merging bisimilar states is a form of code folding not often seen (exception: the "rewinding" by Debois [6]). Distillation accomplishes the effect of both optimisations, and in some cases more sophisticated transformations.

Second, the distiller obtains superlinear speedup for all three programs by introducing *accumulating parameters*. In some cases, e.g., Fibonacci, the speedup is comparable to that of "tupling" of Chin et. al. [4,5]; but distillation does not introduce new constructors.

4.2 Overview of the "old chestnut" examples

Our goal is to relate the efficiencies of a program p and its distilled version p'. The transformation sequence as in Section 3.3 involves the possibly infinite object $\mathcal{D}[\![p]\!] = \text{LTS}^{driven}$.

The following experimental results get around this problem by computing $\mathcal{D}[\![p_s]\!]$ for *fixed* input values s. The idea is to drive a version p_s of p applied to known values s of its free variables. Assuming that p_s terminates, this will yield a finite LTS whose structure can be examined.

Let n be the input size (e.g., a list length or number value). Then

1. The naive reverse algorithm *nrev* runs in quadratic time, while its distilled version runs in linear time. Nonetheless, their driven versions are (strongly) bisimilar, and so observationally equivalent.
   Explanation of speedup: $\mathcal{D}[\![nrev_{(a_1 a_2 \ldots a_n)}]\!]$ has $O(n^2)$ states, including states for the computation of the reverse of every suffix of $(a_1 a_2 \ldots a_n)$. Among these, at the end of execution only $O(n)$ states are live, for the reverse of the full list $(a_1 a_2 \ldots a_n)$.
2. The naive program to compute Factorial sum ($sumfac(n) = 0! + 1! + \ldots n!$) has running time $O(n^2)$ and allocates $O(n^2)$ heap cells, due to repeated recomputation of $0!, 1!, 2!, \ldots$; but the distilled version is linear-time. The two are (again) observationally equivalent since their driven versions are bisimilar. The driven naive Factorial sum LTS has $O(n^2)$ states, but among these, only $O(n)$ are live at the end of execution.
   This example is interesting because both source and transformed programs are purely *tail recursive*, and so typical of compiler intermediate code.
3. A more extreme example: the obvious program fib for the Fibonacci function takes exponential time and will fill up the heap with exponentially many

memory cells. On the other hand, the distilled version of Fibonacci uses an accumulator and runs in linear time (counting $+, *$ as constant-time operations). Even so, the two LTS's are bisimilar.

In contrast to the examples above, the driven program $\mathcal{D}[\![fib_n]\!]$ has $O(1.7^n)$ states, all of which are live. Here speedup source number 3 (Section 4.1) comes into play: only $O(n)$ states are bisimulation-nonequivalent.

The experiments were carried out in SCHEME. The first step was parsing: to transform the input program from the form of Section 2.1 into an LTS, which for clarity we will call LTS^{in}. The driver as implemented realises the one of Section 3.2 (except that it it works on LTS^{in} rather than program p). LTS^{out} is the name of the distiller's output.

5 Can distillation save time by using space?

5.1 Palindrome: an experiment with an unexpected outcome

Long-standing open questions in complexity theory concern the extent to which computation time can be traded off against computation space. Consider the set

$$Pal = \{a_1 a_2 \ldots a_n \mid 1 \leqslant i \leqslant n \Rightarrow a_i = a_{n+1-i} \in \{0, 1\}\}$$

This set in LOGSPACE is decidable by a two-loop `cons`-free program that runs in time $O(n^2)$.

On the other hand, it can also be decided *in linear time* by a simple program *with* `cons`. The idea is first to compute the reverse of the input $xs = a_1 a_2 \ldots a_n$ by using an accumulating parameter; and then to compare xs to its reverse. Both steps can be done in linear time.

Here, using extra storage space (`cons`) led to reduced computation time.

A natural conjecture was that *any* `cons`*-free program deciding membership in Pal must run in superlinear time.* The reasoning was that one would not expect distillation to transform a `cons`-free program into one containing `cons`, as this would involve inventing a constructor not present in the input program. To test this conjecture, we ran an existing distiller on the `cons`-free program palindrome-decider.

The result was unexpected, and disproved our conjecture: distillation yielded a *linear-time* but second-order palindrome recogniser(!)

In effect, the distillation output realises `cons` by means of *second-order functions.* Thus, while it does not create any new `cons`'s its output program, it achieves a similar effect through the use of lambdas. The output is as follows[2]:

[2] Automatically produced but postprocessed by hand to increase readability.

```
p xs xs (λzs.True) where

p xs ys q = case xs of
              Nil       => q ys
            | (:: u us) => p us ys (r q u)

r t u = λws.case ws of
              Nil       => True
            | (:: v vs) => case u of
                             0 => (case v of
                                     0 => t vs
                                   | 1 => False)
                           |  1 => (case v of
                                     0 => False
                                   | 1 => t vs)
```

Furthermore, this `cons`-free second-order program is tail recursive in the sense of Definition 6.13 from [15][3]:

Definition 6. `Cons`*-free program* `p` *is* higher-order tail recursive *if there is a partial order* $\geqslant$ *on function names such that any application* $\mathtt{f}\,\mathtt{x}_1 \ldots \mathtt{x}_m = \ldots \mathtt{e}_1\ \mathtt{e}_2 \ldots$ *such that* $\mathtt{e}_1$ *can evaluate to a closure* $\langle \mathtt{g}, v_1 \ldots v_{arity(\mathtt{g})-1} \rangle$ *satisfies either: (a)* $\mathtt{f} > \mathtt{g}$*, or (b)* $\mathtt{f} \equiv \mathtt{g}$ *and the call* $(\mathtt{e}_1\ \mathtt{e}_2)$ *is in tail position.*

The partial order `p`$>$`r` suffices for the palindrome program.

5.2 A theorem and another conjecture

How general is it that distillation sometimes achieves asymptotic speedups? Are the speedups observed in Palindrome, Naive reverse, Factorial Sum and Fibonacci function accidental? Is there a wide class of programs that the distiller can speed up significantly, e.g., from exponential time to polynomial time?

A lead: Jones [15] studies the computational complexity of `cons`-free programs. Two results from [15] about `cons`-free programs of type `[Bool] -> Bool` in our language: Given a set L of finite bit strings:

1. $L \in$ LOGSPACE iff L is decidable by a first-order `cons`-free program that is tail-recursive
2. $L \in$ PTIME iff L is decidable by a first-order `cons`-free program (not necessarily tail-recursive)

Beauty flaw: The result concerning PTIME, while elegant in form, is tantalising because the very `cons`-free programs that decide exactly problems in PTIME, in general *run in exponential time.* (This was pointed out in [15].)

This problem's source is repeated recomputation: subproblems are solved again and again. A tiny example with exponential running time:

[3] The definition is semantic, referring to all program executions, and so undecidable in general. Abstract interpretation can, however, safely approximate it.

```
f x = if x = [] then True else
      if f(tl x) then f(tl x) else False
```

This can be trivially optimised to linear time, but more subtle examples exist. More generally, Figure 5 in [15] builds from any PTIME Turing machine Z a first-order `cons`-free program `p` that simulates Z. In general `p` solves many subproblems repeatedly; these are not easily optimised away as in the tiny example.

The reason, intuitively: absence of `cons` means that earlier-computed results must be recomputed when needed, as they cannot be retrieved from a store.

The "trick" the distiller used in the Palindrome example was to speed up the given `cons`-free program (first-order, quadratic-time) by adding a function as an argument. The resulting second-order Palindrome program ran in linear time.

We now generalise, showing that *for any* first-order `cons`-free program, even one running in exponential time, there is a polynomial-time equivalent.[4]

Theorem 1. *L is decidable by a first-order* `cons`*-free program iff L is decidable by a second-order* `cons`*-free program* that runs in polynomial time.

Corollary 1. *$L \in$ PTIME iff L is decidable by a second-order* `cons`*-free program that runs in polynomial time.*

Some comments before sketching the proof. First, the condition "that runs in polynomial time," while wordy, is necessary since (as shown in [15]), unrestricted second-order cons-free programs decide exactly the class EXPTIME, a proper superset of PTIME. In fact, second-order cons-free programs can run in double exponential time (another variation on the "beauty flaw" remarks above).

Second, the Corollary is analogous to the standard definition: $L \in$ PTIME iff it is decidable by a Turing machine that runs in polynomial time. The punch line is that no tape or other form of explicit storage is needed; it is enough to allow functions as arguments.

Proof (sketch "if"). Suppose L is decidable by a second-order `cons`-free program that runs in polynomial time. All that is needed is to ensure that L can also be decided by a polynomial-time Turing machine. This follows by the known time behavior of call-by-value implementations of functional programs, e.g., as done by traditional compilers that represent functional values by closures.

Proof (sketch "only if"). First, suppose first-order `cons`-free program `p` decides L. Consider the "cache-based algorithm" to compute [[p]] `as` shown in Figure 8 of [15]. While this runs in polynomial time by *Theorem 7.16*, it is not `cons`-free since it uses storage for the cache.

Second, Figure 8 can be reprogrammed, to replace the cache argument by a second-order function. Rather than give a general construction, we just illustrate the idea for the Fibonacci function, and leave the reader to formulate the general construction. The standard definition of Fibonacci:

[4] Can this be strengthened to linear time? No, since $time(O(n)) \subsetneq time(O(n^2))$.

```
f n = if n <= 1 then 1 else f(n-1) + f(n-2)
```

The "cache" of Figure 8 in [15] is a table. For each function f in `p`, it contains all the arguments and the results $f(arguments)$ that have been computed so far. For a cons-free first-order program and an input of length n there can only be polynomially many different arguments.

Of course the cache of [15] requires some form of storage, e.g., `cons`. The trick in this paper is to go to second-order cons-free form by replacing the cache by a function $c : C = (Arguments \to Outputs)$ and simply applying c when arguments are to be looked up in the cache. The cache c must be updated at the return of every function, so $f : A \to B$ is replaced by $f' : A \times C \to B \times C$.

A cached version for the concrete case of the Fibonacci function is:

```
f n (λn.0) where
f n c =
let cv = c n in
  if cv /= 0 then (cv,c) else
    if n <= 1 then (1,update c n 1) else
      let (u,c1) = f (n-1) c in
        let (v, c2) = f (n-2) c1 in
          let r = u+v in (r,update c n r)
update c n v =
   if c n == v then c else λm.if m==n then v else c m
```

This program runs in polynomial time: it makes $O(n)$ calls of `f` when computing `f(n)`, and the time spent checking the cache contents is also polynomially limited.

⊣

A conjecture strengthening Theorem 1:

Distillation can transform any first-order `cons`-free program into an equivalent second-order `cons`-free program that runs in polynomial time.

Basis for the conjecture: it seems plausible that distillation, if applied to an arbitrary first-order `cons`-free program `p`, can transform it into an equivalent second-order program that runs in polynomial time. Reasoning: the transformation of the proof above, if applied to a general first-order `cons`-free program, seems analogous to the transformation that the distiller realised for the "palindrome" program.

More generally, each of the "accumulating parameters" that distillation generated for the reverse, factorial sum and Fibonacci examples resembles a cache with static structure. Renaining to investigate is whether distillation can yield an accumulating parameter the corresponds to a cache with dynamic structure, as seen in the previous program. (Perhaps a static analysis of the Fiboonacci code above could reveal that `c` is used in a static manner.)

6 Final remarks and conclusions

6.1 A question to be resolved

A better understanding is emerging on the source of these interesting program optimisations, though some questions are still less than perfectly clear. An example: although the "supercompilation" that distillation is based on [12, 21, 23] yields at most linear speedups, distillation sometimes achieves superlinear speedups. The major technical difference (at transformation-time) is that distillation does "generalisation" by a form of second-order pattern matching.

The question: why and how does this make such a difference in the efficiency of transformed programs? Answering this will require a better global insight into second-order generalisation.

6.2 Are there limits to speedup by distillation?

The fact that distillation often yields linear-time programs may at first seem to conflict with well-known results from complexity theory [14]: for example, for any computable function f, there exist computational problems that cannot be solved in time $\leqslant f(n)$ by any program. Consequence: there must be some limit to how much transformation techniques such as distillation can achieve, regardless of how strong the techniques used are.

An interesting question: Is there some sense in which distillation achieves a best possible result, e.g., analogous to a minimal-state finite automaton? This might be so.

However, by Blum's speedup theorem [2] some functions have no best program, precluding the possibility that distillation can always achieve the best possible result in terms of efficiency. Furthermore, the output of distillation must always be a finite program. This requirement could force the output program to be asymptotically less efficient than an infinite LTS $\mathcal{D}[\![p]\!]$ resulting from driving the input program.

6.3 An analogy with the Myhill-Nerode theorem

The Myhill-Nerode theorem [20] concerns definability of sets of finite strings over a finite alphabet, for example a set $L \subseteq \{0,1\}^*$. The starting point is to define an equivalence relation over finite strings $x, y \in \{0,1\}^*$ by

$$x \equiv y \text{ iff } \forall z \,.\, (xz \in L \Leftrightarrow yz \in L)$$

Theorem L is a regular set if and only if the relation $\equiv$ has only finitely many equivalence classes. Furthermore, a minimal-state finite automaton M_L that accepts exactly L can be constructed from $\equiv$.

An interesting fact: the relation $\equiv$ is well-defined for *any* subset $L \subseteq \{0,1\}^*$, whether regular or not. If L is not regular, then M_L will have infinitely many states. In all cases, M_L is a homomorphic image of any automaton (finite- or infinite-state) that accepts L.

A consequence is that one can perform *state minimisation* of an initial automaton M by first constructing the relation $\equiv$ for the set accepted by M, and then constructing M_L from the equivalence classes of $\equiv$.

The analogy: In the case of distillation, an initial program p is given, and the possibly infinite LTS $\mathcal{D}[[p]]$ is constructed from it by driving. Once this is available, the distillation step is applied to construct from it another (finite) program p' that will often be faster than the original program p.

While the goal criteria for theMyhill-Nerode construction and distillation differ (smaller-size state sets for DFA minimisation versus asymptotically faster programs for distillation), the overall pattern seems tantalisingly similar.

6.4 Conclusions

In spite of many remaining open quesions, we hope the material above, particularly Sections 3 and 4, clarifies the way that distillation can yield superlinear program speedups.

The question "how can an $O(n^2)$ program or $O(2^n)$ program be bisimilar to an $O(n)$ program?" has been answered: It is not the runtime state transitions of the two programs that are bisimilar; but rather their driven versions. Furthermore, the numbers of states in their driven versions trace the number of `cons`'s performed, and so reflect the two programs' relative running times.

Finally, a large program set has been identified in which superlinear speedups are likely to be achievable by by distillation: the first-order `cons`-free programs.

Acknowledgement: This paper has been much improved as a result of discussions with Luke Ong and Jonathan Kochems at Oxford University. Referee comments, particularly by Neil Mitchell, were very useful. The work was supported, in part, by DIKU at the University of Copenhagen, and by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

1. Bird, R.: Improving programs by the introduction of recursion. Commun. ACM 20(11), 856–863 (1977)
2. Blum, M.: A machine independent theory of the complexity of recursive functions. J. ACM 14, 322–336 (1967)
3. Burstall, R., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (Jan 1977)
4. Chin, W.N.: Towards an automated tupling strategy. In: PEPM. pp. 119–132. ACM (1993)
5. Chin, W.N., Khoo, S.C., Jones, N.: Redundant call elimination via tupling. Fundam. Inform. 69(1-2), 1–37 (2006)
6. Debois, S.: Imperative program optimization by partial evaluation. In: PEPM (ACM SIGPLAN 2004 Workshop on Partial Evaluation and Program Manipulation). pp. 113–122 (2004)

7. Ershov, A.P.: On the essence of compilation. In: Neuhold, E. (ed.) Formal Description of Programming Concepts. pp. 391–420. MAsterdam: North-Holland (1978)
8. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Higher-Order and Symbolic Computation 12(4), 381–391 (1999)
9. Gordon, A.D.: Bisimilarity as a theory of functional programming. Theoretical Computer Science 228(1–2), 5–47 (1999)
10. Hamilton, G.W.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 61–70 (2007)
11. Hamilton, G.W., Jones, N.D.: Distillation with labelled transition systems. In: PEPM (ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation). pp. 15–24. ACM (2012)
12. Hamilton, G.W., Jones, N.D.: Proving the correctness of unfold/fold program transformations using bisimulation. In: Proceedings of the 8th Andrei Ershov Informatics Conference. Lecture Notes in Computer Science, vol. 7162, pp. 150–166. Springer (2012)
13. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
14. Jones, N.D.: Computability and complexity - from a programming perspective. Foundations of computing series, MIT Press (1997)
15. Jones, N.D.: The expressive power of higher-order types or, life without cons. J. Funct. Program. 11(1), 5–94 (2001)
16. Jones, N.D.: Transformation by interpreter specialisation. Sci. Comput. Program. 52, 307–339 (2004)
17. Lacey, D., Jones, N.D., Wyk, E.V., Frederiksen, C.C.: Compiler optimization correctness by temporal logic. Higher-Order and Symbolic Computation 17(3), 173–206 (2004)
18. Lerner, S., Millstein, T.D., Chambers, C.: Cobalt: A language for writing provably-sound compiler optimizations. Electr. Notes Theor. Comput. Sci. 132(1), 5–17 (2005)
19. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
20. Nerode, A.: Linear automaton transformations. In: Proceedings of the AMS 9. pp. 15–24. AMS (1958)
21. Sørensen, M.H., Glück, R., Jones, N.: A Positive Supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)
22. Turchin, V.F.: Supercompilation: Techniques and results. In: Perspectives of System Informatics. LNCS, vol. 1181. Springer (1996)
23. Turchin, V.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 90–121 (Jul 1986)
24. Wadler, P.: Deforestation: Transforming programs to eliminate trees. In: Ganzinger, H. (ed.) ESOP'88. 2nd European Symposium on Programming, Nancy, France, March 1988 (Lecture Notes in Computer Science, vol. 300). pp. 344–358 (1988)

Extracting Data Parallel Computations from Distilled Programs

Venkatesh Kannan and G. W. Hamilton

School of Computing, Dublin City University, Ireland
{vkannan, hamilton}@computing.dcu.ie

Abstract. To effectively utilise the parallel computing power of the heterogeneous architecture in hardware, potential parallelism in programs needs to be extracted and characterised. The extraction of parallel computations in a given program, though challenging and error-prone in practice, should be automated for both efficiency and accuracy of the parallelisation process.
In this paper, we present our initial work to automate the identification of data parallel computations in a given functional program for their execution on heterogeneous hardware with multi-core CPUs and GPUs. To achieve this, we use a program transformation technique called *distillation* [11, 12], and a data type transformation technique used in *AutoPar* [10] to transform an arbitrary program to operate over flat data types. We then choose a set of *skeletons* that are widely used for parallel program development [8,9,13,14], and use their characteristics to identify and extract instances of the skeletons from the transformed program that operates over flat data types. Following this, we replace these skeleton instances with their equivalent operations in the *Accelerate* library [4], which provides efficient OpenCL implementations for their execution on multi-core CPUs and GPUs.
We are presently working on formally specifying our parallelisation process, before comprehensively evaluating the parallel programs produced by our approach against expert hand-written parallel programs.

1 Introduction

The architecture of today's computing systems is made up of a heterogeneous collection of parallel processing units. The most common parallel processing units found are multi-core CPUs and many-core GPUs. CPUs are better suited to efficiently executing latency-critical programs that may have dynamic control-flow. GPUs, on the other hand, are designed for efficient execution of throughput-critical programs that have minimal control-flow divergence and a large number of identical threads.

In this setting, the development of parallel programs is vital to harness the computing power available in hardware. When it comes to parallelisation of programs and their execution, there are some tasks to be done either by the programmer, or by the implementor of the parallel programming system [8].

- *Problem decomposition*: Identification of computations in a program that can be executed in parallel.
- *Distribution*: A mapping from computations that may be executed in parallel to the available processing units.
- *Code and data sharing*: Decisions on how to spread the code and data for the computations to be executed in parallel across the chosen architecture, aiming at a performance improvement over a sequential execution.
- *Communication and synchronisation*: A mechanism that describes resource sharing and control.

Parallelism in a program can be *implicit* or *explicit* depending on which of the above tasks are specified by the programmer, and which are specified by the implementor in the programming system [8]. A completely explicit parallel program will have all four of the above tasks specified by the programmer, while a completely implicit parallel program will have all of them implemented in the parallel programming system.

Parallelisation of a given program, on the other hand, can be either *manual* or *automated*. In manual parallelisation, given the existing complexity of implementing an algorithm from its design, manually identifying and expressing parallel code can make development tedious and error-prone. Alternatively, automated program parallelisation involves identifying computations in a program that exhibit parallelism through program analysis. Such parallel computations can then be extracted and expressed explicitly using program transformation techniques. However, in practice, such automated parallelisation can be quite difficult for an arbitrary given program, especially while targeting its execution on a heterogeneous parallel architecture.

In this paper, we present our initial work that uses a program transformation technique called *distillation* [11,12], and extracts potential parallel computations from *distilled* programs. This automates the task of problem decomposition, thus making potential parallelism in a given program more obvious. For the purpose of this paper, a detailed description of distillation is not required; it is sufficient to know that the *distilled* expressions are in a specialised form called *distilled form*. To identify and extract parallel computations, we choose a set of *skeletons*, which are algorithmic forms that are common to a wide range of parallelisable problems [8]. The extracted parallel computations are then scheduled for execution on CPUs and GPUs based on their characteristics.

The remainder of this paper is structured as follows. In Section 2, we define the syntax and semantics of the higher-order functional language which we use in the parallelisation process. In Section 3, we elaborate on the characteristics of parallel computations that we use to decide their scheduling on a CPU or a GPU for execution. Also presented in this section are the functional definitions of the chosen skeletons to encompass these characteristics. In Section 4, we present our method to transform data types of a given program into a form that makes it amenable to parallelisation, and our parallelisation technique. In Section 5, we outline the course planned to complete this work. In Section 6, we summarise by considering related work in this context.

2 Language

In this work, we focus on program parallelisation applied to functional languages. This is primarily due to certain advantages that functional languages have. The lack of side-effects in pure functional languages is a major benefit, which makes them easier to analyse, reason about, and manipulate using program transformation techniques. The lack of side-effects also allows parallel evaluation of independent sub-expressions in a program. The higher-order functional language used in this work is presented in Definition 1.

Definition 1 (Language Syntax).

$$
\begin{array}{lll}
e ::= x & & \textit{Variable} \\
\quad \mid\ c\ e_1 \ldots e_k & & \textit{Constructor Application} \\
\quad \mid\ \lambda x.e & & \lambda-\textit{Abstraction} \\
\quad \mid\ f & & \textit{Function Call} \\
\quad \mid\ e_0\ e_1 & & \textit{Application} \\
\quad \mid\ \textbf{case}\ e_0\ \textbf{of}\ p_1 \rightarrow e_1 \mid \ldots \mid p_k \rightarrow e_k & & \textit{Case Expression} \\
\quad \mid\ \textbf{let}\ x = e_0\ \textbf{in}\ e_1 & & \textit{Let Expression} \\
\quad \mid\ e_0\ \textbf{where}\ f_1 = e_1, \ldots,\ f_n = e_n & & \textit{Local Function Definitions} \\
\\
p ::= c\ x_1 \ldots x_k & & \textit{Pattern}
\end{array}
$$

A program in this higher-order language is an expression e, which can be a variable, constructor application, $\lambda-$abstraction, function call, application, **case**, **let**, or **where**. Any variables introduced in the $\lambda-$abstraction, **case** patterns or **let** are *bound*, while all other variables are *free*. Each constructor has a fixed arity. In an expression $c\ e_1 \ldots e_k$, k must be equal to the arity of the constructor c. The patterns in **case** branches may not be nested. Techniques exist to transform nested patterns into equivalent non-nested versions [1,19]. No variable may appear more than once within a pattern and it is also assumed that all patterns are non-overlapping and exhaustive.

3 Parallel Computations and Skeletons

To be able to identify potential parallelism in a given functional program, we classify parallel computations into two categories - *data parallel* and *task parallel* computations. Since GPUs are capable of efficiently executing a large number of identical threads that have minimal control-flow divergence, data parallel computations are better suited for execution on GPUs. However, the cost of transferring data between the system main memory and the GPU memory is non-trivial. Hence, given enough computational work per unit data, data parallel computations that operate on significantly large datasets will have a larger performance gain when executed on GPUs. All other computations (sequential, data parallel

computations operating on smaller datasets, and task parallel computations) are scheduled for execution on CPUs.

Data parallelism can be further classified as *flat* and *nested* data parallelism. Flat data parallelism executes more efficiently on GPUs, as opposed to nested data parallelism. This is because flat data parallelism is closer to the Single Program Multiple Data (SPMD) model that the GPU hardware is based on. Also, the control-flow is more regular and less divergent in the case of flat data parallelism allowing high throughput during execution. Hence, our objective is to

1. transform any given program to operate over flat data types, and
2. identify all potential flat data parallel computations in the transformed program.

We represent flat data parallel computations using *skeletons*, which are algorithmic forms that are common to a wide range of parallelisable problems. Our choice of skeletons is based on Blelloch's work on a vector-model for data parallel computations [2] that includes a study of primitive operations required to implement data parallel computations. To encompass the characteristics of data parallelism, we choose three *skeletons* − *map*, *reduce* and $zipWith$. These skeletons are also widely used in the development of programs that have data parallel computations [8,9,13,14].

The three skeletons defined over lists are presented in Definition 2.

Definition 2 (Skeletons Defined over Lists).

$$
\begin{array}{l}
map\ f\ xs \\
\textbf{\textit{where}} \\
map = \lambda f.\lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad Nil \rightarrow Nil \\
\qquad\qquad\qquad Cons\ x'\ xs' \rightarrow Cons\ (f\ x')\ (map\ f\ xs')
\end{array}
$$

$$
\begin{array}{l}
reduce\ v\ f\ xs \\
\textbf{\textit{where}} \\
reduce = \lambda v.\lambda f.\lambda xs.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \rightarrow v \\
\qquad\qquad\qquad\qquad Cons\ x'\ xs' \rightarrow reduce\ (f\ x'\ v)\ f\ xs'
\end{array}
$$

$$
\begin{array}{l}
zipWith\ f\ xs\ ys \\
\textbf{\textit{where}} \\
zipWith = \lambda f.\lambda xs.\lambda ys. \\
\qquad\qquad \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad Nil \rightarrow Nil \\
\qquad\qquad\qquad Cons\ x'\ xs' \rightarrow \textbf{case}\ ys\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Nil \rightarrow Nil \\
\qquad\qquad\qquad\qquad\qquad\qquad Cons\ y'\ ys' \rightarrow Cons\ (f\ x'\ y') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (zipWith\ f\ xs'\ ys')
\end{array}
$$

- *map* : applies a function f to each element in a list xs, and produces a list of the same size as that of the input.
- *reduce* : collapses a list xs into a single value using an associative binary operator f, with a unit value v, by accumulating the reduction value.
- $zipWith$: combines two lists xs and ys point-wise using a binary operator f on the corresponding elements.

4 Program Parallelisation

Before we apply our parallelisation technique to identify potential data parallel computations in a program, we need to identify if the program may contain data parallelism. In this context, we observe that only some of the data types over which a program is defined are suited to data parallelism such as lists, trees or arrays. Hence, we allow the developer to specify a set of *parallelisable types*, γ, to which our parallelisation technique can be applied.

In our parallelisation approach, we identify instances of the three list skeletons in the result of applying distillation to a given program. To facilitate this, the parallelisable types within a program need to be converted to flat lists. Hence, it is necessary to transform the program using distillation to operate over these flat lists in order to identify potential flat data parallel computations using our skeletons. This is explained in Section 4.1.

Upon transformation of the original program f to operate over lists, we obtain the transformed program f_{list}. Following this, we use the characteristics of the skeletons that are presented in Section 3 to identify and extract their instances from f_{list}. This is explained in Section 4.2.

Our approach to execute the identified skeletons efficiently on CPU or GPU is explained in Section 4.4.

4.1 Data Type Transformation

We use two sets of functions to transform a program f with input data of type T_{in} and output data of type T_{out}, into a semantically equivalent one, f_{list}, defined on list data types.

For input type $T_{in} \in \gamma$, and output type $T_{out} \in \gamma$,

- $flatten_{in}$ and $flatten_{out}$: these functions transform T_{in} and T_{out} into lists of their component types T'_{in} and T'_{out}.
- $unflatten_{in}$ and $unflatten_{out}$: these functions transform lists of component types T'_{in} and T'_{out} back to their corresponding original data types T_{in} and T_{out}.

The type signatures of these functions are presented in Definition 3. These functions use Dever's work on data partitioning [10] that defines functions $flatten$ and $unflatten$ to provide transformations between an instance of type T and a list of its component types T'.

Definition 3 (Signatures of Type Transformation Functions).

$$flatten_{in} :: T_{in} \rightarrow (List\ T'_{in})$$
$$unflatten_{in} :: (List\ T'_{in}) \rightarrow T_{in}$$

$$flatten_{out} :: T_{out} \rightarrow (List\ T'_{out})$$
$$unflatten_{out} :: (List\ T'_{out}) \rightarrow T_{out}$$

It is to be noted that if $T_{out} \in \gamma$, as in the case of *map* and $zipWith$ skeletons, then $flatten_{out}$ transforms T_{out} into $List\ T'_{out}$. If $T_{out} \notin \gamma$, as in the case of *reduce* skeleton, then the output of $flatten_{out}$ is the original output data type T_{out}. The function $unflatten_{out}$ also works in a similar fashion.

As illustrated in Fig. 1, a composition of $unflatten_{in}$, the original program f, and $flatten_{out}$ yields a program that is defined over list data types.

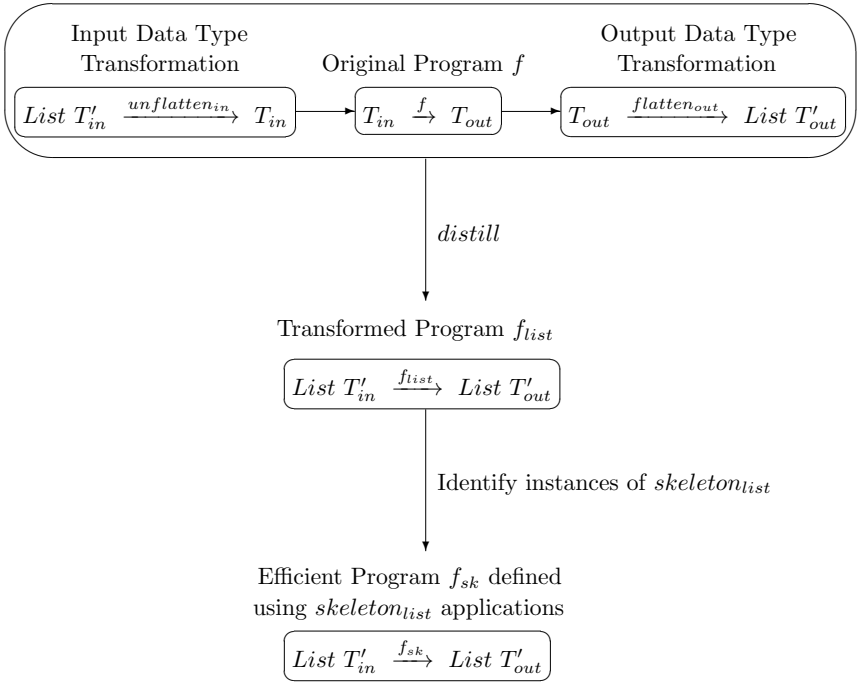


Fig. 1. Transformation of Original Program

4.2 Parallelisation Technique

Fig. 1 also illustrates our parallelisation technique. As a first step, we *distill* the composition of $unflatten_{in}$, f and $flatten_{out}$. This yields a program f_{list}, which is semantically equivalent to f and is defined on list data types.

The definition of f_{list} is in the *distilled form*, which is presented in Definition 4. In an expression in the distilled form, de^{ρ}, resulting from distillation, ρ denotes the set of variables that have been introduced in **let** expressions, and cannot therefore appear as a selector in a **case** expression. As a result of this, expressions in distilled form do not create intermediate data structures.

Definition 4 (Syntax of Distilled Form).

$$
\begin{array}{lll}
de^{\rho} ::= & x & \\
| & c\ de_1^{\rho} \ldots de_k^{\rho} & \\
| & \lambda x.de^{\rho} & \\
| & f & \\
| & de^{\rho}\ x & \\
| & \textbf{case}\ x\ \textbf{of}\ p_1 \rightarrow de_1^{\rho} \mid \ldots \mid p_k \rightarrow de_k^{\rho} & \textit{where}\ x\ \notin \rho \\
| & \textbf{let}\ x = de_0^{\rho}\ \textbf{in}\ de_1^{(\rho\ \cup\ \{x\})} & \\
| & f\ x_1\ \ldots\ x_n\ \textbf{where}\ f = \lambda x_1\ \ldots\ \lambda x_n.de^{\rho} &
\end{array}
$$

Additionally, we have found that the three skeletons described in Section 3 can be associated with the following three characteristics of recursive functions that a given program in distilled form may have.

- *Case 1* : A recursive function has one decreasing parameter, which is of the same type as the result. This would indicate the presence of a *map*-like computation.
- *Case 2* : A recursive function has one decreasing parameter, which is of a different type to the result. This would indicate the presence of a *reduce*-like computation.
- *Case 3* : A recursive function has more than one decreasing parameter. This would indicate the presence of a *zipWith*-like computation.

In addition to problems that fit one of the three cases mentioned above, we may also have problems with any combination of these cases indicating the need for a composition of skeletons.

Using these characteristics of sub-expressions, we identify instances of the skeletons in the transformed program f_{list}. As a result, instances of the skeletons embedded in f_{list} are extracted. This yields the program f_{sk} that is defined using applications of the skeletons $skeleton_{list}$.

4.3 An Example : Find Maximum

The parallelisation of a program, $findMax$, to find the largest positive element in a list using the proposed parallelisation approach, is presented in Example 1.

The input program to the transformation process is shown in expression (1). Here xs is the list to be parsed through to find the largest element. The definition consists of two functions: *bigger* to find the larger of two given elements, and $findMax$ to find the largest positive element in a given list.

The distillation of expression (1), without identifying instances of list skeletons, produces the distilled form of $findMax$ presented in expression (2).

Expression (3) is the result of identifying list skeletons in expression (2) using our parallelisation technique. Here, f is the reduction operation that distillation has extracted in a definition that uses an application of the *reduce* skeleton.

Example 1 (Find Maximum in List).

Expression (1) : Original Program

$$
\begin{array}{l}
findMax\ xs\ 0 \\
\textbf{where} \\
findMax = \lambda xs.\lambda v.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \quad\quad\ \ \rightarrow v \\
\qquad\qquad\qquad\qquad Cons\ x'\ xs' \rightarrow bigger\ x'\ (findMax\ xs'\ v) \\
bigger \quad\ = \lambda x.\lambda v.\textbf{case}\ (x > v)\ \textbf{of} \\
\qquad\qquad\qquad\qquad True \quad\quad\ \ \rightarrow x \\
\qquad\qquad\qquad\qquad False \quad\quad \rightarrow v
\end{array}
$$

Expression (2) : Distilled Program

$$
\begin{array}{l}
findMax\ xs\ 0 \\
\textbf{where} \\
findMax = \lambda xs.\lambda v.\textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Nil \quad\quad\ \ \rightarrow v \\
\qquad\qquad\qquad\qquad Cons\ x'\ xs' \rightarrow \textbf{let}\ v' = (\textbf{case}\ (x' > v)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad True \ \rightarrow x' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad False \ \rightarrow v) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ findMax\ xs'\ v'
\end{array}
$$

Expression (3) : Distilled Program with Skeletons Identified

$$
\begin{array}{l}
findMax\ xs\ 0 \\
\textbf{where} \\
findMax = \lambda xs.\lambda v.\textbf{let}\ f = \lambda x'.\lambda v'.(\textbf{case}\ (x' > v')\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad True \ \rightarrow x' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad False \ \rightarrow v') \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ reduce\ v\ f\ xs
\end{array}
$$

For parallel evaluation of the application of *reduce* skeleton, it is required that the reduction operator be associative. As a result, we have to prove the associativity of a reduction operator, f, used by the extracted *reduce* skeleton. This

can be achieved by distilling the two expressions $f\ (f\ x\ y)\ z$ and $f\ x\ (f\ y\ z)$ using the definition of f. If the distilled forms of both expressions are syntactically equal, then f is associative.

4.4 Execution of Data Parallel Computations

The skeleton applications identified in the distilled program represent data parallel computations. Skeleton applications that work on smaller datasets are scheduled for execution on CPU, while those that are computation intensive and work on significantly larger datasets are scheduled for execution on GPU. This is due to the potentially larger overhead involved in shipping the data between the system main memory and the GPU memory. To allow the execution of the skeleton applications on CPUs and GPUs alike, we make use of the *Accelerate* library of operations [4].

Accelerate Library. This is a domain-specific purely functional high-level language embedded in Haskell. The library contains efficient data parallel implementations for many operations including the chosen skeletons : *map*, *reduce* and *zipWith*. We replace the identified skeleton applications with calls to the corresponding *Accelerate* library operations, which have efficient OpenCL implementations. This allows their scheduling and execution on CPUs and GPUs, among other OpenCL-compatible processing units.

The *Accelerate* library operations are defined over their custom *Array sh e* data type. Here, *sh* is a type variable that represents the shape of the *Accelerate* array. It is implemented as a heterogeneous *snoc*-list where each element in the list is an integer to denote the size of that dimension. A scalar valued array is represented in *sh* by Z, which acts as both the type and value constructor. A dimension can be added to the array by appending the size of that dimension to *sh*. The type variable *e* represents the data type of the elements stored in the *Accelerate* array.

To execute the data parallel computations in f_{sk} on a CPU or GPU, we replace each application of $skeleton_{list}$ with a call to the corresponding *Accelerate* library operation $skeleton_{acc}$. The resulting $skeleton_{acc}$ calls operate over the *Accelerate* array types; inputs of type $Array\ sh_{in}\ T_{in}^{sk'}$, and output of type $Array\ sh_{out}\ T_{out}^{sk'}$. Consequently, we need to transform the input and output data of the original program f to and from the *Accelerate* array type.

These transformations are illustrated in Fig. 2 and Fig. 3 as "Input Data Transformation" and "Output Data Transformation", and are explained below:

1. *Input Data Transformation*
   - $flatten_{in}$: This function transforms the input data for f of type $T_{in} \in \gamma$ into $List\ T'_{in}$ for input to f_{sk}.
   - Each skeleton application $skeleton_{list}$ in f_{sk} operates over input data of type T_{in}^{sk}. We replace these $skeleton_{list}$ applications with calls to corresponding $skeleton_{acc}$ operations that operate over *Accelerate* array types.

- *toAcc* : This function transforms the input data for $skeleton_{list}$ of type T_{in}^{sk} into an *Accelerate* array of type $Array\ sh_{in}\ T_{in}^{sk'}$.
  To define *toAcc*, we use the $fromList$ function that is available in the *Accelerate* library [3], which creates an *Accelerate* array from a list.

2. *Output Data Transformation*
   - $fromAcc$: This function transforms the output from a $skeleton_{acc}$ operation of type $Array\ sh_{out}\ T_{out}^{sk'}$ back to the output type T_{out}^{sk} of the corresponding $skeleton_{list}$ application.
     To define $fromAcc$, we make use of the *toList* function that is available in the *Accelerate* library, which converts an *Accelerate* array into a list.
   - This output from the $skeleton_{list}$ application is then plugged back into its context in f_{sk}.
   - $unflatten_{out}$: This function transforms the output from f_{sk} of type $List\ T'_{out}$ back to the output of type T_{out} of the original program f.

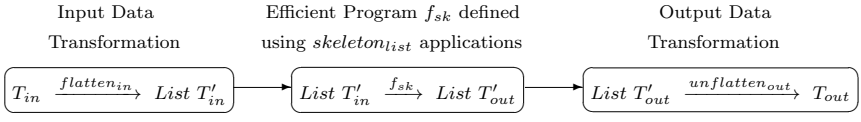


Fig. 2. Transformation of Data for Transformed Program f_{sk}

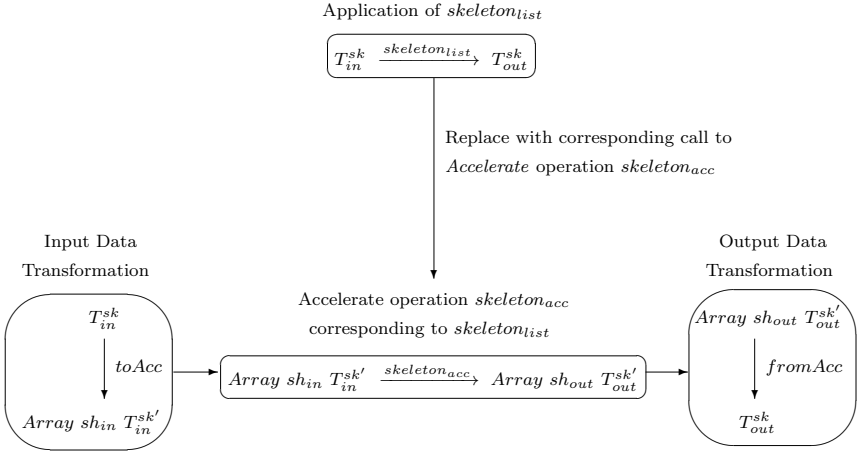


Fig. 3. Transformation of Data for *Accelerate* Operations $skeleton_{acc}$

5 Future Work

We are currently working on formally specifying the transformations rules to parallelise a distilled program, which was described in Section 4. Following this, we will comprehensively evaluate our approach qualitatively and quantitatively.

To evaluate our parallelisation approach, we require a suite of benchmark programs with diverse definitions for each. These programs will include vector dot-product, point-wise vector/matrix arithmetic, matrix multiplication, search algorithm, sort algorithm, histogram generation, image rotation, image convolution, string reverse algorithm, and maximum segment sum algorithm.

One part of our evaluation will be a qualitative analysis of the *coverability* of our representation of parallel computations and of the proposed parallelisation technique. We define *coverability* as the ability to identify potential data parallel computations in a spectrum of diverse definitions of benchmark programs. All the programs we have chosen for benchmarking have potential data parallelism. By distilling these programs and applying our transformation to identify the skeletons, we evaluate their coverability over different forms in which data parallel computations may be expressed in the benchmark programs. Following this, we will address the inclusion of additional skeletons or combinations of skeletons to identify data parallel computations that are otherwise manually identifiable.

Another part of our evaluation will be a quantitative analysis of the performance of our parallelisation technique and the execution of the parallelised programs that are produced. We will perform a quantitative analysis using the benchmark programs by comparing the performance metrics listed below for different configurations of CPU-GPU based hardware and OpenCL program execution environment settings such as varying sizes of datasets, number of threads created, number of work-groups, and work-group sizes. We will also determine the overhead involved in program and data transformation as a factor of the difference in execution times of parallelised and non-parallelised versions of the benchmark programs.

The performance metrics for the quantitative analysis are:

1. *Execution time*
   (a) Time to transfer data between system main memory and GPU memory.
   (b) Time to transfer code from system main memory to GPU memory.
   (c) Time to execute the identified data parallel computations on GPU, and remaining computations on CPU.
   (d) Time to execute the parallelised input program on a multi-core CPU.
   (e) Time to execute the given input program without parallelisation on CPU.
2. *Program transformation time*
   (a) Time to parallelise a given input benchmark program using our approach.
3. *Data transformation time*
   (a) Time to $flatten$ input data into lists.
   (b) Time to transform input data into *Accelerate* array using $fromList$.
   (c) Time to transform output data from *Accelerate* operations using $toList$.
   (d) Time to $unflatten$ the output data to get the final result.

With respect to OpenCL code, metrics 1a, 1b and 1c listed above will be collected by time-stamping the corresponding OpenCL APIs that are used to transfer code and data between the system main memory and the GPU memory, and to schedule the execution of OpenCL kernel functions on the device.

To collect the metrics 1c, 1d, 1e, 2a, 3a, 3b, 3c and 3d that are related to the execution of computations on CPU, we intend to use the *Criterion* [17] or the *ThreadScope* [18] performance measurement packages for Haskell.

6 Conclusion and Related Work

To summarise, we automate the process of extracting potential parallelism in a given functional program to enable its execution on a heterogeneous architecture with CPUs and GPUs. For this, we choose a set of skeletons that are widely used to define data parallel computations in programs. Presently, we aim to use the characteristics of these skeletons to identify and extract their instances from programs in distilled form. Applications of the identified skeleton instances will then be replaced with calls to the corresponding operators in the *Accelerate* library, which provides efficient implementations for the skeletons in OpenCL so that they can be executed on multi-core CPUs and GPUs. In the next steps, we will complete the transformations required to call *Accelerate* library operators, and evaluate the efficiency of the enlisted skeletons to identify potential parallelism in a suite of benchmark programs. This will also include evaluation of the speedups gained from executing the data parallel computations on the GPU. Finally, we plan to investigate enlisting additional skeletons to encompass more parallel computation patterns including task parallelism, and address a wider range of input programs.

Previously, the use of skeletons as building blocks during program development has been widely studied. The early works by Murray Cole [8] and Darlington et. al. [9] exhaustively investigate the use of higher-order skeletons for the development of parallel programs. They present a repertoire of skeletons that cover both task parallel and data parallel computations that may be required to implement algorithms. Later on, addressing the possible difficulties in choosing appropriate skeletons for a given algorithm, Hu et. al. proposed a transformation called *diffusion* in [13]. Diffusion is capable of decomposing recursive definitions of a certain form into several functions, each of which can be described by a skeleton. They also present an algorithm that can transform a wider range of programs to a form that is decomposable by diffusion. This work has further led to the *accumulate* skeleton [14], a more general parallel skeleton to address a wider range of parallelisable problems.

Following the use of skeletons as building blocks in parallel program development, there has been significant work on including parallel primitives as an embedded language in widely used functional languages such as Haskell. One of the earliest works can be traced to Jouret [16], who proposed an extension to functional languages with data parallel primitives and a compilation strategy onto an abstract SIMD (Single Instruction Multiple Data) architecture. Obsid-

ian is a language for data parallel programming embedded in Haskell, developed by Claessen et. al. [6]. Obsidian provides combinators in the language to express parallel computations on arrays, for which equivalent C code is generated for execution on GPUs. An evaluation by Alex Cole et. al. [7] finds that the performance results from generating GPU code from Haskell with Obsidian are acceptably comparable to expert hand-written GPU code for a wide range of problems. Among others, *Accelerate* [4] provides a domain-specific high-level language that works on a custom array data type, embeddeded in Haskell. Using a library of array operations, which have efficient parameterised CUDA and OpenCL implementations, *Accelerate* allows developers to write data parallel programs using the skeleton-based approach that can be executed on GPUs.

Despite the extensive work on identifying and developing skeletons, these approaches require manual analysis and identification of potential parallelism in a problem during development. As stated earlier, this can be quite tedious in non-trivial problems. On the other hand, a majority of the literature on skeletons involve *map* and *reduce* for data parallel computations, which are integral in our work to automate parallelisation. We include the *zipWith* skeleton to address problems that operate on multiple flat datasets.

Another option for a parallel programmer is Data Parallel Haskell (DPH), an extension to the Glasgow Haskell Compiler (GHC), which supports nested data parallelism with focus on multi-core CPUs. Though flat data parallelism is well understood and supported, and better suited for GPU hardware, nested data parallelism can address a wider range of problems with irregular parallel computations (such as divide-and-conquer algorithms) and irregular data structures (such as sparse matrices and tree structures). To resolve this, DPH, which focuses on such irregular data parallelism, has two major components. One is a vectorisation transformation that converts nested data parallelism expressed by the programmer, using the DPH library, into flat data parallelism [15]. The second component is a generic DPH library that maps flat data parallelism to GHC's multi-threaded multi-core support [5]. It is worth pointing out that our method to automate parallelisation includes flattening steps that are similar to the vectorisation transformation in DPH. This flattening step provides a common form to the transformed input program and our enlisted skeletons, thereby aiding in the extraction of flat data parallel computations.

Acknowledgement

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (`www.lero.ie`).

References

[1] L. Augustsson. Compiling Pattern Matching. *Functional Programming Languages and Computer Architecture*, 1985.

[2] Guy E. Blelloch. Vector Models for Data-Parallel Computing. *The MIT Press*, 1990.
[3] Manuel M. T. Chakravarty, Robert Clifton-Everest, Gabriele Keller, Sean Lee, Ben Lever, Trevor L. McDonell, Ryan Newtown, and Sean Seefried. An Embedded Language For Accelerated Array Computations. *http://hackage.haskell.org/package/accelerate*.
[4] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell Array Codes with Multicore GPUs. *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, 2011.
[5] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. *Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, 2007.
[6] Koen Claessen, Mary Sheeran, and Joel Svensson. Obsidian: GPU Programming In Haskell. *Proceedings of 20th International Symposium on the Implementation and Application of Functional Languages (IFL 08)*, 2008.
[7] Alex Cole, Alistair A. McEwan, and Geoffrey Mainland. Beauty And The Beast: Exploiting GPUs In Haskell. *Communicating Process Architectures*, 2012.
[8] Murray Cole. Algorithmic Skeletons: Structured Management of Parallel Computation. *MIT Press, Cambridge, MA, USA*, 1991.
[9] John Darlington, A. J. Field, Peter G. Harrison, Paul Kelly, D. W. N. Sharp, Qiang Wu, and R. Lyndon While. Parallel Programming Using Skeleton Functions. *Lecture Notes in Computer Science, 5th International PARLE Conference on Parallel Architectures and Languages Europe*, 1993.
[10] Michael Dever and G. W. Hamilton. Automatically Partitioning Data to Facilitate the Parallelization of Functional Programs. *PSI'14, 8th International Andrei Ershov Memorial Conference*, 2014.
[11] G. W. Hamilton. Distillation: Extracting the essence of programs. *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 2007.
[12] G. W. Hamilton and Neil D. Jones. Distillation With Labelled Transition Systems. *Proceedings of the ACM SIGPLAN 2012 workshop on Partial Evaluation and Program Manipulation*, 2012.
[13] Zhenjiang Hu, Masato Takeichi, and Hideya Iwasaki. Diffusion: Calculating Efficient Parallel Programs. *In 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 99)*, 1999.
[14] Hideya Iwasaki and Zhenjiang Hu. A New Parallel Skeleton For General Accumulative Computations. *International Journal of Parallel Programming*, 2004.
[15] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'08)*, 2008.
[16] Guido K. Jouret. Compiling Functional Languages For SIMD Architectures. *Parallel and Distributed Processing, IEEE Symposium on*, 1991.
[17] B. O'Sullivan. The Criterion Package. *http://hackage.haskell.org/package/criterion*.
[18] Satnam Singh, Simon Marlow, Donnie Jones, Duncan Coutts, Mikolaj Konarski, Nicolas Wu, and Eric Kow. The ThreadScope Package. *http://hackage.haskell.org/package/threadscope*.
[19] Philip Wadler. Efficient Compilation of Patten Matching. *S. P. Jones, editor, The Implementation of Functional Programming Languages*, 1987.

On Valentin Turchin's Works on Cybernetic Philosophy, Computer Science and Mathematics

Andrei V. Klimov*

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
klimov@keldysh.ru

Abstract. The history, context, main ideas, motivation and research objectives of Valentin Turchin's works on philosophy, cybernetics, computer science and mathematics are outlined. Valentin Turchin's scientific legacy comprises three parts: cybernetic philosophy with the notion of a metasystem transition as a quantum of evolution; application of the philosophy to analysis of evolution of human society; and application of the philosophy to science — cybernetics, computer science and mathematics. In computer science, his main contributions are the programming language Refal and program transformation technique known as supercompilation. In mathematics, he has developed a new constructive foundation of mathematics referred to as the Cybernetic Foundation of Mathematics.

Keywords: Cybernetic Philosophy, Metasystem Transition Theory, evolution, programming language Refal, supercompilation, Cybernetic Foundation of Mathematics.

1 Introduction

Valentin Fedorovich Turchin (14.2.1931–07.4.2010) was a scholar of a rare kind, for whom the purpose of scientific work was a derivative of his philosophical worldview and the meaning of life.

In this paper an attempt is made to give a bird's eye view of Valentin Turchin's scientific and philosophical legacy. His works on philosophy, society and Cybernetic Foundation of Mathematics are just briefed, while his achievements in computer science are discussed in more detail.

This is a slightly revised and updated version of the Russian paper [11].

* Supported by Russian Foundation for Basic Research grant No. 12-01-00972-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

2 Valentin Turchin's Biography in Brief

After graduation from the Physics Department of Moscow State University in 1952, Valentin Turchin worked at the Institute of Physics and Power Engineering in Obninsk, USSR. Shortly after defending his doctoral thesis on the physics of slow neutrons (which later became a book [17]), he was invited by academician M.V. Keldysh to the Institute of Applied Mathematics of the USSR Academy of Sciences and moved to Moscow in 1964. At that time, he also changed the direction of his scientific work from theoretical physics to philosophy, cybernetics, computer science, and programming. He expressed that theoretical physics was approaching its crisis, and to resolve it, we needed high automation of manipulation of physical and mathematical theories and formal linguistic models by means of computers. As a first step towards that great goal, in the 1960s he developed a programming language *Refal* for processing symbolic information, and in the 1970s laid the foundation for a new program transformation method, which he called *supercompilation*. In those same years a group of undergraduate and graduate students of Moscow State University, Moscow Engineering Physics Institute and the Moscow Institute of Physics and Technology started working with him on the implementation of Refal and supercompilation.

In 1972, Valentin Turchin joined the Central Research Institute of Automation Systems in Construction (TsNIPIASS), where he headed a laboratory. Together with his disciples, he continued to develop Refal and its applications, as well as supercompilation. In summer 1974, he was dismissed for his political and human rights activities (e.g., [15]) and advocacy of Andrei Sakharov and Alexander Solzhenitsyn. Before he and his family had to emigrate to the United States in 1977, he was unemployed and continued his scientific work at home. However weekly research seminars with his students on Tuesdays did not interrupt.

Upon arrival in the United States in 1978, Valentin Turchin worked for one and a half years in the Courant Institute of Mathematical Sciences in New York with Jacob Schwartz, the author of the SETL language. Thereafter, until his retirement in 1999, he was a professor at the City College of the City University of New York (CC CUNY). During these years he made his main contributions to supercompilation (see Section 6 below) and foundation of mathematics (Section 7), and continued development of cybernetic philosophy (Sections 8).

3 Valentin Turchin's Trilogy

In 1977, shortly before his departure from the Soviet Union, Valentin Turchin told about his conception of scientific and philosophical "trilogy", part of which he had already realized, and was going to work on the remaining areas:

The philosophical part comprises the cybernetic philosophy elaborated in his book "The Phenomenon of Science: a cybernetic approach to human evolution" [24]. He introduced the concept of a *metasystem transition* as a uniform quantum of evolution and illustrated it by many examples from the origin of the life to the emergence of the human being and then to human culture with science as

its highest point. A metasystem transition is the emergence of a new level of control over already existing systems, which usually multiply, and construction of a system of a new kind comprised of the control subsystem and a variety of controlled subsystems. From Valentin Turchin's viewpoint, the evolutionary worldview also gives us an insight into moral problems what is good and bad as well as the meaning of life: "good" is what contributes to the evolution and creation, "bad" is destruction and inhibition of evolution. In the basis of the meaning of life lies the idea of creative immortality as a personal contribution to the eternal evolution.

The social part is application of the cybernetic philosophy to the evolution of society and social issues, presented in his book "The Inertia of Fear and the Scientific Worldview" [29]. Here he outlined his view of the history of the mankind, having separated it into three stages, based on the method of integration of people into society: physical coercion (slavery), economic coercion (capitalism) and liberal integration based on ideological "coercion", to which humanity is still going. To refer to that new social organization of the future, he used the old term "socialism". From his viewpoint, the transition from capitalism to socialism is a large-scale metasystem transition. As the evolution is essentially non-deterministic and metasystem transitions can occur in various ways, deadlocks and kickbacks are possible rather then stable forward movement to more and more complicate organization (as it is commonly assumed with the concept of "progress"). The socialism in the Soviet Union was an attempt to build a society, which integrates people based on ideas and spiritual culture, but it failed. Instead of proper socialism, the result was totalitarianism, in which individuals have lost their main trait — the free will, while a successful metasystem transition should create a new level of control to integrate subsystems in the system without destroying the basic features of the subsystems, but further developing them.

The scientific part is application of the concept of a metasystem transition to obtaining scientific results. Valentin Turchin told that a new philosophical ideas are impossible to promote without demonstrating their fruitfulness for specific scientific achievements. Only if a philosophy helps to produce new ideas and to analyze various phenomena in science and society, it will allure people. When Valentin Turchin talked about these plans in 1977, one of his achievements in the field of programming has been created, the second one was at the level of general principles and methods, and he created the third one later in the United States:

- the Refal programming language
- the method of program transformation referred to as supercompilation
- the Cybernetic Foundation of Mathematics

4 Refal as a Metalanguage

Valentin Turchin promoted the view of science as *formal linguistic modeling* of the reality, rather than learning absolute truths [6,24]. In particular, mathemat-

ics studies the properties of formal models rather than any ideal objects. With the advent of computers, the mankind has obtained a tool independent of the human mind, which worked with linguistic models. Since then, a person can avoid routine manual manipulation of symbols and place them on a computer, reserving to himself the development of algorithms and programs. The advent of computers and programming as a human activity is a large-scale metasystem transition in the history of mankind in general and science in particular, aimed at automating manipulation of formal linguistic models.

The next metasystem transition was automation of computer programming and emergence of programming languages and programs that produce programs, that is compilers. The programs themselves became objects of manipulation by programs.

In the mid 1960s, Valentin Turchin created the language Refal [18–20] as a meta-language for processing texts in formal languages to enable further development of this large-scale metasystem transition. In those days, Refal was a language of higher level than the existing languages for processing symbolic information like Lisp, Snobol, etc. From its inception to the end of the 1980s, Refal was actively used in the USSR for writing compilers, macro generators, interpreters, as well as for other kinds of text manipulation and transformation of linguistic models: computer algebra, artificial intelligence, construction and check of mathematical proofs, etc.

A major problem, which Valentin Turchin wanted to solve with the use of Refal and which is still open, is computer verification of mathematical texts from the Bourbaki treatise. Once in early 1970s, Valentin Turchin asked Sergei Romanenko, then a student of Moscow State University, to conduct experiments on expanding definitions in the formal language of the first volume of the Bourbaki treatise "Set Theory". However, even a very simple formula exploded exponentially in size. Being very surprised, he expressed an idea that some metasystem transition had to be performed, but then it was unclear how. Later he conceived supercompilation as a tool for performing such metasystem transitions (see Section 5 below). Although the problem of analysis of mathematical texts like that of the Bourbaki treatise is not solved yet, at least one promising approach that uses supercompilation has been identified [7].

In the area of programming languages, Valentin Turchin expected explosion of the number of domain-specific languages, according to the general principles of his *Metasystem Transition Theory*. In "The Phenomenon of Science" he formulated the *Law of Expansion of the Penultimate Level*, which says that when a new level of control emerges, subsystems of the prior level multiply quantitatively and/or qualitatively. In the 1960s and 1970s, we observed the development of formal and software means for description of languages, systems for programming interpreters and compilers (referred to as compiler compilers), macro generators, etc. This was the making of a metasystem over programming languages. The creation of new languages was facilitated, resulting in the growth of their number and complexity. Valentin Turchin emphasized that language creation is a natural human trait. Each application area must generate its own formal lan-

guage. Considering the importance of this task, he demonstrated how to build a macro system using Refal as a macro language [23].

Interestingly, this prediction initially did not come true as he expected. In the 1980s, macro systems and compiler compilers went out of fashion. By the 1990s, it seemed that new programming languages ceased to appear. But then came the 2000s and construction of domain-specific languages became popular. As in the 1960s and 1970s, we have been observing multiplication of programming languages (e.g., scripting languages, domain-specific languages), data formats (e.g., XML-based ones), interface specification languages, and many other kinds of languages.

5 Emergence of Supercompilation

After the first applications of Refal for formal text processing, having practically observed their complexity, Valentin Turchin started finding ways to further automate the creation of efficient meta-programs, programs that generate and transform programs. Once at a seminar in 1971, he said: "We must learn how to manipulate computer programs like we manipulate numbers in Fortran." Definitions of programming languages should become subject of transformations as well. This is a one more metasystem transition.

For Valentin Turchin, the Metasystem Transition Theory was not just conceptual framework, with which he analyzed phenomena, but also a "guide to action". By learning how metasystem transitions happens, he revealed typical patterns and used them to intentionally construct new metasystem transitions in order to accelerate the development of scientific areas of interest to him. A lot of such material were supplied by metamathematics and mathematical logic. He paid much attention to these topics in his working seminars in Keldysh Institute in early 1970s. At that time he conceived supercompilation as a method of equivalence transformation of Refal programs [21, 22].

First Valentin Turchin expressed these ideas at a series of seminars in winter 1971–72. He wrote on a blackboard a simple program — an interpreter of arithmetic expressions in Refal, and suggested to perform "computation in general form" of a sample expression with variables instead of some numbers. He taught us that introduction of variables and computation in general form is a common metasystem transition used in mathematics, and we should take it for program manipulation as well. At that seminar, having manually performed computations over a function call with an arithmetic expression with variables, he got a text resembling the result of translation of the expression to computer code. This was what is now known as *specialization* of a programs with respect to a part of arguments.

Shortly thereafter, in 1972, he presented this technique in formal form [21]. This paper marks the beginning of the history of supercompilation. However, it contained its basic level only, called *driving* in the next more detailed publication in 1974 [22]. In the first paper [21], he also defined a universal resolving algorithm

(URA) based on driving. By a remarkable coincidence, the Prolog language, which is based on a similar technique, appeared in 1972 as well.

From 1974, being unemployed (for political reasons, as a dissident), Valentin Turchin could not publish papers, and the birth of supercompilation occurred in form of lectures at a series of 7 weekly seminars in the winter 1974–75, which took place in "ASURybProekt", Central Design and Technological Institute for Automated Control Systems of the USSR Ministry of Fisheries, where one of the permanent members of the seminar Ernest Vartapetyan worked.

All of the main ideas of supercompilation were presented in those lectures. In particular, at the third seminar, Valentin told how to produce a compiler from an interpreter as well as a compiler compiler by means of three metasystem transitions using a supercompiler. This was what later became known as *Futamura projections*. (Yoshihiko Futamura conceived these results before Valentin Turchin: the first two projections in 1971 [4], and the third projection — a compiler compiler — in 1973 [5].)

Supercompilation became a classical example of building a complex formal system by means of a number of metasystem transitions:

1. The first metasystem transition over computations: transition from concrete states to representation of sets of states in form of states with variables (referred to as *configurations*), from computation on ordinary values to computation with variables (referred to as *driving*), from computations along one thread to unrolling a potentially infinite tree representing all possible computation threads.
2. The second metasystem transition over driving: folding a (potentially) infinite tree into a finite graph by folding (looping back).
3. The next metasystem transitions over folding operations: various strategies to stop driving, performing generalization and reconstruction of the tree and graph. Valentin Turchin called the second metasystem transition and first versions of the third metasystem transition *configuration analysis*. (Notice that separation into metasystem transitions may be ambiguous, especially when they occurred simultaneously.)
4. Further metasystem transitions over computations and driving: *neighborhood analysis* — generalized computations with two kinds of variables (variable of the second kind are naturally called *meta*variables) in order to obtain not only the tree and graph of computations, but also a set of states and the set of configurations that go along the same way of concrete or generalized computations. Valentin conceived the neighborhood analysis to improve the configuration analysis as the next level of control, allowing to select better points of generalization and folding. (Later Sergei Abramov suggested other interesting application of neighborhood analysis, including testing [1].)

Prior to his departure from the USSR, Valentin Turchin was unable to publish anything on supercompilation proper. Only in a book on Refal [42], published in 1977 without attribution in order not to mention the name of Valentin Turchin, five pages (92-95) had been inserted in Chapter "Techniques of using Refal"

at the end of Section "Translational problem", which contained formulas to generate a compiler and a compiler compilers.

A year before the departure, in Fall 1976, Valentin Turchin learned that Andrei Ershov conducted research into *mixed computation*, which has close objectives. When Andrei Ershov visited Moscow, they met in hotel "Academic" and exchanged ideas. They put into correspondence the Ershov's notion of a *generating extension* and Valentin Turchin's compiler generation formula. In [3], A.P. Ershov called the formula of the second metasystem transition that reduces a generating extension to a specializer (supercompiler), the *double run-though theorem by V.F. Turchin*. (Here "run-through" means "driving".)

6 Further Development of Supercompilation

In his first years in the United States in Courant Institute (1978-1979), Valentin Turchin described the ideas and methods on Refal and supercompilation in a big report [26]. Several of those ideas await further development and implementation still, e.g., walk grammars, the notions of *metaderivative* and *metaintegral* based on the walk grammars, metasystem formulas, self-application. The methods that deal with walks in the graph of configuration lie at the next meta-level compared to the *configuration analysis*, which deal with configurations. This work outlined the next metasystem transition to higher-level supercompilers. However, only recently the first two-level supercompiler [12] has been constructed, although based on other ideas. (Notice that this demonstrates the ambiguity and indeterminacy of metasystem transitions).

In the 1980s, working in CC CUNY together with a small group of students, he created and implemented on IBM PC a new version of Refal, Refal-5 [35], and developed a series of supercompilers for Refal. On those weak computers, they managed to carry out first experiments with supercompilers [25, 32, 43]. He also outlined approaches to problem-solving using supercompilers [27], in particular, to theorem proving [28].

The first journal paper on supercompilation [31], which is highly cited, was published in 1986.

Gradually researchers from other universities joined Valentin Turchin's work on supercompilation, most notably at the Department of Computer Science at the University of Copenhagen, chaired by Neil D. Jones, who has developed another method of program specialization — *partial evaluation*. However here we restrict ourselves to the line of Valentin Turchin's works and just mention some of the related papers of other authors.

In 1987, Dines Bjorner together with Neil Jones and Andrei Ershov organized a Workshop on Partial Evaluation and Mixed Computation in Denmark. At the workshop, Valentin Turchin presented a paper [34] on the first practical *whistle* — a criterion to terminate driving and generalize a configuration. The previous whistles were either too inefficient, or produce bad residual programs in automatic mode, or required help from the user.

In 1989, under "perestroika", Valentin Turchin resumed regular contacts with his Russian disciples. From then on, he visited Russia together with his wife almost every year, and intensive workshops were held in Moscow, Obninsk (1990) and Pereslavl-Zalessky.

In the late 1980s, he wrote a draft book on supercompilation. However, it remained unpublished, since he planed to improve it. The book contained the description of the second version of supercompiler SCP2. The copies of the book chapters were distributed among the participants of supercompilation workshop in Obninsk in summer 1990.

In the early 1990s, he returned back to the idea of building a more powerful supercompiler based on transformation of walk grammars. He restricted the class of grammars to regular expressions (compared to the Report-80 [27]), and presented the methods in paper [38]. However, they were not completed in form of algorithms or strategies and are still waiting for further development.

In 1994, Andrei Nemytykh worked almost a year with Valentin Turchin and they started development of the fourth version of a Refal supercompiler SCP4, which was completed by Andrei during next years [13, 14].

In 1995, Morten Heine Sorensen and Robert Gluck suggested [16] to use a *well-quasi-order* on terms (*homeomorphic embedding*) as a whistle. Valentin Turchin liked this method and since then suggested to use it as a primary whistle in supercompilers. In particular, it was implemented in SCP4.

In 1996, Valentin Turchin wrote two last papers on supercompilation [40, 41] with an overview of achievements and his vision of future work.

A comprehensive bibliography of his works on supercompilation may be found in [8].

Valentin Turchin always thought of putting supercompilation into practice. In 1998 he raised an investment and founded Supercompilers, LLC, with the goal of developing a Java supercompiler [10]. He was fond of this topic, but unfortunately the Java supercompiler has not been completed yet.

In recent years we observe a burst of interest to supercompilation, which Valentin Turchin dreamed about. The method of supercompilation has been polished, simplified and improved by many researchers. A number of experimental supercompilers for simple languages has been developed. But that's another story.

7 Cybernetic Foundation of Mathematics

Another great scientific achievement by Valentin Turchin lies in the foundation of mathematics. He was a constructivist in the spirit close to .. Markov. He said that no actual infinity exists. There are just language models and potentially infinite processes and enumerations. Like A.A. Markov, he sought to reduce all mathematical concepts to constructive algorithmic ones. He was not confused by the failure of the Markov's constructive mathematics to express all mathematical entities as algorithms. He saw the limitations of Markov constructive approach in that it is a closed system incapable of evolution. By expressing everything in

terms of deterministic algorithms, we have a world in which metasystem transitions "degenerate", "saturated", cease to generate new quality.

Valentin Turchin managed to build an "open" constructive system by extending the world of algorithms with a model of the user of mathematics, a mathematician. This is a metasystem transition similar to the appearance of the concept of an observer in the modern physics. The wheels of his theory, referred to as Cybernetic Foundation of Mathematics, model multiple metasystem transitions. In this theory, he actually demonstrated what are formal metasystem transitions in action. As a result, he was able to give constructive interpretation of the notion of a set and the Zermelo-Fraenkel axioms based on the extended notion of an algorithm.

In 1983 he published a book on Cybernetic Foundations of Mathematics as a university report [30]. After that, he squeezed the material to two articles that have been accepted for publication in a journal, but with the condition that he would reduce them to a single article [33]. Unfortunately, the published version is badly readable due to excessive density of presentation. The full version is still awaiting its publication and translation to Russian.

It is naturally to assume that Valentin Turchin planned to combine the works on supercompilation and Cybernetic Foundation. He said he did not believe in the fruitfulness of the methods of machine theorem proving based on logic inference, and sought for constructive approaches. Algorithmic definitions of mathematical notions may become subject to manipulation and inference using methods similar to supercompilation. In turn, supercompilation can be enriched with effective means of constructing proofs of statements about programs.

8 Further Development of Cybernetic Philosophy

In the 1990s, Valentin Turchin continued the work on cybernetic philosophy together with Francis Heylighen and Cliff Joslyn. They founded the Principia Cybernetica Project [9] with a goal to elaborate various philosophic and scientific questions on the common cybernetic ground and to gather a community of researchers interested in these topics. Unfortunately, after about a decade of development the project became silent and wait for new enthusiasts.

In these years he also published several articles on the concept of the metasystem transition [39], cybernetic ontology [36] and epistemology [37]. Further discussion of these very interesting topics requires a separate paper.

9 Conclusion

The supercompilation history dates back for almost 40 years. Valentin Turchin planed to get a program transformation tool to implement his further ideas on manipulation of formal linguistic models. But the history moves much slower. Now that we have several experimental supercompilers, it is clear that it was impossible to implement such a system either on mainframes in the 1970s, or on personal computers in 1980s. But now with modern computers, supercompiler

experiments became more and more successful. The time for such methods to go into wide programming practice has come. Then the great challenge of mass manipulation of formal linguistic models will come to the agenda, and someday these methods will lead to the burst of automated construction of physical theories, what Valentin Turchin dreamed about at the start of his scientific carrier.

Acknowledgements. I was very lucky to meet Valentin Turchin when I was a schoolboy and to belong to a group of students who started their scientific activity under his scientific mentorship. We became friends and professional team-mates for life.

I am very grateful to those of them with whom we still collaborate on the topics founded by our Teacher: Sergei Abramov, Arkady Klimov, Andrei Nemytykh, Sergei Romanenko, as well as to young researches who joined the community of Valentin Turchin's disciples and generate new ideas: Sergei Grechanik, Yuri Klimov, Ilya Klyuchnikov, Anton Orlov, Artem Shvorin, and to other participants of the Moscow Refal seminars.

It's a great pleasure for me to spend time together with Robert Glück, when we have a chance to meet and study Valentin Turchin's legacy and further develop his ideas.

We are very glad to meet and discuss various computer science topics with Neil D. Jones, who noticed Valentin Turchin's works at the early stage and whose deep scientific ideas and achievements have been influencing the works of our team very much.

Finally I express our love and gratitude to Tatiana Turchin for all she has been doing for us and for keeping alive the memory of Valentin Turchin.

References

1. S. M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications)*. Nauka, Moscow, 1995. (In Russian).
2. O. Danvy, R. Glück, and P. Thiemann, editors. *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
3. A. P. Ershov. On the essence of compilation. In E. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.
4. Y. Futamura. Partial evaluation of computing process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
5. Y. Futamura. EL1 partial evaluator (progress report). Internal report submitted to Dr. Ben Wegbreit, Center for Research in Computer Technology, Harward University, Cambridge, MA, USA, January 24, 1973.
6. R. Glück and A. V. Klimov. Metacomputation as a tool for formal linguistic modeling. In R.Trappl, editor, *Cybernetics and Systems '94*, volume 2, pages 1563–1570, Singapore, 1994. World Scientific.
7. R. Glück and A. V. Klimov. Metasystem transition schemes in computer science and mathematics. *World Futures: the Journal of General Evolution*, 45:213–243, 1995.

8. R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In Danvy et al. [2], pages 253–275.
9. F. Heylighen, C. Joslyn, and V. F. Turchin. *Principia Cybernetica Web*. `http://pespmc1.vub.ac.be`.
10. A. V. Klimov. An approach to supercompilation for object-oriented languages: the Java Supercompiler case study. In *First International Workshop on Metacomputation in Russia, Proceedings. Pereslavl-Zalessky, Russia, July 2–5, 2008*, pages 43–53. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008.
11. A. V. Klimov. O rabotakh Valentina Fedorovicha Turchina po kibernetike i informatike (on Valentin Fedorovich Turchin's works on Cybernetics and Informatics). In A. Tomilin, editor, *Proceedings of SORUCOM-2011*, pages 149–154, 2011. `http://sorucom.novgorod.ru/np-includes/upload/2011/09/05/15.pdf`. (In Russian).
12. I. G. Klyuchnikov and S. A. Romanenko. Towards higher-level supercompilation. In A. Klimov and S. Romanenkop, editors, *Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia, July 1-5, 2010, Pereslavl-Zalessky, Russia*, pages 82–101. Ailamazyan Program Systems Instiute of RAS, 2010. `http://meta2010.pereslavl.ru/accepted-papers/paper-info-5.html`.
13. A. P. Nemytykh. *Superkompilyator SCP4: Obshchaya struktura (The Supercompiler SCP4: General Structure)*. URSS, Moscow, 2007. (In Russian).
14. A. P. Nemytykh, V. Pinchik, and V. Turchin. A self-applicable supercompiler. In Danvy et al. [2], pages 322–337.
15. A. D. Sakharov, R. A. Medvedev, and V. F. Turchin. A reformist program for democratization. In S. F. Cohen, editor, *An End to Silence. Uncensored Opinion in the Soviet Union from Roy Medvedev's Underground Magazine Political Diary*, pages 317–327. W. W. Norton & Company, New York, London, 1970. (Reprinted 1982).
16. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *International Logic Programming Symposium, December 4-7, 1995, Portland, Oregon*, pages 465–479. MIT Press, 1995.
17. V. F. Turchin. *Slow Neutrons*. Israel Program for Scientific Translations, Jerusalem, Israel, 1965.
18. V. F. Turchin. Metajazyk dlja formal'nogo opisanija algoritmicheskikh jazykov (A metalanguage for formal description of algorithmic languages). In *Cifrovaja Vychislitel'naja Tekhnika i Programmirovanie*, pages 116–124. Sovetskoe Radio, Moscow, 1966. (In Russian).
19. V. F. Turchin. A meta-algorithmic language. *Cybernetics*, 4(4):40–47, 1968.
20. V. F. Turchin. Programmirovanie na jazyke Refal. (Programming in the language Refal). Preprints 41, 43, 44, 48, 49, Institute of Applied Mathematics, Academy of Sciences of the USSR, Moscow, 1971. (In Russian).
21. V. F. Turchin. Ehkvivalentnye preobrazovanija rekursivnykh funkcij na Refale (Equivalent transformations of recursive functions defined in Refal). In *Teorija Jazykov i Metody Programmirovanija (Proceedings of the Symposium on the Theory of Languages and Programming Methods)*, pages 31–42, 1972. (In Russian).
22. V. F. Turchin. Ehkvivalentnye preobrazovanija programm na Refale (Equivalent transformations of Refal programs). *Avtomatizirovannaja Sistema upravlenija stroitel'stvom. Trudy CNIPIASS*, 6:36–68, 1974. (In Russian).
23. V. F. Turchin. Refal-makrokod (Refal macrocode). In *Trudy Vsesojuznogo seminara po voprosam makrogeneratsii (Proceedings of the All-Union Seminar of Macrogeneration)*, pages 150–165, 1975. (In Russian).

24. V. F. Turchin. *The Phenomenon of Science: A Cybernetic Approach to Human Evolution*. Columbia University Press, New York, 1977.
25. V. F. Turchin. A supercompiler system based on the language REFAL. *SIGPLAN Not.*, 14(2):46–54, Feb. 1979.
26. V. F. Turchin. The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.
27. V. F. Turchin. Semantic definitions in REFAL and automatic production of compilers. In N. D. Jones, editor, *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science*, pages 441–474. Springer-Verlag, 1980.
28. V. F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 645–657. Springer-Verlag, 1980.
29. V. F. Turchin. *The Inertia of Fear and the Scientific Worldview*. Columbia University Press, 1981.
30. V. F. Turchin. The Cybernetic Foundation of Mathematics. Technical report, The City College of the City Univeristy of New York, 1983.
31. V. F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
32. V. F. Turchin. Program transformation by supercompilation. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 257–281. Springer-Verlag, 1986.
33. V. F. Turchin. A constructive interpretation of the full set theory. *The Journal of Symbolic Logic*, 52(1):172–201, 1987.
34. V. 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 531–549. North-Holland, 1988.
35. V. F. Turchin. *Refal-5, Programming Guide and Reference Manual*. New England Publishing Co., Holyoke, Massachusetts, 1989.
36. V. F. Turchin. The cybernetic ontology of action. *Kybernetes*, 22(2):10–30, 1993.
37. V. F. Turchin. On cybernetic epistemology. *Systems Research*, 10(1):3–28, 1993.
38. V. F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.
39. V. F. Turchin. A dialogue on metasystem transition. *World Futures*, 45:5–57, 1995.
40. V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In Danvy et al. [2], pages 481–509.
41. V. F. Turchin. Supercompilation: techniques and results. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Perspectives of System Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25-28, 1996. Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 227–248. Springer, 1996.
42. V. F. Turchin, A. V. Klimov, A. V. Klimov, V. F. Khoroshevsky, A. G. Krasovsky, S. A. Romanenko, I. B. Shchenkov, and E. V. Travkina. *Bazisnyj Refal i ego realizacija na vychislitel'nykh mashinakh (Basic Refal and its implementation on computers)*. GOSSTROJ SSSR, CNIPIASS, Moscow, 1977. (In Russian).
43. V. F. Turchin, R. Nirenberg, and D. Turchin. Experiments with a supercompiler. In *Conference Record of the ACM Symposium on Lisp and Functional Programming*, pages 47–55. ACM Press, 1982.

Construction of Exact Polyhedral Model for Affine Programs with Data Dependent Conditions

Arkady V. Klimov

Institute of Design Problems in Microelectronics, Russian Academy of Sciences, Moscow, Russia
arkady.klimov@gmail.com

Abstract. Parallelizing compilers usually build polyhedral model (PM) for program parts which are referred to as static control parts (SCoP) and normally include regular nested loops with bounds and array indices being affine expressions of surrounding loop variables and parameters. Usually, PM has the form of a parametric (depending on integer parameters) graph that connects all array reads with respective array writes. Sometimes certain extensions of program class are allowed. We present our graph representation language and our original way to build it. Our buider allows for some dynamic control elements, e.g. arbitrary structured `if` with data dependent condition. Data dependent array indices can be incorporated in similar way.
Our PM representation can be treated as a program, equivalent to the original one, in a programming language with specific semantics. Two different semantic definitions are proposed. The ordinary one is SRE, System of Recursive Equations, and another, our original one, is DF, a dataflow semantics which is in some sense inverse to SRE. In particular, this means that our model is exact, whereas existing approaches yield generally, for programs with dynamic control elements, only approximate, or fuzzy models (in the sense that they appoint to some read not a specific write, but a set of possible writes).
As the PM carries the total semantics of the program, it can be used with various purposes for analysis, transformation, equivalence testing, etc. instead of original programs.

Keywords: polyhedral model, affine loop nests, data dependent conditionals, recurrence equations, dataflow semantics, program analysis, program transformation, equivalence testing

1 Introduction

The concept of polyhedral model (PM) appeared in the domain of automated parallelization. Most modern parallelizing compilers use the polyhedral model as an important source of information about the source program on behalf of the ability of reordering or parallelizing statements. Unfortunately, the class of

programs for which the model can be built is strongly restricted. Normally, it embraces affine loop nests with assignments in between, in which loop bounds and array element indices are affine expressions of surrounding loop variables and fixed structure parameters (array sizes etc.). `If`-statements with affine conditions are also allowed. Methods of building exact polyhedral model are well developed [3–6, 18] for this class of programs.

The polyhedral model provides statement instance-wise and array element-wise information on the dependences between all array element reads and array element writes. The compiler usually wants to know whether there is a dependence between given two statements under certain conditions. However, for the use in program parallelization, this model generally does not need to be precise: the exact information flow is irrelevant and false positive dependences are admissible.

In contrast, our aim is to totally convert the source program into the dataflow computation model such that it could be executed in a suitable machine. Thus we need the exact flow dependence information, and any kind of approximation is unacceptable. But as we know exactly all flow (true) dependences, we may ignore all other kinds of dependences, such as input, output, or anti-dependences.

When the source program is purely affine, the usual polyhedral model is exact and sufficient for our purpose. In such a model for each instance of read (load) operation there is an indication of the unique instance of write (store) operation that has written the value being read. This indication is usually represented in the form of a function (the so-called source function) that takes iteration vector of the read (and structure parameters) and produces the name and the iteration vector of a write or symbol $\perp$ indicating that such writes do not exist and the original state of memory is read.

However when the source program contains also one or several `if`-statements with non-affine (e.g. data dependent) conditions the known methods suggest only approximate model which identifies a set of possible writes for each read. Authors usually refer such models as fuzzy [6]. In some specific cases their model may provide a source function that uses as its input also values of predicates associated with non-affine conditionals in order to produce the unique source. As a rule these cases are those in which the number of such predicate values is finite (uniformly bounded).

In contrast, we built for arbitrary affine program with non-affine conditionals an exact and complete polyhedral model. Our model representation language is extended with predicate symbols corresponding to non-affine conditions of the source code. From such a model the exact source function for each read can be easily extracted. The resulting source function depends generally on iteration vector of the read and structure parameters as well as on an unlimited number of predicate values.

But the source function is not our aim. Rather it is the complete dataflow model, which can be treated, independently of the source program, as another program in a dataflow computation model. From the parallelization perspective this program carries implicitly the maximum amount of parallelism that is

reachable for the source program. A more detailed motivation and presentation of our approach can be found in [12,13].

In this paper we describe our original way of building dataflow model for affine programs and then show how it can be expanded to programs with non-affine conditionals. The affine class and the concept of affine solution tree are defined in Section 2. Sections 3–5 describe our algorithm of construction of the PM. It comprises of several passes. The first pass building effects is described in Section 3. In Section 3.1 we introduce the concept of statement effect and define the process of its construction along the AST. Simultaneously we build a resulting graph skeleton which is described in Section 3.2. Section 3.3 (together with 5.2) describes our method of dealing with non-affine conditionals. Section 4 is about the second pass building states. In Section 4.1 we introduce the concept of state and define the process of its computation along the AST. In Section 4.2 we use states to build all source functions comprising the source graph (S-graph). In Section 5 we describe the third pass in which S-graph is inverted (Section 5.1) into a use graph (U-graph) which allows for direct execution in the dataflow computation model. Section 5.2 explains additional processing required for non-affine conditionals. Several examples are presented in Section 6. Section 7 is devoted to possible applications of polyhedral model. Section 8 compares and bridges our approach and achievements with those described in the literature.

2 Some Formalism

Consider a Fortran program fragment P, subroutine for simplicity. First of all, we are interested in memory accesses that have the form of array element access and are divided into reads and writes. Usually, in an assignment statement, there are zero or several read accesses and a single write access. For simplicity and without loss of generality we allow accesses only to an individual array element, not to a whole array or subarray. Scalars are treated as 0-dimension arrays.

We define a computation graph by simply running the program P with some input data. The graph consists of two kinds of nodes: reads and writes, corresponding respectively to executions of read or write memory accesses. There is a link from a write instance w to a read instance r if r reads the value written by w. In other words, r uses the same memory cell as w and w is the last write to this cell before r.

Now that our purpose is to obtain a compact parametric description of all such graphs for a given program P, we consider a limited class of programs, for which such a description is feasible. Such programs must fit the so-called affine class, which can be formally defined by the set of constructors presented in Fig.1. The right hand side e of an assignment may contain array element access $\mathsf{A}(i_1, \ldots, i_k)$, $k \geq 0$. All index expressions as well as bounds e_1 and e_2 of `do`-loops must be affine in surrounding loop variables and structure parameters. *Affine* expressions are those built from variables and integer constants with addition, subtraction and multiplication by literal integer. Also, in an affine expression,

Λ	(empty statement)
$\mathtt{A}(i_1, \ldots, i_k) = e$	(assignment, $k \geq 0$)
$S_1; S_2$	(sequence)
`if` c `then` S_1`; else` S_2`; endif`	(conditional)
`do` $v = e_1, e_2$`;` S`; enddo`	(`do`-loop)

Fig. 1. Affine program constructors

we allow whole division by literal integer. Condition c also must be affine, i.e. equivalent to $e = 0$ or $e > 0$ where e is affine.

Programs that satisfy these limitations are often called static control programs (SCoP) [6,7]. Their computation graph depends only on symbolic parameters and does not depend on dynamic data values. Further we remove the restriction that conditional expression c must be affine. Programs of the extended class are usually called dynamic control programs (DCoP). We wont consider programs with while-loops or non-affine array indices leaving it to future investigation.

A point in the computation trace of an affine program may be identified as (s, I_s), where s is a (name of a) point in the program and I_s is the iteration vector, that is a vector of integer values of all enclosing loop variables of point s. The list of these variables will be denoted as $\boldsymbol{I}_s$, which allows to depict the point s itself as $(s, \boldsymbol{I}_s)$. (Here and below boldface symbols denote variables or list of variables as syntactic objects, while normal italic symbols denote some values as usual).

Thus, denoting an arbitrary read or write instance as (r, I_r) or (w, I_w) respectively, we represent the whole computation graph as a mapping:

$$\mathrm{F}_P : (r, I_r) \mapsto (w, I_w) \tag{1}$$

which for any read node (r, I_r) yields the write node (w, I_w) that has written the value being read, or yields $\perp$ if no such write exist and thus the original contents of the cell is read. In other words, it yields a source for each read. Thus this form of graph is called a source graph, or S-graph.

However, for translation to our dataflow computation model we need the reverse: for each write node to find all read nodes (and there may exist several or none of them) which read the very value written. So, we need the multi-valued mapping

$$\mathrm{G}_P : (w, I_w) \mapsto \{(r, I_r)\} \tag{2}$$

which for each write node (w, I_w) yields a set of all read nodes $\{(r, I_r)\}$ that read that very value written. We will refer to this form of computation graph as a use graph, or U-graph.

A subgraph of S-graph (U-graph) associated with a given read r (write w) will be referred to as an r-component (w-component).

For each program statement (or point) s we define the domain $\mathsf{Dom}(s)$ as a set of values of iteration vector I_s, such that (s, I_s) occurs in the compu-

tation. The following proposition summarizes the well-established property of affine programs [4–7, 15, 18, 23] (which is also justified by our algorithm).

Proposition 1. *For any statement* $(s, \boldsymbol{I}_s)$ *in affine program* P *its domain* $\mathsf{Dom}(s)$ *can be represented as finite disjoint union* $\bigcup_i D_i$*, such that each subdomain* D_i *can be specified as a conjunction of affine conditions of variables* $\boldsymbol{I}_s$ *and structure parameters, and, when the statement is a read* $(r, \boldsymbol{I}_r)$*, there exist such* D_i *that the mapping* F_P *on each subdomain* D_i *can be represented as either* $\perp$ *or* $(w, (e_1, \ldots, e_m))$ *for some write* w*, where each* e_i *is an affine expression of variables* $\boldsymbol{I}_r$ *and structure parameters.*

This property suggests the idea to represent each r-component of F_P as a solution tree with affine conditions at branching vertices and terms of the form $S\{e_1, \ldots, e_m\}$ or $\perp$ at leaves. A similar concept of quasi-affine solution tree, *quast*, was suggested by P. Feautrier [5].

A single-valued solution tree (S-tree) is a structure used to represent r-components of a S-graph. Its syntax is shown in Fig.2. It uses just linear expressions (L-expr) in conditions and term arguments, so a special vertex type was introduced in order to implement integer division.

S-tree	$::= \perp$	
	$\mid$ *term*	
	$\mid (L\text{-}cond \to S\text{-}tree_t : S\text{-}tree_f)$	(branching)
	$\mid (L\text{-}expr =: num\ var + var \to S\text{-}tree)$	(integer division)
term	$::= name\{L\text{-}expr_1, \ldots, L\text{-}expr_k\}$	($k \geq 0$)
var	$::= name$	
num	$::= \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid 3 \mid \ldots$	
L-cond	$::= L\text{-}expr = 0 \mid L\text{-}expr > 0$	(affine condition)
L-expr	$::= num \mid numvar + L\text{-}expr$	(affine expression)
atom	$::= \perp \mid name\{num_1, \ldots, num_k\}$	(ground term, $k \geq 0$)

Fig. 2. Syntax for single-valued solution tree

Given concrete integer values of all free variables of the S-tree it is possible to evaluate the tree with a ground term as a result value. Here are evaluation rules, which must be applied iteratively while it is possible.

A branching like $(c \to T_1 : T_2)$ evaluates to T_1 if conditional expression c evaluates to true, otherwise to T_2.

A division $(e =: m\boldsymbol{q} + \boldsymbol{r} \to T)$ introduces two new variables $(\boldsymbol{q}, \boldsymbol{r})$ that take respectively the quotient and the remainder of integer division of integer value of e by positive constant integer m. The tree evaluates as T with parameter list extended with values of these two new variables. Note that the whole tree does not depend on variables $\boldsymbol{q}$ and $\boldsymbol{r}$ because they are bound variables.

It follows from Proposition 1 that for an affine program P the r-component of the S-graph F_P for each read $(r, \boldsymbol{I}_r)$ can be represented in the form of S-tree T depending on variables $\boldsymbol{I}_r$ and structure parameters.

However the concept of S-tree is not sufficient for representing w-components of U-graph, because those must be multi-valued functions in general. So, we extend the definition of S-tree to the definition of multi-valued tree, M-tree, by two auxiliary rules shown on Fig 3.

$$
\begin{array}{lll}
M\text{-}tree ::= \ldots \text{the same as for } S\text{-tree} \ldots & & \\
\quad \mid (\&M\text{-}tree_1 \ldots M\text{-}tree_n) & \text{(finite union, } n \geq 2) & \\
\quad \mid (@\, var \rightarrow M\text{-}tree) & \text{(infinite union)} &
\end{array}
$$

Fig. 3. Syntax for multi-valued tree

The semantics also changes. The result of evaluating M-tree is a set of atoms. Symbol $\perp$ now represents the empty set, and the term $N\{\ldots\}$ represents a singleton.

To evaluate $(\&T_1, \ldots, T_n)$ one must evaluate sub-trees T_i and take the union of all results. The result of evaluating $(@\boldsymbol{v} \rightarrow T)$ is mathematically defined as the union of infinite number of results of evaluating T with each integer value v of variable $\boldsymbol{v}$. In practice the result of evaluating T is non-empty only within some bound interval of values v. In both cases the united subsets are supposed to be disjoint.

Below we present the scheme of our algorithm of building a S-graph (Sections 3 and 4) and then a U-graph (Section 5).

3 Building Statement Effect

3.1 Statement Effect and its Evaluation

Consider a program statement S, which is a part of an affine program P, and some k-dimensional array A. Let $(w\mathrm{A}, \boldsymbol{I}_{w\mathrm{A}})$ denote an arbitrary write operation on an element of array A within a certain execution of statement S, or the totality of all such operations. Suppose that the body of S depends affine-wise on free parameters $p_1, \ldots, p_l$ (in particular, they may include variables of loops surrounding S in P). We define the effect of S with respect to array A as a function

$$\mathsf{E}_{\mathrm{A}}[S] : (p_1, \ldots, p_l; q_1, \ldots, q_k) \mapsto (w\mathrm{A}, \boldsymbol{I}_{w\mathrm{A}}) + \perp$$

that, for each tuple of parameters $p_1, \ldots, p_l$ and indices $q_1, \ldots, q_k$ of an element of array `A`, yields an atom $(w\mathrm{A}, \boldsymbol{I}_{w\mathrm{A}})$ or $\perp$. The atom indicates that the write operation $(w\mathrm{A}, \boldsymbol{I}_{w\mathrm{A}})$ is the last among those that write to element $\mathrm{A}(q_1, \ldots, q_k)$ during execution of S with affine parameters $p_1, \ldots, p_l$ and $\perp$ means that there are no such operations.

The following statement is another form of Proposition 1: *the effect can be represented as an S-tree with program statement labels as term names.* We shall call them simply *effect trees.*

Building effect is the core of our approach. Using S-trees as data objects we implemented some operations on them that are used in the algorithm presented on Fig.4. A good mathematical foundation of similar operations for similar trees has been presented in [8].

The algorithm proceeds upwards along the AST from primitives like empty and assignment statements. Operation Seq computes the effect of a statement sequence from the effects of component statements. Operation Fold builds the effect of a do-loop given the effect of the loop body. For conditional statement with affine condition the effect is built just by putting the effects of branches into the new conditional node.

$\mathsf{E}_\mathrm{A}[\Lambda] = \bot$	(empty statement)
$\mathsf{E}_\mathrm{A}[S_1; S_2] = \mathsf{Seq}(\mathsf{E}_\mathrm{A}[S_1], \mathsf{E}_\mathrm{A}[S_2])$	(sequence)
$\mathsf{E}_\mathrm{A}[LA:\ \mathrm{A}(e_1, \ldots, e_k) = e] =$	(assignments to A)
$(\boldsymbol{q}_1 = e_1 \to \ldots (\boldsymbol{q}_k = e_k \to LA\{\boldsymbol{I}\} : \bot) \ldots : \bot)$	
where $\boldsymbol{I}$ is a list of all outer loop variables	
$\mathsf{E}_\mathrm{A}[LB:\ \mathrm{B}(\ldots) = e] = \bot$	(other assignments)
$\mathsf{E}_\mathrm{A}[\texttt{if}\ c\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{endif}] = (c \to \mathsf{E}_\mathrm{A}[S_1] : \mathsf{E}_\mathrm{A}[S_2])$	(conditional)
$\mathsf{E}_\mathrm{A}[\texttt{do}\ \boldsymbol{v} = e_1, e_2\texttt{;}\ S\texttt{;}\ \texttt{enddo}] = \mathsf{Fold}(\boldsymbol{v}, e1, e2, \mathsf{E}_\mathrm{A}[S])$	(do-loop)

Fig. 4. The rules for computing effect tree wrt k-dimensional array A

The implementation of function Seq is straight. To compute Seq(T_1,T_2) we simply replace all $\bot$ leaves in T_2 with a copy of T_1. The result is then be simplified by a function Prune which prunes unreachable branches by checking the feasibility of affine conjunctions (the check is known as Omega-test [18]).

The operation $\mathsf{Fold}(\boldsymbol{v}, e_1, e_2, T)$, where $\boldsymbol{v}$ is a variable and e_1 and e_2 are affine expressions, produces another S-tree T' that does not depend on $\boldsymbol{v}$ and represents the following function. Depending on all other parameters of e_1, e_2 and T we find the maximum value v of variable $\boldsymbol{v}$ in between values of e_1 and e_2, for which T evaluates to a term t (not $\bot$), and yield the term t for that value v as the result. Building this T' usually involves the solution of parametric integer programming problems (1-dimensional) and combining the results.

3.2 Graph Node Structure

In parallel with building the effect of each statement we also compose a graph skeleton, which is a set of nodes with placeholders for future links. For each assignment a separate node is created. At this stage the graph nodes are associated with AST nodes, or statements, in which they were created, for the purpose that will be explained below in Section 4. The syntax (structure) of a graph node description is presented in Fig.5.

Non-terminals ending with s usually denote an arbitrary number of its base word (a repetition), e.g. *ports* signifies *list of ports*. A node consists of a header

node ::= (*node* (*name context*)
(*dom conditions*)
(*ports ports*)
(*body computations*)
)
context ::= *names*
condition ::= *L-cond* | *TF-tree*
port ::= (*name type source*)
computation ::= (*eval name type expression destination*)
source ::= *S-tree* | IN
destination ::= *M-tree* | OUT

Fig. 5. Syntax for graph node description

with name and context, domain description, list of ports that describe inputs and a body that describes output result. The context here is just a list of loop variables surrounding the current AST node. The domain specifies a condition on these variables for which the graph node instance exists. Besides context variables it may depend on structure parameters. Ports and body describe inputs and outputs. The source in a port initially is usually an atom (or, generally, an *S*-tree) depicting an array access (array name and index expressions), which must be eventually resolved into a *S*-tree referencing other graph nodes as the sources of the value (see Section 4.2). A computation consists of a local name and type of an output value, an expression to be evaluated, and a destination placeholder $\perp$ which must be replaced eventually by a M-tree that specifies output links (see Section 5). The tag IN or OUT declares the node as input or output respectively.

Consider for example a statement `S=S+A(i)` of the summation program in Fig.8a. The initial view of the corresponding graph node is shown in Fig.6. Note that the expression in **eval** clause is built from the right hand side by replacing all occurrences of scalar or array element references with their local names (that became port names as well). A graph node for assignment usually has a single **eval** clause that represents the generator of values written by the assignment. Thereby a term of effect tree may be considered as a reference to a graph node output.

(**node** (S1 i)
(**dom** ($i \geq 1$)($i \leq n$))
(**ports** (s1 double S{}) (a1 double A{i}))
(**body** (**eval** S double (s1 + a1) $\perp$))
)

Fig. 6. An initial view of graph node for statement `S=S+A(i)`

3.3 Processing Non-affine Conditionals

When the source program contains a non-affine conditional statement S, special processing is needed. We add a new kind of condition, a predicate function call, or simply predicate, depicted as

$$name^{bool\text{-}const}\{L\text{-}exprs\} \tag{3}$$

that may be used everywhere in the graph where a normal affine expression can. It contains a name, sign T or F (affirmation or negation) and a list of affine arguments.

However, not all operations can deal with such conditions in argument trees. In particular, the Fold cannot. Thus, in order that Fold can work later we perform the elimination of predicates immediately after they appear in the effect tree of a non-affine conditional statement.

First, we drag the predicate p, which is initially on the top of the effect tree $\mathsf{E}_\mathrm{A}[S] = (p \to T_1 : T_2)$, downward to leaves. The rather straightforward process is accomplished with pruning. In the result tree, T_S, all copies of predicate p occur only in downmost positions of the form $(p \to A_1 : A_2)$, where each A_i is either term or $\bot$. We call such conditional sub-trees *atomic*. In the worst case the result tree will have a number of atomic sub-trees being a multiplied number of atoms in sub-trees T_1 and T_2.

Second, each atomic sub-tree can now be regarded as an indivisible composite value source. When one of A_i is $\bot$, this symbol depicts an implicit rewrite of an old value into the target array cell $\mathtt{A}(q_1, \ldots, q_k)$ rather than just no write. With this idea in mind we now replace each atomic sub-tree U with a new term $\boldsymbol{U}_\mathrm{new}\{i_1, \ldots, i_n\}$ where argument list is just a list of variables occurring in the sub-tree U. Simultaneously, we add the definition of $\boldsymbol{U}_\mathrm{new}$ in the form of a graph node (associated with the conditional statement S as a whole) which is shown in Fig.7. This kind of nodes will be referred to as blenders as they blend two input sources into a single one. The domain of the new node is that of statement

(**node** ($\boldsymbol{U}_\mathrm{new}$ $i_1 \ldots i_n$)
(**dom** $\mathsf{Dom}(S)$ + path-to-U-in-T_S)
(**ports** (a t ($p \to \mathsf{RW}(A_1) : \mathsf{RW}(A_2)$)))
(**body** (**eval** $a\ t\ a\ \bot$))
)

Fig. 7. Initial contents of the blender node for atomic subtree U in $\mathsf{E}_\mathrm{A}[S] = T_S$

S restricted by conditions on the path to the sub-tree U in the whole effect tree T_S. The result is defined as just copying the input value a (of type t). The most intriguing is the source tree of the sole port a. It is obtained from the atomic sub-tree $U = (p \to A_1 : A_2)$. Each A_i is replaced (by operator RW) as follows. When A_i is a term it remains unchanged. Otherwise, when A_i is $\bot$, it is replaced with

explicit reference to the array element being rewritten, $\mathtt{A}(q_1, \ldots, q_k)$. However, an issue arises: variables $q_1, \ldots, q_k$ are undefined in this context. The only variables allowed here are $i_1, \ldots, i_n$ (and fixed structure parameters). Thus we need to express indices $q_1, \ldots, q_k$ through known values $i_1, \ldots, i_n$.

To resolve this issue consider the list of (affine) conditions L on the path to the subtree U in the whole effect tree T_S as a set of equations connecting variables $q_1, \ldots, q_k$ and $i_1, \ldots, i_n$.

Proposition 2. *Conditions L specify a unique solution for values $q_1, \ldots, q_k$ depending on $i_1, \ldots, i_n$.*

Proof. Consider another branch A_j of subtree U, which must be a term. We prove a stronger statement, namely, that given exact values of all free variables occurring in A_j, $\mathsf{Vars}(A_j)$, all q-s are uniquely defined. The term A_j denotes the source for array element $\mathtt{A}(q_1, \ldots, q_k)$ within some branch of the conditional statement S. Note, however, that this concrete source is a write on a single array element only. Hence, array element indices $q_1, \ldots, q_k$ are defined uniquely by $\mathsf{Vars}(A_j)$. Now recall that all these variables are present in the list $i_1, \ldots, i_n$ (by definition of this list). □

Now that the unique solution does exist, it can be easily found by our affine machinery. See Section 5 in which the machinery used for graph inversion is described.

Thus, we obtain, for conditional statement S, the effect tree that does not contain predicate conditions. All predicates got hidden within new graph nodes. Hence we can continue the process of building effects using the same operations on trees as we did in the purely affine case. Also for each predicate condition a node must be created that evaluates the predicate value.

We shall return back to processing non-affine conditionals in Section 5.2.

4 Evaluation and Usage of States

4.1 Computing States

A state before statement $(s, \boldsymbol{I}_s)$ in affine program fragment P with respect to array element $\mathtt{A}(q_1, \ldots, q_k)$ is a function that takes as arguments the iteration vector $I_s = (i_1, \ldots, i_n)$, array indices $(q_1, \ldots, q_k)$ and values of structure parameters and yields the write (w, I_w) in the computation of P that is the last among those that write to array element $\mathtt{A}(q_1, \ldots, q_k)$ before (s, I_s).

In other words this function presents an effect of executing the program from the beginning up to the point just before (s, I_s) wrt array $\mathtt{A}$. It can be expressed as an S-tree, which may be called a *state tree* at program point before statement s for array $\mathtt{A}$.

To compute state trees for each program point we use the following method.

So far for each statement B in an affine program fragment P we have computed the S-tree $\mathsf{E}_{\mathrm{A}}[B]$ representing the effect of B wrt array $\mathtt{A}$. Now we are to

compute for each statement B the S-tree $\Sigma_{\mathrm{A}}[B]$ representing the state before B wrt array `A`.

For the starting point of program P we set

$$\Sigma_{\mathrm{A}}[P] = (\boldsymbol{q}_1 \geq l_1 \to (\boldsymbol{q}_1 \leq u_1 \to \ldots \mathrm{A\_init}\{q_1, \ldots\} \ldots : \bot) : \bot) \tag{4}$$

where term $\mathrm{A\_init}\{q_1, \ldots, q_k\}$ signifies an untouched value of array element $\mathtt{A}(q_1, \ldots, q_k)$ and l_i, u_i are lower and upper bounds of array dimensions (which are only allowed to be affine functions of fixed parameters). Thus, (4) signifies that all `A`'s elements are untouched before the whole program P.

The further computation of Σ_{A} is described by the following production rules:

1. Let $\Sigma_{\mathrm{A}}[B_1; B_2] = T$. Then $\Sigma_{\mathrm{A}}[B_1] = T$. *The state before any starting part of B is the same as that before B.*
2. Let $\Sigma_{\mathrm{A}}[B_1; B_2] = T$. Then $\Sigma_{\mathrm{A}}[B_2] = \mathsf{Seq}(T, \mathsf{E}_{\mathrm{A}}[B_1])$. *The state after the statement B_1 is that before B_1 combined by* Seq *with the effect of B_1.*
3. Let $\Sigma_{\mathrm{A}}[\mathtt{if}\ c\ \mathtt{then}\ B_1\ \mathtt{else}\ B_2\ \mathtt{endif}] = T$. Then $\Sigma_{\mathrm{A}}[B_1] = \Sigma_{\mathrm{A}}[B_2] = T$. *The state before any branch of* `if`*-statement is the same as before the whole* `if`*-statement.*
4. Let $\Sigma_{\mathrm{A}}[\mathtt{do}\ \boldsymbol{v} = e_1, e_2\mathtt{;}\ B\mathtt{;}\ \mathtt{enddo}] = T$. Then

$$\Sigma_{\mathrm{A}}[B] = \mathsf{Seq}(T, \mathsf{Fold}(\boldsymbol{v}, e_1, \boldsymbol{v}{-}1, \mathsf{E}_{\mathrm{A}}[B])) \tag{5}$$

The state before the loop body B with the current value of loop variable $\boldsymbol{v}$ is that before the loop combined by Seq *with the effect of all preceding iterations of B.*

The last form (5) needs some comments. It is the case in which the upper limit in the Fold clause depends on $\boldsymbol{v}$. To be formally correct, we must replace all other occurrences of $\boldsymbol{v}$ in the clause with a fresh variable, say $\boldsymbol{v}'$. Thus, the resulting tree will (generally) contain $\boldsymbol{v}$, as it expresses the effect of all iterations of the loop before the v-th iteration. The situation is much like that of

$$\int_0^x f(x)dx.$$

Using the rules 1-4 one can achieve (and compute the state in) any internal point of the program P (a point may be identified by a statement following it). For speed, we do not compute the state wrt array `X` at some point if there are no accesses to `X` within the current block after that point. Also, we compute only once the result of Fold with a variable as the upper limit and then use the result both for the effect of the whole loop and for the state at the beginning of the body.

The following proposition limits the usage, within a state tree T, of terms whose associated statements are enclosed in a conditional statement with non-affine condition. It will be used further in Section 5.2.

Proposition 3. *Let a conditional statement S with non-affine condition be at a loop depth m within a dynamic control program P. Consider a state tree $ST_p =$*

$\Sigma_{\mathrm{A}}[p]$ in a point p within P w.r.t. an array `A`*. Let $A\{i_1, \ldots, i_k\}$ be a term in ST_p, whose associated statement, also A, is inside a branch of S. Then the following claims are all true:*

- *$m \leq k$,*
- *p is inside the same branch of S and*
- *indices $i_1, \ldots, i_m$ are just variables of loops enclosing S.*

Proof. Let A be a term name, whose associated statement A is inside a branch b of a conditional statement S with non-affine condition. It is either assignment to an array, say `A`, or a blender node emerged from some inner conditional (performing a "conditional assignment" to `A`). From our way of hiding predicate conditions described in Section 3.3 it follows that the effect tree of S, $\mathsf{E}_{\mathrm{A}}[S]$, as well as of any other statement containing S, will not contain a term with name A. Hence, due to our way of building states from effects described above, this is also true for the state tree of any point outside S, including the state ST_S before the S itself. Now, consider the state ST_p of a point p within a branch b_1 of S. (Below we'll see that $b_1 = b$). We have

$$ST_p = \mathsf{Seq}(ST_S, ST_{S-p}), \tag{6}$$

where ST_{S-p} is the effect of executing the code from the beginning of the branch b_1 to p (recall that the state before the branch b_1, ST_{b_1}, is the same as ST_S according to Rule 3 above). Consider a term $A\{i_1, \ldots, i_k\}$ in ST_p. As it is not from ST_S, it must be in ST_{S-p}. Obviously, ST_{S-p} contains only terms associated with statements of the same branch with p. Thus, $b_1 = b$. And these terms are only such that their initial m indices are just variables of m loops surrounding S. Thus, given that the operation Seq does not change term indices, we have the conclusion of Proposition 3. □

4.2 Resolving Array Accesses

Now we shall use states before each statement to accomplish building the source graph F_P. Consider a graph node example shown on Fig.6. Initially, source trees in **ports** clause contain terms denoting references to array element, like $\mathtt{A}\{\mathtt{e_1}, \ldots, \mathtt{e_k}\}$. We want to resolve each such array access term into subtree with only normal source terms. Recall that each graph node is associated with a certain point p in the AST (that is, a statement, or the program point before the statement) and that we already have a state $\Sigma_{\mathrm{A}}[p]$. Now we apply $\Sigma_{\mathrm{A}}[p]$ as a function to indices $(e_1, \ldots, e_k)$ and use the result as a replacement for the term $\mathtt{A}\{\mathtt{e_1}, \ldots, \mathtt{e_k}\}$. As we do the application symbolically (just as a substitution with subsequent pruning), the result will be a S-tree. Having expanded each such array access term we get the port source tree in which all terms refer only to other graph nodes. And the set of all port source trees in all graph nodes comprises the source graph F_P.

Recall that each graph node X has a domain $\mathsf{Dom}(X)$ which is a set of possible context vectors. It is specified by a list of conditions, which are collected

from surrounding loop bounds and `if` conditions. The list may contain also predicate conditions. We write $D \Rightarrow p$ to indicate that the condition p is valid in D (or follows from D). In case of a predicate condition $p = \boldsymbol{p}^b\{e_1, \ldots, e_k\}$ it signifies that the list D just contains p (or contains some other predicate $p = \boldsymbol{p}^b\{f_1, \ldots, f_k\}$ such that $D \Rightarrow (e_i = f_i)$ for all $i = 1, \ldots, k$). For a S-graph built so far the following proposition limits the usage of atoms $A\{\ldots\}$ for which $\mathsf{Dom}(A)$ has predicate condition.

Proposition 4. *Suppose that B is a regular node (not a blender) whose source tree T contains a term $A\{i_1, \ldots, i_k\}$ (corresponding to an assignment to an array* `A`*). Let $\mathsf{Dom}(A) \Rightarrow p$, where $p = \boldsymbol{p}^b\{j_1, \ldots, j_m\}$ is a predicate condition. Then:*

- $m \le k$,
- $j_1 = i_1, \ldots, j_m = i_m$, *and all these are just variables of loops enclosing the conditional statement with predicate p,*
- $\mathsf{Dom}(B) \Rightarrow p$.

Proof. As $\mathsf{Dom}(A) \Rightarrow \boldsymbol{p}^b\{j_1, \ldots, j_m\}$, the predicate p denotes the condition of a conditional statement S enclosed by m loops with variables $j_1, \ldots, j_m$, and this S contains the statement A in the branch b (by construction of Dom). The source tree T was obtained by a substitution into the state tree before B, $ST_B = \Sigma_{\mathrm{A}}[B]$, which must contain a term $A\{i'_1, \ldots, i'_k\}$. It follows, by Proposition 3, that statement B is inside the same branch b (hence, $\mathsf{Dom}(B) \Rightarrow p$), $m \le k$ and $i'_1, \ldots, i'_m$ are just variables $j_1, \ldots, j_m$. However the substitution replaces only formal array indices and does not touches enclosing loop variables, here $j_1, \ldots, j_m$. Hence $i'_1 = i_1, \ldots, i'_m = i_m$. □

When B is a blender the assertion of the Proposition 4 is also valid but $\mathsf{Dom}(B)$ should be extended with conditions on the path from the root of the source tree to the term $A\{\ldots\}$. The details are left to the reader.

5 Building the Dataflow Model

5.1 Building *U*-graph by Inverting *S*-graph: Affine Case

In dataflow computation model the data flow from source nodes to use nodes. Thus, the generating node must know exactly which other nodes (and by which port) need the generated value and sends the data element to all such node-port pairs. This information, known also as use graph G_P, is to be represented in the form of destination M-trees in the eval clauses of graph nodes. Initially they are all set to $\perp$ as just placeholders.

Suppose the source program fragment P is purely affine. Having the source graph F_P in the form of affine S-trees in node ports, it is not difficult to produce the inversion resulting in M-trees.

The graph is inverted path-wise: first, we split each tree into paths. Each path starts with header term $R\{i_1, \ldots, i_n\}$, ends with term $W\{e_1, \ldots, e_m\}$ and

has a list of affine conditions extended with quotient/remainder definitions like ($e =: k\boldsymbol{q}+\boldsymbol{r}$) in between ($k$ is a literal integer here). Only variables $i_1, \ldots, i_n$ and structure parameters can be used in affine expressions $e, ei, \ldots$. New variables $\boldsymbol{q}$ and $\boldsymbol{r}$ may be used only to the right of their definition. The InversePath operation produces the inverted path that starts with header term $W\{j_1, \ldots, j_m\}$ with new formal variables $j_1, \ldots, j_m$, ends with term $R\{f_1, \ldots, f_n\}$ and has a list of affine conditions and divisions in between. Also, universally quantified variables can be introduced by clause ($@\boldsymbol{v}$). All affine expressions f_i are built with variables $j_1, \ldots, j_m$, structure parameters and $\boldsymbol{q}/\boldsymbol{r}/@\boldsymbol{v}$-variables defined earlier in the list. The inversion involves solving the system of linear Diophantine equations. In essence, it can be viewed as a projection or variable elimination process. When a variable cannot be eliminated it is simply introduced with @-clause.

In general, one or several paths can be produced. All produced paths are grouped by new headers, each group being an M-tree for respective graph node, in the form (& T_1 $T_2 \ldots$) where each T_i is a 1-path tree. Further, the M-tree can be simplified by the operation SimplifyTree. This operation also involves finding bounds for @-variables, which are then included into @-vertices in the form:

$$(@\boldsymbol{v}(l_1 u_1)(l_2 u_2) \ldots T)$$

where l_i, u_i are affine lower and upper bounds of i-th interval, and v must belong to one of the intervals.

5.2 Inverting *S*-graph for Programs with Non-affine Conditionals

When program P has non-affine conditionals the above inversion process will probably yield some M-trees with predicate conditions. Hence, a node with such M-tree needs the value of the predicate as its input. However, this value may not necessarily be needed always, and thus it may induce redundant dependences. So, when a predicate vertex in M-tree does not dominate all term leaves, we should cut the vertex off and create another node with the sub-tree as a destination tree. Otherwise, we just add to the node a Boolean port connected to a predicate evaluating node. We must do so repetitively until all predicates in M-trees refer to Boolean-valued ports.

In either case some nodes need an additional port for the value of predicate. We call such nodes *filters*. In the simplest case a filter has just two ports, one for the main value and one for the value of the predicate, and sends the main value to the destination when the predicate value is true (or false) and does nothing otherwise.

Generally, the domain of each token and each node may have several functional predicates in the condition list. Normally, a token has the same list of predicates as its source and target nodes. However, sometimes these lists may differ by one item. Namely, a filter node generates tokens with a longer predicate list whereas the blender node makes the predicate list one item shorter compared to that of incoming token. In the examples below arrows are green (dotted) or red (dashed) depending on the predicate value.

However, our aim is to produce not only U-graph, but both S-graph and U-graph which must be both complete and mutually inverse. To simplify our task we update the S-graph before inversion such that inversion does not produce predicates in M-trees. To achieve this we check for each port whether its source node has enough predicates in its domain condition list. When we see the difference, namely that the source node has less predicates, then we insert a filter node before that port. And the opposite case, that the source has more predicates, is impossible, as it follows immediately from Proposition 4.

6 Examples

A set of simple examples of a source program (subroutine) with the two resulting graphs S-graph and U-graph are shown in Figs. 8,9,11. All graphs has been built automatically in textual form and then redrawn manually in graphical view. Nodes are rectangles or other shapes and data dependences are arrows between them. Usually a node has several input and one output ports. A port is usually identified as a point on node boundary. The domain is usually shown once for a group of nodes with the same domain (at the up side in curly braces). Those groups are separated by a dotted line. Each node should be considered as a collection of instance nodes of the same type that differ in domain parameters from each other. Arrows between nodes may fork depending on some condition (usually it is affine condition of domain parameters), which is then written near the start of the arrow immediately after the fork. When arrow enters a node it carries a new context (if it has changed) written there in curly braces. The simplest and purely affine example in Fig.8 explains the notations. Arrows in the S-graph are directed from a node port to its source (node output). The S-graph

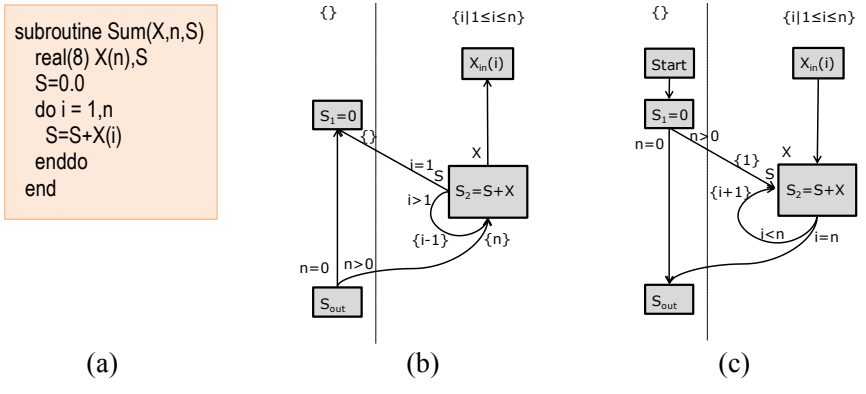


Fig. 8. Fortran program Sum (a), its S-Graph (b) and U-graph (c)

arrows can be interpreted as the flow of requests for input values. More exact semantics will be described in Section 7.2.

In the U-graph, vice versa, arrows are directed from node output to some node port. In contrast with the S-graph, they denote actual flow of data. U-graph execution obeys semantic of dataflow computation model described in Section 7.3.

In the U-graph there also appears the need to get rid of zero-port nodes which arise from assignments with no one read operation. We simply insert into such nodes a dummy port which receives a dummy value. It looks as if we use, in the right hand side of the assignment, a dummy scalar variable that is set at the start of the program P. Thus a node `Start`, which generates a token for node S1 (corresponding to the assignment `S=0.0`), appeared in the U-graph of our example.

A simplest example with non-affine conditions is shown on Fig.9. The textual view of graphs was generated automatically, whereas the graphical view was drawn by hand.

When a source program contains a non-affine conditional, in the S-graph there appears a new kind of node, the blender, depicted as a blue truncated triangle (see Fig. 9b). Formally, it has a single port, which receives data from two different sources depending on the value of the predicate. Thus, it has another implicit port for Boolean value (on top). The main port arrows go out from side edges; true and false arrows are dotted green and dashed red respectively. The S-graph semantics of the blender is:

1. Invoke the predicate and wait for the result.
2. Depending on the result execute true (green) or false (red) branch.

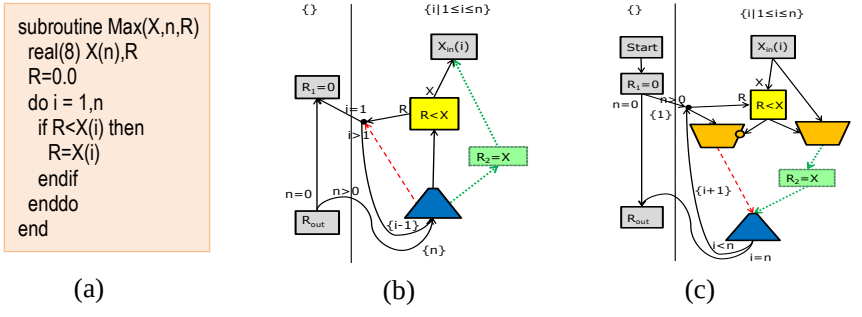


Fig. 9. Fortran program Max (a), its S-Graph (b) and U-graph (c)

In the U-graph the blender does not need a condition: in either case it receives a value token on its unique port without knowing which node has sent it and under which condition. However, when the source itself is not under the needed

condition, a filter node must be inserted in between the source node and the receiver port (it is shown in Fig.8c as an inverted yellow trapezoid). The predicate value coming into a side edge and a circle at the entry point indicate that the main value is passed when the condition is false.

The textual view of the blender from Fig.9 is shown in Fig.10. Note the predicate FP1{i} on top of the S-tree of the unique port R. The S-tree contains a reference to BR{i-1} under conjunction $(\mathrm{FP1}^{\mathrm{F}}\{\mathrm{i}\})(i > 1)$. The backward data arrow of the U-graph (in the M-tree of **eval**clause) goes through the filter node FR1.R{i+1}.

(**node** (BR i)
  (**dom** (1 ≤ i)(i ≤ n))
  (**ports** (R double (FP1{i} → R2{i} : (i = 1 → R1{} : BR{i − 1}))))
  (**body** (**eval** R double R(i = n → R_out{} : (&P1.R{i} FR1.R{i + 1})))

Fig. 10. The blender from example on Fig.9

A more complex and interesting example, a bubble sort program and its graphs, is shown on Fig.11. In contrast with previous ones, the U-graph exhibits high parallelism: its parallel time is $2n$ instead of $n(n + 1)/2$ for sequential execution.

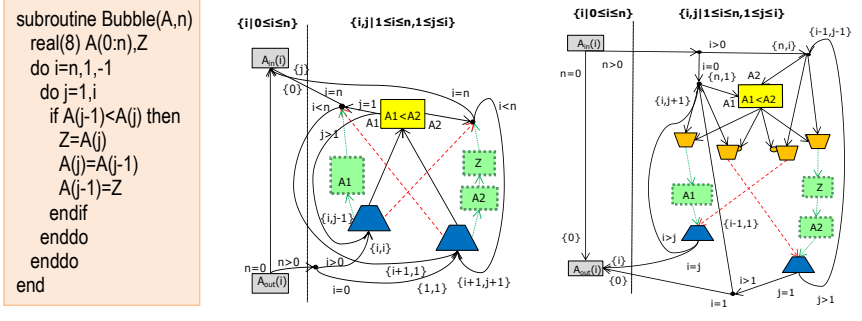


Fig. 11. Fortran program Bubble (a), its S-Graph (b) and U-graph (c)

7 Properties and Usage of Polyhedral Model

7.1 General Form of Dataflow Graph

We start from the purely static control graph, which is a set of nodes with syntax shown in Fig. 5. In this form both the S-graph and the U-graph are presented. Ignoring destination trees in **eval** clauses we get the S-graph, ignoring source trees in ports we get the U-graph. Both graphs represent the same dependence relation. It means that the respective set of trees must be mutually inverse. Recall that source trees representing the S-graph are S-trees (single-values), while destination trees forming the U-graph are M-trees (multi-valued).

Then we introduce a special kind of node called a *predicate* which produces a Boolean condition. This value is used directly by source tree of a *blender*, which is an identity node with a single port with the source tree of the form $(p \to T_1 : T_2)$, where predicate condition p has the form (3).

As we avoid predicate conditions appearing in destination trees, we introduce *filter* nodes, which are in some sense inverse to blenders. Conceptually, *filter* is an identity node with the usual input port and the destination tree of the form $(p \to T_{out} : \bot)$. But instead of predicate condition p of the form $\boldsymbol{p}^b\{e_1, \ldots, e_k\}$, we add a port named p with atom $P\{e_1, \ldots, e_k\}$ as a source tree and either $(p \to T_{out} : \bot)$ or $(p \to \bot : T_{out})$ as a destination tree. Thus filter is used as a gate which is open or closed depending on the value on port p: the gate is open, if the value is b, otherwise closed. Note that filters are necessary in U-graph, but not in S-graph.

The S-graph must satisfy the two following constraints. The first is a consistency restriction. Consider a node $\mathrm{X}\{\boldsymbol{I}\}$ with domain D_X and a source tree T. Let $I \in D_\mathrm{X}$. Then $T(I)$ is some atom $\mathrm{Y}\{J\}$ such that $J \in D_\mathrm{Y}$. The second constraint requires that the S-graph must be *well-founded*, which means that no one object node $\mathrm{X}\{I\}$ may transitively depend on itself.

7.2 Using the *S*-graph as a Program

The S-graph can be used to evaluate output values given all input values. Also, all structure parameters must be known. We assume that each node produces a single output value (otherwise atom names in source trees should refer to a node-output pair rather than just a node).

Following [6] we transform the S-graph into a System of Recurence Equations (SRE), which can be treated as a recursive functional program. Each node of S-graph is presented as definition of recursive function whose arguments are context variables. Its right hand side is composed of a body expression with ports as calls to additional functions, whose right hand sides in turn are obtained from their source trees with atoms and predicates as function calls. Input nodes are functions defined elsewhere. In Fig.12 is presented a simplified SRE for the S-graph from Fig.9b. Execution starts with invocation of the output node function. Evaluation step is to evaluate the right hand side calling other

$$
\begin{aligned}
&\mathrm{P(i)} = \mathrm{R(i)} < \mathrm{X(i)}\\
&\mathrm{B(i)} = \text{if P(i) then X(i) else R(i)}\\
&\mathrm{R(i)} = \text{if i} = 1 \text{ then R1() else if i} > 1 \text{ then B(i} - 1\text{) else } \bot\\
&\mathrm{R1()} = 0\\
&\mathrm{R_{out}} = \text{if n} = 0 \text{ then R1() else if N} > 0 \text{ then B(n) else } \bot
\end{aligned}
$$

Fig. 12. System of Recurrent Equations equivalent to S-graph on Fig.9b

invocations recursively. For efficiency it is worth doing tabulation so that neither function call is executed twice for the same argument list.

Note, that both the consistency and the well-foundedness conditions together provide the termination property of the S-graph program.

7.3 Computing the U-graph in the Dataflow Computation Model

The U-graph can be executed as program in the dataflow computation model. A node instance with concrete context values *fires* when all its ports get data element in the form of data token. Each fired instance is executed by computing all its **eval** clauses sequentially. All port and context values are used as data parameters in the execution. In each **eval** clause the expression is evaluated, the obtained value is assigned to a local variable and then sent out according to the destination M-tree. The tree is executed in an obvious way. In the conditional vertex, the left or right subtree is executed depending on the Boolean value of the condition. In &-vertices, all sub-trees are executed one after another. An -vertex acts as a `do`-loop with specified bounds. Each term of the form $R.x\{f_1, \ldots, f_n\}$ acts as a token send statement, that sends the computed value to the graph node R to port x with the context built of values of f_i. The process stops when all output nodes get the token or when all activity stops (quiescence condition). To initiate the process, tokens to all necessary input nodes should be sent from outside.

7.4 Extracting Source Functions from S-graph

There are two ways to extract the source function from the S-graph. First, we may use the S-graph itself as a program that computes the source for a given read when the iteration vector of the read as well as values of all predicates are available. We take the SRE and start evaluating the term $R(i_1, \ldots, i_n)$, where $i_1, \ldots, i_n$ are known integers, and stop as soon as some term of the form $W(j_1, \ldots, j_m)$ is encountered (where W is a node name corresponding to a write operation and $j_1, \ldots, j_m$ are some integers).

Also, there is a possibility to extract the general definition of the source function for a given read in a program. We may do it knowing nothing about predicate values. We start from the term $R\{\boldsymbol{i}_1, \ldots, \boldsymbol{i}_n\}$ where $\boldsymbol{i}_1, \ldots, \boldsymbol{i}_n$ are symbolic variables and proceed unfolding the S-graph symbolically into just the S-tree. Having encountered the predicate node we insert the branching with symbolic

predicate condition (without expanding it further). Having encountered a term $W\{e_1, \ldots, e_m\}$ we stop unfolding the branch. Proceeding this way we will generate a possibly infinite S-tree representing the source function in question. If were lucky the S-tree will be finite. It seems that, in previous works on building polyhedral models for programs with non-affine `if`-s [6, 7], the exact result is produced only when the above process stops with a finite S-tree as a result.

But we can also produce a good result even when the generated S-tree is infinite (note, that this is the case in examples `Max` and `Bubble`). Having encountered a node already visited we generalize, i.e. cut-off the earlier generated sub-tree from that node replacing it with invocation of another function and schedule the generation of a new function starting from that node. The process converges to a set of mutually recursive function definitions that implements the source function in the most unalloyed form.

The process we have just described is a particular case of the well known *supercompilation* [10,17,19,21]. In a more general setting this concept can provide a very elaborate and productive tool for transforming programs represented in the dataflow (or polyhedral) model. An interesting open issue is to invent a good *whistle* and *generalization strategy* for a supercompiler that deals with polyhedral *configurations*.

7.5 Analysis

The polyhedral model may be used for various purposes. Many useful properties of original programs can be detected. Here are some examples: array bounds checks, dead/unused code detection and feasible parallelism. Various questions can be studied by means of abstract interpretation of the polyhedral model instead of the original program.

7.6 Transformations

Initially our compiler generates a very fine grained graph in which a node corresponds to a single assignment, or condition expression, or it is a blender or a filter. Thus, it is a good idea to apply a *coagulation* transformation, that takes, say, a `Bubble` program U-graph shown in Fig.11(c), and produces a more coarse grained data flow graph as shown in Fig.13(b). Within coagulation, several small nodes are combined into a large single node. The body of a resulting node is a fusion of bodies of original nodes. Data transfer between original small nodes transforms into just variable *def* and *use* within the body of the resulting node. This code can be efficiently evaluated on a special multiprocessor system.

In Fig.13(c) is shown the computation graph for $n = 4$. Each node instance is marked with its context values. Here one can see clearly the possible parallelism.

A form of coagulation is vectorization. It involves gluing together several nodes of the same type. This transformation is the analogue of tiling for affine loop programs.

An inverse to coagulation, *atomization*, can also be useful. For example, it is beneficial before testing equivalence.

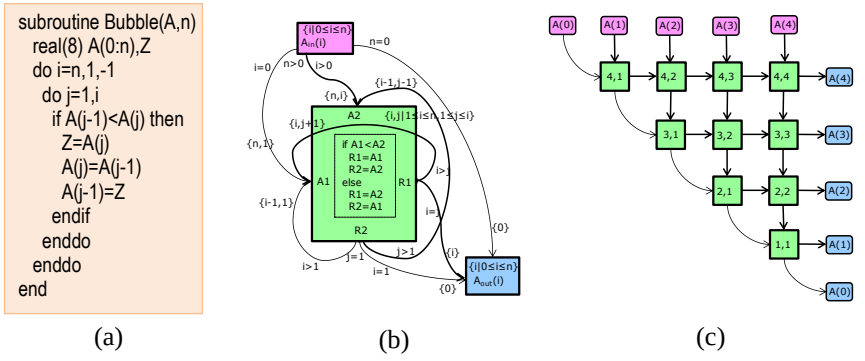


Fig. 13. Fortran program Bubble (a), its coagulated U-Graph (b) and the graph expansion for $n = 4$ (c)

7.7 Equivalence Testing

The S-graph form can be used for testing two affine programs for equivalence. Consider, for example, another version of bubble sort program, `Bubble2`, shown on Fig.14(a), its coagulated U-graph (b) and respective computation graph for $n = 4$ (c). It is easy to see that the computation graphs of both programs `Bubble` and `Bubble2` are essentially the same: they differ only in the way of numbering the nodes. To prove it one needs to find the affine mapping of contexts that would make the graphs equal.

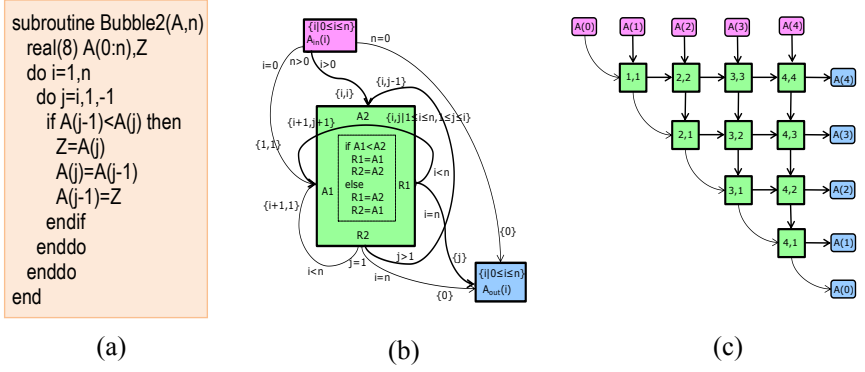


Fig. 14. Fortran program Bubble2 (a), its coagulated U-Graph (b) and the graph expansion for $n = 4$ (c)

Generally we need a regular procedure to establish equivalence of two polyhedral graphs. A good relevant procedure is presented in [22]. It allows for graphs which have a generalized form of static control dependence graph with affine dependences. Graph vertices are adorned by abstract operation symbols with single output. Thus, strictly speaking within their approach our example is not tractable: if we use coagulated form (as in Figs. 13-14) then the node function with two outputs R1 and R2 does not meet the requirement of a single output, and if we consider atomistic form (as in Fig.11) then the property of static control does not hold. Perhaps the first problem is just technical and the approach can be easily generalized. However, data dependent conditions generally cannot be easily eliminated. So, we want to generalize the approach of [22] to admit some restricted dynamic control.

Since our work is not yet finished, we just describe here the variety of dynamic control graphs which we want to be allowed (Section 7.1 above).

In our terms, the equivalence testing procedure deals with a pair of S-graphs. Following [22], it starts from output nodes denoting values of output arrays, and tries to prove that the respective values are computed by essentially the same function compositions from values of input nodes. When a predicate condition, p, is encountered in a source tree, the predicate value is 'requested' and the result is used symbolically for the selection of the source. Thus, a split appears in the proof three due to the unknown predicate value. Now we expand the equivalence testing procedure from [22], so as to correctly deal with such splits.

8 Related Work

In this Section we compare our approach with other attempts of building polyhedral models for affine programs with non-affine conditionals.

The foundations of dependence (data flow) analysis for arrays have been well established in the 90-s by Feautrier [4–6], Pugh [18], Collard and Griebl [3, 7], Maslov [15] and others [9, 16]. Their methods use the Omega test and Integer Programming libraries and yield an exact solution for dependence between any pair of read and write references in affine program. Thus in the pure affine case our work adds almost nothing more (except that we use the resulting polyhedral model further to produce a program in the dataflow computation model). However in the case of affine programs extended with non-affine conditions (the so-called dynamic control programs), the state-of-the-art is to yield in the general case a fuzzy solution [6]. It is fuzzy in the sense that the source function produces a set of possible sources, not the unique and exact source. The authors claim that it is the best that can be done. But it seems that the claim proceeds from assumption that the result should be represented in the form of the finite quasi-affine solution tree (quast). And as we have seen in Section 7.4, generally the source function can be represented as a finite or infinite quast, but it can always be represented as a finite S-graph.

Our base affine machinery of building the exact S-graph also differs. Whereas it is a common practice to build the polyhedral model by considering each read-

write pair independently, our method of building a dataflow model first produces effects and then states using only writes, and then resolves all reads against states. It is interesting to notice a similarity between our effect/state building process and the process of backward traversing the control flow graph presented in [3,7] which fail, in general, to produce exact (not fuzzy) results. Both processes are moving along the same path but in opposite directions. The authors usually argue for moving backward noticing that the process can stop when the total source is found (cf. also [15]). It is a good idea, and it can be incorporated into our algorithm simply by porting it to a lazy language, e.g. to Haskell, or by somehow emulating the laziness. In the lazy setting, the tree T will not be built at all in applications like $\mathsf{Seq}(T, t)$, where t is a term (or a $\perp$-less tree).

Speaking of parallelization, one must not forget about the distribution of computations in space and time. On this subject, there are many works in which an optimal (in terms of communications volume and load balancing) mapping in multidimensional space and time is sought, and then on its basis an inverse translation into a loop nest program with parallel loops is made [1, 7]. In our dataflow computational model the knowledge of the distribution functions, although not mandatory, can significantly improve the efficiency of execution. The project of a real multiprocessor that can directly execute the dataflow model is being developed in our institute IDPM RAS [2, 14, 20].

9 Conclusion

Our aim was to build the converter of a program P that belongs to a specific class into the dataflow computation model. Thus we need not only to build the exact and complete data flow model (which is usually referred to as the polyhedral model and comprises of exact source functions for each read operation in the program P), but also to invert it and thus obtain the exact use function for each write operation. The latter representation can be used as an equivalent program in a specific dataflow computation model, in which the maximum parallelism inherent to program P is exhibited. At the present time, the described machinery is implemented in a prototype translator which is written totally in the functional language Refal, version 6 [11]. Currently, it admits as input an arbitrary affine program extended with non-affine conditions in `if`-statements (provided that unlimited computing resources are available).

However, the intermediate source graph also appears interesting. It can also be treated as an independent semantic representation of the input program, namely, the SRE. Partially evaluating the SRE one can use it as an exact source function definition, that is evaluate the write statement of input program that wrote the value being read by the given read statement. Or one can produce a more refined form of the source function for a given read operation. Note that all this is possible for arbitrary affine programs with non-affine conditionals.

The work was supported by Russian Academy of Sciences Presidium Program for Fundamental Research "Fundamental Problems of System Programming" in 2009–2013.

References

1. Uday Bondhugula, Muthu Manikandan Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In Laurie J. Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 132–146. Springer, 2008.
2. V.S. Burtsev. "Vybor novoj sistemy organizacii vypolneniya vysokoparallelnyh vychislitelnyh processov, primery vozmozhnyh arhitekturnyh reshenij postroeniya superEVM" (The choice of a new organization system of execution of highly-parallel computation processes and examples of possible supercomputer architecture solutions). In V.S. Burtsev, editor, *Parallelizm vychislitelnyh processov i razvitie arhitektury superEVM*, pages 41–78. IVVS RAS, Moscow, 1997.
3. Jean-Francois Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 286–302, London, UK, UK, 2000. Springer-Verlag.
4. Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, 1988.
5. Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
6. Paul Feautrier. Array dataflow analysis. In Santosh Pande and Dharma P. Agrawal, editors, *Compiler Optimizations for Scalable Parallel Systems*, pages 173–219. Springer-Verlag New York, Inc., New York, NY, USA, 2001.
7. Martin Griebl. *Automatic parallelization of loop programs for distributed memory architectures.* Habilitation thesis, Department of Informatics and Mathematics, University of Passau, 2004.
8. S.A. Guda. Operations on the tree representations of piecewise quasi-affine functions. *"Informatika i ee primeneniya" (Informatics and its applications)*, 7(1):58–69, 2013.
9. Gautam Gupta and Sanjay V. Rajopadhye. The Z-polyhedral model. In Katherine A. Yelick and John M. Mellor-Crummey, editors, *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 237–248. ACM, 2007.
10. Andrei V. Klimov. An approach to supercompilation for object-oriented languages: the Java Supercompiler case study. In *First International Workshop on Metacomputation in Russia, Proceedings. Pereslavl-Zalessky, Russia, July 2–5, 2008*, pages 43–53. Pereslavl-Zalessky: Ailamazyan University of Pereslavl, 2008.
11. Ark.V Klimov. Refal-6. URL: http://refal.net/ arklimov/refal6/index.html, 2004.
12. Ark.V. Klimov. The use of selection trees for describing states in parallelizing compiler. In *Proceedings of All-Russian Scientific Conference Scientific service in Internet*, pages 238–240. Moscow, MSU Press, 2009. URL: http://agora.guru.ru/abrau2009/pdf/238_NSSI_2009_Abrau-2009.pdf.
13. Ark.V. Klimov. Transforming affine nested loop programs to dataflow computation model. In *Ershov Informatics Conference, PSI Series, 8-th edition, Preliminary Proceedings, June, 27 July, 1*, pages 274–285, Akademgorodok, Novosibirsk, Russia, 2011.
14. Ark.V. Klimov, N.N. Levchenko, S.A. Okunev, and A.L. Stempkovsky. Supercomputers, memory hierarchy and dataflow computation model. *Program systems: theory and applications*, 5(1):15–36, 2014.

15. Vadim Maslov. Lazy array data-flow dependence analysis. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 311–325. ACM Press, 1994.
16. Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 1–14. ACM, 1991.
17. Andrei P. Nemytykh, Victoria Pinchik, and Valentin Turchin. A self-applicable supercompiler. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation. Dagstuhl Castle, Germany, February 1996*, volume 1110 of *Lecture Notes in Computer Science*, pages 233–252. Springer-Verlag, 1996.
18. William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In Joanne L. Martin, editor, *SC*, pages 4–13. IEEE Computer Society / ACM, 1991.
19. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
20. A.L. Stempkovsky, N.N. Levchenko, S.A. Okunev, and V.V. Tsvetkov. Parallel dataflow computing system the further development of architecture and the structural organization of the computing system with automatic distribution of resources. *Informatsionnye tekhnologii*, (10):2–7, 2008.
21. Valentin F. Turchin. Program transformation by supercompilation. In Harald Ganzinger and Neil D. Jones, editors, *Programs as Data Objects*, volume 217 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 1985.
22. Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 599–613. Springer, 2009.
23. V.V. Voevodin and Vl.V. Voevodin. *"Parallel'nyje vychislenija" (Parallel computations)*. BKhV-Peterburg, St. Petersburg, 2004.

Nullness Analysis of Java Bytecode via Supercompilation over Abstract Values*

Ilya G. Klyuchnikov

JetBrains; Keldysh Institute of Applied Mathematics of RAS

Abstract. Code inspections in the upcoming release of IntelliJ IDEA take into account how binary Java libraries used in a project deal with null references. For this purpose Java libraries are annotated with results of nullness analysis under the hood. The paper reveals one of several non-trivial technical aspects of nullness analysis of Java binaries performed by IDEA: supercompilation over abstract values. A case study project *Kanva-micro* – a tool for inference of `@NotNull` method parameters – is used to illustrate this aspect step-by-step. A method parameter is annotated by *Kanva-micro* as `@NotNull` if the method cannot complete normally when `null` is passed to this parameter. The source code of *Kanva-micro*'s core is provided and explained in details. The paper may also serve as a tutorial on using supercompilation methods for program analysis.

1 Introduction

This paper starts a series of tutorial papers explaining details of how nullness analysis of Java bytecode is implemented in the upcoming IntelliJ IDEA 14 release. The papers are organized around two self-sufficient ready-to-run tutorial projects:

- The *Kanva-micro* project [8] focuses on the essence of the method (supercompilation over abstract values) at the cost of simplifications.
- The *Faba* project [4] is about how this method may be implemented in a production system.

The current paper describes the *Kanva-micro* project step-by-step.

1.1 Nulls in Java, richer type systems and the problem of interoperability

The majority of mainstream programming languages (including Java programming language) allow null references (`null` in Java). Dereference of `null` results into a runtime error. Tony Hoare, the creator of `null`, has stated that the `null` was "the billion dollar mistake" in the language design.

* Supported by Russian Foundation for Basic Research grant No. 12-01-00972-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

Ironically, compile-time checks performed by compilers of statically typed languages are to guarantee the absence of runtime errors if code compiles, and null pointer errors are not covered by these checks in major mainstream languages.

Dereference of `null` results into `NullPoiterException` in Java.

There are practical alternatives to enjoy null safety when programming for JVM:

- Migrate to alternative JVM language with nullable types such as Kotlin [9] or Ceylon [2].
- Enrich Java code with nullity annotations like `@NotNull` and `@Nullable` and use additional tools to check such annotations. There are several tools understanding nullity annotations: Eclipse Java compiler with additional null analysis [3], null analysis inspections in IntelliJ IDEA [6], the Checker framework [10].

Anyway, a Java programmer enjoys null safety only when working in a new richer type system. When there is a need to use existing Java libraries, the problem of `null` references appears on the boundary of two worlds. There is no clear practical way to make use of Java libraries *completely* null safe. However, it is possible to infer *some* nullity information automatically to make use of Java libraries safer.

1.2 The task

Some null analysis tools, including IntelliJ IDEA and the Checker framework, allow to use external annotations that are stored separately from library bytecode. The Kotlin compiler allows to specify external annotations as well. So inference of such annotations is of practical usage.

Kanva-micro [8] focuses on inference of `@NotNull` annotations for method parameters. A method parameter is annotated as `@NotNull` if *in any situation* when `null` is passed to this parameter the method cannot complete normally. Practically, it means that an author of this library method doesn't expect `null` to be passed to this parameter. From a client's point of view this is the same as the author put explicit `@NotNull` annotation in original source code.

The task of *Kanva-micro* is to automatically infer such external annotations for Java libraries. The phrase "*in any situation*" in the previous paragraph is significant: inferred annotations should not forbid to use a library. A `@NotNull` annotation is considered incorrect if under *some* conditions when `null` is passed to this parameter, the method completes normally.

There are no problems with the following `@NotNull` annotation: all possible executions result in `NullPointerException` when `view` is `null`.

```
1  void loadConfig(@NotNull View view) {
2    File f = getConfigFile();
3    if (f != null) {
4        view.loadConfigFromFile(f);
5    } else {
```

```
6         view.loadDefaultConfig();
7     }
8   }
```

However, the next `@NotNull` annotation is incorrect because this method completes normally when `getConfigFile()` returns `null` and `view` is `null`.

```
1   void saveConfig(@NotNull View view) {
2       File f = getConfigFile();
3       if (f != null) {
4           view.saveConfigToFile(f);
5       }
6   }
```

1.3 Handling of asserts

Many real-world Java libraries check that parameter is not `null` in the following way:

```
1   if (array == null) {
2       throw new IllegalArgumentException("array␣is␣null");
3   }
```

Sometimes such checks may be implicit:

```
1   if (!(object instanceof Serializable)) {
2       throw new IllegalArgumentException("object␣is␣not␣Serializable");
3   }
```

So to handle all these checks as well and infer `@NotNull` for corresponding parameters is desirable.

1.4 Inference of a subset of `@NotNull` annotations

The task to infer *all* correct `@NotNull` annotations in general case is undecidable. The root cause of undecidability of this task is undecidability of the subtask to detect whether a certain execution path in a program is reachable.

So, the practical task is to infer as much correct `@NotNull` annotations for method parameters as possible.

Kanva-micro assumes that all branches of conditionals that do not directly depend on nullity of parameters are reachable. Under this assumption annotations inferred by *Kanva-micro* are sound (since a superset of all possible execution paths is considered). But some annotations are lost, so *Kanva-micro* infers a subset of all correct annotations.

1.5 Algorithm presentation

Kanva-micro relies heavily on ASM library which is "an all purpose Java bytecode manipulation and analysis framework" [1, 12]. ASM library is de-facto a standard tool for Java bytecode processing in many production projects. The main part of *Kanva-micro* is quite high-level, since all low-level boilerpate may be gracefully delegated to ASM library. It turns out that it is clearer and easier to present the technical details of *Kanva-micro* in well-established ASM terms rather than creating a special complicated formalism for simple things. So *all* technical details of the core logic of *Kanva-micro* are described just in listings.

1.6 Outline

Section 2 describes the core ingredients of the algorithm and also provides a necessary background about Java bytecode internals, section 3 goes into main technical details of the implementation, section 4 discusses experimental results of annotating some Java libraries, the cost of simplifications made in *Kanva-micro* and sketches how *Faba* overcomes these simplifications and section 5 mentions related work.

2 Algorithm

Kanva-micro performs intra-procedural analysis: each method is analyzed separately. Also *Kanva-micro* considers a general case: a Java library may be already partially annotated.

The technical side of the problem can be easily formulated without going into all subtle details of JVM semantics (these details are easily abstracted away). This section deals with Java bytecode from the point of view of nullness analysis and describes the algorithm of *Kanva-micro* informally.

2.1 A crash course of Java bytecode

Java source code comprises of a set of classes, each class in turn comprises of a set of fields and methods. A method in some sense is a "unit of execution". Java virtual machine is a stack virtual machine. A chain of method invocations is organized traditionally via call stack (composed of frames). Execution of a method is associated with a frame which has a storage for variables defined in this methods and also an operand stack. On a method invocation a current frame is pushed on the call stack (with a return address), a new frame is created and initialized with respect to passed arguments, a control is passed to instructions of called method. When execution of the method is completed, the previous frame is popped from the stack, a return value is put on the operand stack and control is transferred to restored return address.

There are more than 200 Java bytecode instructions, however, only a relatively small subset of them may cause null pointer error. Here is a list of such bytecode instructions with descriptions when there may be a `NPE` (`NullPointerException`) during execution of this instruction.

- `GETFIELD`, `PUTFIELD` – load/store a field of an object. Error if the corresponding object (an owner of the field) is `null`.
- `ARRAYLENGTH` – get the length of an array. Error if the corresponding array is `null`.
- `IALOAD`, `LALOAD`, `FALOAD`, `DALOAD`, `AALOAD`, `BALOAD`, `CALOAD`, `SALOAD` – load an int, long, float, double, reference, boolean, char, short value from an array. Error if the corresponding array is `null`.

- `IASTORE`, `LASTORE`, `FASTORE`, `DASTORE`, `AASTORE`, `BASTORE`, `CASTORE`, `SASTORE` – store an int, long, float, double, reference, boolean, char, short value in an array. Error if the corresponding array is `null`.
- `MONITORENTER` – synchronization by entering monitor of an object. Error if the corresponding object (monitor's owner) is `null`.
- `INVOKESTATIC` – call of a static class method (with arguments). Error if a `null` argument is passed into a parameter annotated as `@NotNull`.
- `INVOKEVIRTUAL`, `INVOKESPECIAL`, `INVOKEINTERFACE` – call of an instance method of an object. Error if the corresponding object is `null` or a `null` argument is passed into a parameter annotated as `@NotNull`.
- `ATHROW` – throw an exception. The case when an exceptions is thrown as a result of comparison of a parameters with `null` corresponds to assertions.

2.2 The core idea - inspection of process graph

Kanva-micro performs analysis for each method parameter of a reference type. Thus, for a method with three reference parameters three independent analyses will be run. The simple core idea is to consider *all* possible executions paths inside the method assuming that a parameter of interest is `null`. If *none* of these executions paths completes normally, the parameter is annotated as `@NotNull`. We are going to build a process tree [13] and inspect each branch of this tree for errors caused by `null` value of the parameter of interest. *Kanva-micro* doesn't try to build a perfect process tree – trees built by *Kanva-micro* may have unreachable branches. However, the existence of such branches doesn't affect the soundness of inference, but simplifies inference a lot.

It is sufficient to abstract away concrete values of local variables and operands in the operand stack in the current frame (corresponding to execution of a method being analyzed) and consider just two abstract values:

- `ParamValue` – a value passed into parameter of interest.
- `BasicValue` – any value (there is no information whether it corresponds to a parameter or not).

Obviously, `ParamValue` $\subseteq$ `BasicValue`. With ASM library it is easy to get a control flow graph for a method. To build all branches of the process tree it is enough to explicitly *unfold* all possible paths in this control flow graph. Of course, if there are cycles in the original control flow graph, then process tree will be infinite in general case. However, there is a simple *folding* strategy to fold such process tree into a finite process graph. For the purposes of inference it will be enough just to inspect resulting process graph.

2.3 Configurations and folding strategy

Nodes in a process tree are labeled with *configurations*. *Kanva-micro* represents a configuration as a pair of a program point and a store of abstract values. More precisely, the configuration is a pair `(insnIndex, frame)`:

- `insnIndex` – an index of a current instruction, each method is represented as a finite sequence of bytecode instructions,
- `frame` – a store (list) of local variables and stack operands, where abstract values are of two kinds: `ParamValue` and `BasicValue`. For each program point the size of the store is fixed and known in advance.

A nice fact is that a number of all possible configurations of a method process tree is finite. So, there is a natural folding strategy: during construction of the process graph to fold a current configuration to a more general configuration in the history of the current branch (when a current configuration is an instance of some previous configuration). Folding is performed when $c \subseteq c'$, where c is a current configuration and c' is a previous configuration. The relation $c \subseteq c'$ holds when instruction indices are the same and corresponding values (stored in slots with the same index i) are related as $v_i \subseteq v'_i$. Moreover, there is no need to construct a traditional back folding edge. So, when opportunity for folding is detected, development of the current branch of a process tree is stopped and the current node is just marked as a "cycle" leaf. Taken this into account, in what follows terms *process tree*, *process graph* and *graph of configurations* are used interchangeably.

A process graph (or a graph of configurations) built in this way is similar in a spirit to one built during supercompilation [18]. There are two main differences from traditional supercompilation:

- Driving (unfolding of a control flow graph) is done over abstract values.
- No residual program is generated, but the constructed process graph is used for a quite specific task: approximation of a method execution in the perspective of a possible dereference of a `null` parameter.

2.4 Tracking dereferences, keeping only interesting branches

The process graph is constructed to answer the following question:

> Let a certain parameter be `null`, do all possible executions of the method result in errors caused by this `null`?

If the answer is "yes", it is correct to annotate this parameter as `@NotNull`.

So, if there is a conditional of the form

```
1  if (param == null) {
2      ...
3  } else {
4      ...
5  }
```

there is no interest in the `else`-branch and no development of such branch is done in the constructed process tree. A subtree in a process tree corresponding to `then`-branch is a *null-aware subtree* (the intuition is that a programmer consider a case when a parameter is `null` explicitly).

During driving step, when an instruction is executed over abstract values, it is possible to detect situations listed in subsection 2.1 when dereference of `ParamValue` happens. Such transitions are said to be *dereferencing transitions*.

2.5 Approximating method execution

So, using described folding strategy, tracking dereferences of a parameter and keeping only interesting branches a *finite* process tree is developed. Additional information used for nullness analysis is stored in nodes and edges:

- Some leaves are marked as *cycle* leaves.
- Some edges are marked as *dereferencing* ones.
- Some subtrees are marked as *null-aware* ones.

Based on this labeling information, another labels ("nullness labels") describing a method behavior are produced in a bottom-up manner. First, leaves of the process tree are labeled with following values:

- `RETURN` – a leaf contains a return instruction and no dereferencing edge was taken on the path from the root.
- `NPE` – a dereferencing edge was taken on the path from the root, or a leaf's configuration points to a `ATHROW` instruction and this leaf belongs to a null-aware subtree.
- `ERROR` – a leaf's configuration corresponds to a `ATHROW` instruction but this leaf doesn't belong to a null-aware subtree.
- `CYCLE` – a leaf is a cycle leaf.

Next, nullness labels for other nodes are inferred from labels of its children. If a node has a single child node, then label is just propagated from the child to the parent. If a node has more child nodes then child labels are combined according to the following table:

	RETURN	NPE	ERROR	CYCLE
RETURN	RETURN	RETURN	RETURN	RETURN
NPE	RETURN	NPE	NPE	NPE
ERROR	RETURN	NPE	ERROR	ERROR
CYCLE	RETURN	NPE	ERROR	CYCLE

Finally, the root node is labeled. If it is labeled with `NPE`, then the corresponding parameter is annotated as `@NotNull`.

A nullness label in the root node is in a sense an approximation of method execution with the following meaning:

- `RETURN` – there is a possible execution path which completes normally and no proof that a given parameter is dereferenced on this path was found.
- `NPE` – all possible execution paths result in an exception and there is at least one path when this exception is caused by `null` value of parameter.
- `ERROR` – all possible execution paths result in an exception but there is no information whether or not such error is caused by `null` value of parameter.
- `CYCLE` – just a loop.

The reason why `NPE` and `ERROR` labels are distinguished is that there may be a method which just throws an exception without checking parameters like in the following code:

```
1  public void log(String msg) {
2      throw new UnsupportedOperationException();
3  }
```

```
1  package kanva.analysis
2
3  import org.objectweb.asm.tree.*
4  import org.objectweb.asm.tree.analysis.*
5  import kanva.declarations.*
6  import kanva.graphs.*
7
8  fun buildCFG(method: Method, methodNode: MethodNode): Graph<Int> =
9    ControlFlowBuilder().buildCFG(method, methodNode)
10
11 private class ControlFlowBuilder(): Analyzer<BasicValue>(BasicInterpreter()) {
12   private class CfgBuilder: GraphBuilder<Int, Int, Graph<Int>>(true) {
13     override fun newNode(data: Int) = Node<Int>(data)
14     override fun newGraph() = Graph<Int>(true)
15   }
16
17   private var builder = CfgBuilder()
18
19   fun buildCFG(method: Method, methodNode: MethodNode): Graph<Int> {
20     builder = CfgBuilder()
21     analyze(method.declaringClass.internal, methodNode)
22     return builder.graph
23   }
24
25   override protected fun newControlFlowEdge(insn: Int, successor: Int) {
26     val fromNode = builder.getOrCreateNode(insn)
27     val toNode = builder.getOrCreateNode(successor)
28     builder.getOrCreateEdge(fromNode, toNode)
29   }
30 }
```

Fig. 1. Construction of a control-flow graph

2.6 Correctness

Correctness of inference is almost obvious – a parameter is assumed to be `null` and *all* possible execution paths are considered. A parameter is annotated as `@NotNull` only if all execution paths result in an exception, which, in turn, satisfies the requirement that the method cannot complete normally when parameter is `null`.

3 Implementation

Initially this task has arisen in the context of development of the Kotlin programming language pursuing safer interoperability of Kotlin and Java. So, *Kanva-micro* is coded in Kotlin. The implementation is rather concise since many lower-level things are delegated to ASM library [12].

Technically, the full cycle of annotating a Java library consists of following stages:

1. *Context construction.* Context is a list of all signatures, their bytecode in ASM representation and a storage for inferred annotations. At the next steps inferred annotations are put in the context. Inferred annotations can be fetched from the context by a method signature.

```
class ParamValue(tp: Type?): BasicValue(tp)
class InstanceOfCheckValue(tp: Type?): BasicValue(tp)
class Configuration(val insnIndex: Int, val frame: Frame<BasicValue>)
fun startConfiguration(
    method: Method, methodNode: MethodNode,paramIndex: Int
): Configuration {
    val frame = Frame<BasicValue>(methodNode.maxLocals, methodNode.maxStack)
    val returnType = Type.getReturnType(methodNode.desc)
    val returnValue =
      if (returnType == Type.VOID_TYPE) null else BasicValue(returnType)
    frame.setReturn(returnValue)
    val args = Type.getArgumentTypes(methodNode.desc)
    var local = 0
    if (!method.access.isStatic()) {
        val thisValue=
            BasicValue(Type.getObjectType(method.declaringClass.internal))
        frame.setLocal(local++, thisValue)
    }
    for (i in 0..args.size - 1) {
        val value =
          if (i == paramIndex) ParamValue(args[i]) else BasicValue(args[i])
        frame.setLocal(local++, value)
        if (args[i].getSize() == 2)
          frame.setLocal(local++, BasicValue.UNINITIALIZED_VALUE)
    }
    while (local < methodNode.maxLocals)
        frame.setLocal(local++, BasicValue.UNINITIALIZED_VALUE)
    return Configuration(0, frame)
}
```

Fig. 2. Construction of a start configuration

2. *Construction of dependency graph, calculation of strongly connected components.* What described in the previous section is just one iteration of the inference cycle. Annotations of different methods may depend on each other, since inference of annotations for a given method relies on annotations for methods called from the current method. So, annotating is an iterative process. To minimize the number of iterations, the graph of dependencies between methods is constructed, strongly connected components are calculated and then sorted in reverse topological order.
3. *Iterative inference within a single component.* All members of a component are put in a queue. Then members are pulled from this queue one by one and **the described algorithm is run for each of not yet annotated parameters**. If a new annotation is inferred, dependent methods are added into the queue. Obviously, this process converges.

Steps 1 and 2 are rather trivial and implemented in a standard way. An interested reader may consult the full source code for details. However, a single iteration of inference is rather interesting from a technical point of view. And this part heavily relies on ASM library.

First, a method's control flow graph is built. ASM provides a number of utilities for bytecode analyses. One of such utilities is `Analyzer`. `Analyzer` performs basic bytecode analyses given a semantic bytecode interpreter. Also ASM library provides a simple interpreter `BasicInterpreter`. `Analyzer` and

```
class ParamSpyInterpreter(val context: Context): BasicInterpreter() {
  var dereferenced = false
  fun reset() {
    dereferenced = false
  }

  public override fun unaryOperation(
   insn: AbstractInsnNode, value: BasicValue
  ): BasicValue? {
    if (value is ParamValue)
      when (insn.getOpcode()) {
        GETFIELD, ARRAYLENGTH, MONITORENTER ->
          dereferenced = true
        CHECKCAST ->
          return ParamValue(Type.getObjectType((insn as TypeInsnNode).desc))
        INSTANCEOF ->
          return InstanceOfCheckValue(Type.INT_TYPE)
      }
    return super.unaryOperation(insn, value);
  }

  public override fun binaryOperation(
    insn: AbstractInsnNode, v1: BasicValue, v2: BasicValue
  ): BasicValue? {
    if (v1 is ParamValue)
      when (insn.getOpcode()) {
        IALOAD, LALOAD, FALOAD, DALOAD, AALOAD,
        BALOAD, CALOAD, SALOAD, PUTFIELD ->
          dereferenced = true
      }
    return super.binaryOperation(insn, v1, v2)
  }

  public override fun ternaryOperation(
    insn: AbstractInsnNode, v1: BasicValue, v2: BasicValue, v3: BasicValue
  ): BasicValue? {
    if (v1 is ParamValue)
      when (insn.getOpcode()) {
        IASTORE, LASTORE, FASTORE, DASTORE,
        AASTORE, BASTORE, CASTORE, SASTORE ->
          dereferenced = true
      }
   return super.ternaryOperation(insn, v1, v2, v3)
  }

  public override fun naryOperation(
    insn: AbstractInsnNode, values: List<BasicValue>
  ): BasicValue? {
    if (insn.getOpcode() != INVOKESTATIC)
      dereferenced = values.first() is ParamValue
    if (insn is MethodInsnNode) {
      val method = context.findMethodByMethodInsnNode(insn)
      if (method != null && method.isStable())
        for (pos in context.findNotNullParamPositions(method))
          dereferenced = dereferenced || values[pos.index] is ParamValue
    }
    return super.naryOperation(insn, values);
  }
}
```

Fig. 3. Semantic interpreter for driving and tracking dereference of `ParamValue`

`BasicInterpreter` are used by *Kanva-micro* to construct a method's control flow graph. How this is done is shown in a listing in Figure 1. The function

```
class NullParamSpeculator(val methodContext: MethodContext, val pIdx: Int) {
  val method = methodContext.method
  val cfg = methodContext.cfg
  val methodNode = methodContext.methodNode
  val interpreter = ParamSpyInterpreter(methodContext.ctx)
  fun shouldBeNotNull(): Boolean = speculate() == Result.NPE
  fun speculate(): Result = speculate(
    startConfiguration(method, methodNode, pIdx), listOf(),false, false
  )

  fun speculate(
    conf: Configuration, history: List<Configuration>,
    alreadyDereferenced: Boolean, nullPath: Boolean
  ): Result {
    val insnIndex = conf.insnIndex
    val frame = conf.frame
    if (history.any{it.insnIndex==insnIndex && isInstanceOf(frame, it.frame)})
      return Result.CYCLE
    val cfgNode = cfg.findNode(insnIndex)!!
    val insnNode = methodNode.instructions[insnIndex]
    val (nextFrame, dereferencedHere) = execute(frame, insnNode)
    val nextConfs =
      cfgNode.successors.map{Configuration(it.insnIndex, nextFrame)}
    val nextHistory = history + conf
    val dereferenced = alreadyDereferenced || dereferencedHere
    val opCode = insnNode.getOpcode()
    return when {
      opCode.isReturn() && dereferenced -> Result.NPE
      opCode.isReturn()                 -> Result.RETURN
      opCode.isThrow() && dereferenced  -> Result.NPE
      opCode.isThrow() && nullPath      -> Result.NPE
      opCode.isThrow()                  -> Result.ERROR
      opCode == IFNONNULL && Frame(frame).pop() is ParamValue ->
        speculate(nextConfs.first(), nextHistory, dereferenced, true)
      opCode == IFNULL && Frame(frame).pop() is ParamValue ->
        speculate(nextConfs.last(), nextHistory, dereferenced, true)
      opCode == IFEQ && Frame(frame).pop() is InstanceOfCheckValue ->
        speculate(nextConfs.last(), nextHistory, dereferenced, true)
      opCode == IFNE && Frame(frame).pop() is InstanceOfCheckValue ->
        speculate(nextConfs.first(), nextHistory, dereferenced, true)
      else ->
        nextConfs.map{
          speculate(it, nextHistory, dereferenced, nullPath)
        } reduce{ r1, r2 -> r1 join r2}
    }
  }

  fun execute(
    frame: Frame<BasicValue>, insnNode: AbstractInsnNode
  ): Pair<Frame<BasicValue>, Boolean> = when (insnNode.getType()) {
    AbstractInsnNode.LABEL, AbstractInsnNode.LINE, AbstractInsnNode.FRAME ->
      Pair(frame, false)
    else -> {
      val nextFrame = Frame(frame)
      interpreter.reset()
      nextFrame.execute(insnNode, interpreter)
      Pair(nextFrame, interpreter.dereferenced)
    }
  }
}
```

Fig. 4. Inference of `@NotNull` annotation

`buildCFG` builds a directed graph whose nodes are labeled with indices of instructions of a method and edges correspond to transitions between instructions.

`BasicValue` introduced in subsection 2.2 is already implemented in ASM. The class `Frame` provided by ASM corresponds to a frame holding abstract values. `BasicInterpreter` already implements execution of bytecode instructions over `BasicValue`s is the desired way. *Kanva-micro* extends `BasicInterpreter` in order to distinguish between `BasicValue` and `ParamValue`. Notions of `ParamValue`s and configurations are depicted in a listing in Figure 2. Class `InstanceOfCheckValue` is for tracking `instanceof` checks. The function `startConfiguration` presented in Figure 2 creates a start configuration (placed in the root node of a process tree) for a given method and an index of a parameter being analyzed. The main logic of `startConfiguration` is that all values in frame except a given parameter are initialized with `BasicValue`.

`BasicInterpreter` provided by ASM already has almost everything needed for driving. The missed parts are:

- Tracking of dereferencing of `ParamValue`.
- Handling of `instanceof` checks of `ParamValue`.
- Knowledge about already inferred annotations (to detect dereferencing).
- Propagation of `ParamValue` during class casting.

All these parts are implemented in class `ParamSpyInterpreter` shown in Figure 3. The most interested lines are 52-56: if the current parameter of interest is passed as an argument to another parameter (of some method) already annotated as `@NotNull`, it is handled in the same way as dereferencing of the current parameter.

The main analysis is implemented in the class `NullParamSpeculator` shown in Figure 4. `NullParamSpeculator` holds a `methodContext`, which contains everything needed for inference, and an index of a parameter being annotated. The method `shouldBeNotNull` returns `true` if an approximation of method execution is `NPE`. A process tree is not constructed explicitly here, since it is enough to get a nullness label for the root configuration. The call to `speculate(conf, history, alreadyDereferenced, nullPath)` results in one of `RETURN`, `NPE`, `ERROR`, `CYCLE` nullness labels. The call arguments are:

- `conf` – the current configuration (for which nullness label should be calculated),
- `history` – a list of already encountered configurations (for folding),
- `alreadyDereferenced` – whether dereferencing was already detected on a path from the root to current configuration,
- `nullPath` – if `nullPath` is `true`, it means that current configuration belongs to a null-aware subtree.

Let's iterate through the code of the `speculate` method line-by-line.

If there is a more general configuration in the history, folding is performed, the corresponding label is `CYCLE`. Otherwise, the current instruction is executed – the `execute` method returns a pair of a next frame and a boolean whether there

was dereferencing of the parameter during instruction execution. If the current instruction is a return or throw instruction, then a nullness label is calculated based on the `dereferenced` and `nullPath` flags. Otherwise, if the current instruction is `IFNULL` or `IFNONNULL` and a value being tested is `ParamValue` (it corresponds to conditionals `if (param == null)` and `if (param != null)`), a corresponding null-aware subtree is processed (the `nullPath` flag to `true`).

The same logic applies to handling of `if (param instanceof SomeClass)` conditional. When `param` is `null`, this check results in `false`. The implementation is a bit verbose since there is no special instruction in Java bytecode for such conditional and this check is compiled into the sequence of two instructions: `INSTANCEOF` and `IFEQ`. The `INSTANCEOF` instruction is handled by `ParamSpyInterpreter`: if an operand is `ParamValue`, then a special `InstanceOfCheckValue` value is produced. The `IFEQ` instruction is handled inside the `speculate` method: when the current instruction is `IFEQ` and an operand on the top of the stack is `InstanceOfCheckValue`, then the `if (param instanceof SomeClass)` construction is recognized and only a branch that corresponds to null parameter is considered. (Handling of the `IFNE` instruction corresponds to `if (!(param instanceof SomeClass))` construction.)

Otherwise, nullness labels for child configurations are calculated and combined. This concludes the discussion of the implementation.

4 Discussion

In a sense, *Kanva-micro* performs domain-specific supercompilation [15] of Java bytecode, abstracting away almost all aspects of operational semantics not associated with nullness analysis. Because of these abstractions, representation of configurations becomes extremely simple – just a bit vector. The interesting fact is that configurations are so generalized in advance, that *no traditional online generalization* is required to ensure termination of supercompilation. But this comes for the price that a constructed process tree of method execution is not perfect in a general case.

4.1 The cost of simplifications

The main point of the *Kanva-micro* project is simplicity, focusing on the essence of the method and ignoring some technical details for the sake of brevity of presentation. However, there are two significant drawbacks that simplifications that make *Kanva-micro* not ready for production use.

Exponential complexity The main drawback of *Kanva-micro* is that this algorithm is of exponential complexity in general case. This complexity may exploded by a method with sequential conditionals.

```
1  if ( ... ) {
2
3  }
```

```
4  if ( ... ) {
5
6  }
7  if ( ... ) {
8
9  }
```

If there are n sequential conditionals in a method, then process tree constructed by *Kanva-micro* will contain 2^n branches in the worst case.

Memory usage The bytecode of a library method is processed by *Kanva-micro* more than one time: the first time when a graph of dependencies between methods is constructed and then during iterative inference of annotations inside a strongly connected component of the graph of dependencies. Loading and parsing the bytecode of a method from scratch every time without additional processing of binaries is problematic, so *Kanva-micro* loads all library bytecode into memory at once in advance. This means that the amount of memory required by *Kanva-micro* is proportional to the size of a library, which is not acceptable from a practical point of view.

4.2 The Faba project

The *Faba* project [4] overcomes mentioned drawbacks by smart handling of the library bytecode. *Faba* processes a binary library in two stages:

1. Indexing a library: the result of indexing is a set of equations over a lattice.
2. Solving equations.

At the first stage each method is processed exactly once. After the bytecode for a method is indexed, it is unloaded. Equations are not memory consuming, so the problem of memory usage disappears.

During indexing a method, *Faba* exploits memoization and sharing facilities. The main observation is that in a sequence of conditionals in *real libraries* the majority of conditionals are irrelevant to nullness analysis (do not test a parameter for nullity). Driving of both branches of "irrelevant" conditions result *in most cases* in the same configurations in two nodes of the process tree, these nodes are joined. In general case *Faba* is also of exponential complexity, but this exponential complexity is not exploded by *real-world libraries*.

Both problems may be tamed in naive but simple ways: the memory usage problem may be solved via unloading the bytecode for a method after its bytecode is processed by the current iteration and loading it from the scratch from the disk. A simple ad-hoc way to mitigate exponential complexity of *Kanva-micro* is just to limit the number of processed configurations. When this limit is reached, analysis for the method stops and infers nothing.

4.3 Experiments and more details

The *Kanva-micro* project [8] provides utilities to annotate popular Java libraries (available as Maven artifacts) in a simple way. The project page also has a set of

experimental result for running inference with different settings. The interesting fact is that limiting the number of processed configurations by a reasonable number (say, by 5000) *Kanva-micro* infers about 95 percent of annotations inferred by *Faba* in comparable time.

An interested reader may also consult the *Kanva-micro*'s wiki for more more technical details related to implementation and experiments.

5 Related work

Initially *Kanva-micro* was developed in the context of JetBrains KAnnotator tool [7]. KAnnotator is based on abstract interpretation and infers different nullness annotations (for method parameters and for method results) in a single pass. So, abstract domains and logic of approximations in KAnnotator is much more complex that of *Kanva-micro*.

On the contrary, *Kanva-micro* is specialized to infer just one type of nullness annotations. The *Faba* project infers not only `@NotNull` annotations for method parameters, but also `@NotNull` annotations for method results and `@Contract` annotations [5]. All *Faba* inferencers are quite similar and based on supercompilation but have very different abstract domains, logic of approximations and logic for sharing configurations.

The pragmatic observation from developing *Kanva-micro* and *Faba* is that it is more practical to have a set of specialized inferencers which run independently and may reuse results of each other via context rather than a tool that runs different analyses together in a single pass.

The main goal of KAnnotator, *Kanva-micro* and *Faba* is to annotate existing Java libraries for safer usage. Inference of annotations happens on bytecode level, no source is required.

Surprisingly, as we can judge from existing literature, this task was not addressed in academia from practical point of view before. The closest existing tool is NIT [14]. NIT infers `@NotNull` and `@Nullable` annotations but these annotations have different semantics. NIT considers Java bytecode as a single application and starts analysis from so called entry points. A `@NotNull` parameter annotation in NIT setting means that during execution of an application `null` will never be passed into this parameter, other annotations have similar semantics – they describe which values may be passed to parameters and returned from methods during executions of a specific application. NIT doesn't consider bytecode at library level. NIT motivation is that such analysis maybe used to detect bugs in an applications. Another possible application of NIT annotations is bytecode optimizations – removing unnessecary checks from bytecode.

Another tool that infers nullness information from bytecode is Julia [17]. Again, this information is inferred with the goal of analysis – the main application is to generate a set of warnings about possible null pointer exceptions.

There is a tool called JACK [11,16] which verifies Java bytecode with respect of `@NotNull` annotations, ensuring that `null` will never be passed to a `@NotNull` variable or parameter.

Note that *Kanva-micro* annotations are semantic-based. There is a lot of works devoted to checking and inferencing nullness annotations in source code, but these annotations have different semantics, since they may forbid some executions paths not resulting in null pointer exception. Also many source-based annotation inferencers require an additional user's input.

References

1. ASM Framework. http://asm.ow2.org/.
2. Ceylon programming language. http://ceylon-lang.org/.
3. Eclipse user guide: using null annotations. http://help.eclipse.org/kepler/topic/org.eclipse.jdt.doc.user/tasks/task-using_null_annotations.htm.
4. Faba, fast bytecode analysis. https://github.com/ilya-klyuchnikov/faba.
5. IntelliJ IDEA 13.1 Help. @Contract Annotations. http://www.jetbrains.com/idea/webhelp/@contract-annotations.html.
6. IntelliJ IDEA How-To, Nullable How-To. https://www.jetbrains.com/idea/documentation/howto.html.
7. KAnnotator. https://github.com/JetBrains/kannotator.
8. Kanva-micro. https://github.com/ilya-klyuchnikov/kanva-micro.
9. Kotlin programming language. http://kotlin.jetbrains.org/.
10. The Checker Framework. Custom pluggable types for Java. http://types.cs.washington.edu/checker-framework/.
11. The Java Annotation Checker (JACK). http://homepages.ecs.vuw.ac.nz/~djp/JACK/.
12. E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.
13. R. Glück and A. Klimov. Occam's razor in metacompuation: the notion of a perfect process tree. In *WSA '93: Proceedings of the Third International Workshop on Static Analysis*, pages 112–123, London, UK, 1993. Springer-Verlag.
14. L. Hubert. A non-null annotation inferencer for java bytecode. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 36–42. ACM, 2008.
15. A. Klimov, I. Klyuchnikov, and S. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation. In *Third International Valentin Turchin Workshop on Metacomputation*, 2012.
16. C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Compiler Construction*, pages 229–244. Springer, 2008.
17. F. Spoto. The nullness analyser of julia. In E. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 405–424. Springer Berlin Heidelberg, 2010.
18. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

An Approach for Modular Verification of Multi-Result Supercompilers
(Work in Progress)

Dimitur Nikolaev Krustev

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

Abstract. Multi-result supercompilation is a recent and promising generalization of classical supercompilation. One of its goals is to permit easy construction of different supercompilers from a reusable top-level algorithm and independently implemented components for driving, folding, etc. The problem of preservation of semantics by multi-result supercompilers has not yet been studied in detail. So, while the implementation of a new multi-result supercompiler is simplified by the high degree of modularity, its verification is not. To alleviate this burden, we search for a set of sufficient conditions on the basic building blocks (such as driving or folding), which – if met – can be plugged into general theorems, to ensure supercompiler correctness. If the proposed approach proves successful, it will make multi-result supercompiler verification as easy and modular as the implementation itself.

1 Introduction

Multi-result supercompilation [10,11] is a recent generalization of classical supercompilation [18,19]. One of its key insights is to permit generalization to happen at any moment, and to consider and collect the different graphs of configurations arising from different choices about generalization. Recall that in classical supercompilation generalization is only applied when the whistle blows and folding is not possible. Paradoxically it turns out that in certain situations early generalization can lead to an optimal result, which cannot be obtained using the classical approach [8].

Another advantage of multi-result supercompilation (MRSC[1]) is that, from the beginning, it was designed in a modular way, as follows:

- a generic high-level algorithm, which is largely independent of the particular choice of object language;
- a small set of primitive operations, which encapsulate the language-specific parts of the supercompiler algorithm.

[1] The abbreviation *MRSC* is usually reserved for the original implementation in Scala, "The MRSC Toolkit". We take the liberty to use it also for multi-result supercompilation in general, for brevity.

This modularity allows the programmer to easily create supercompilers for different object languages, including highly specialized ones for particular DSLs [8]. A further refinement of modularity is present in a recent Agda formalization of MRSC [4], which is based on a more streamlined set of primitive operations. Moreover, the Agda formalization uses a "big-step" definition of MRSC + whistles based on inductive bars [3], which further simplify the main data structures and algorithms. The main goal of Grechanik et al. [4] is to formalize a more compact representation of the whole set of results produced by MRSC, to prove this representation correct w.r.t. the original representation (simple list of graphs), and to show that many useful operations (filtering the set of results, selecting an optimal result by certain criteria) can be directly and more efficiently performed on the compact representation. The formalization of object language semantics and verification of the preservation of this semantics by MRSC is beyond the scope of that paper.

As verification of semantics preservation by supercompilers is an interesting and practically useful topic in itself [12–14], the approach we describe here aims to fill this gap, and to propose a way for verifying semantics preservation of supercompilers based on big-step MRSC. We take the Agda formalization of Grechanik et al. [4] as a starting point (ported to Coq) and augment it with:

- a set of primitives for describing the semantics of the object language (Sec. 3)
- based on these primitives:
  - a formal semantics of trees of configurations produced by multi-result driving + generalization (Sec. 3);
  - a formal semantics of graphs of configurations produced by multi-result supercompilation (Sec. 4);
- a more precise representation of backward graph nodes, which result from folding (Sec. 4), and an MRSC algorithm adapted to this representation (Sec. 5).

We further propose an approach for the modular verification of semantics preservation for any supercompiler built using the proposed components. The main idea is to provide, as much as possible, general proofs of correctness for the high-level parts of the supercompiler, which do not depend on the object language. Implementers of particular supercompilers then only need to fill those parts of the correctness proof that are specific to the object language and the particular choice of supercompiler primitive operations (generalization, folding, whistle, ...). A key idea for simplifying and modularizing the overall correctness proof is the assumption that any graph created by multi-result supercompilation, if unfolded to a tree, can also be obtained by performing multi-result driving alone. This important assumption allows to decompose the correctness verification in several stages:

- verification of semantics preservation for driving + generalization in isolation. In other words, all trees obtained from driving must preserve language semantics (Sec. 3). As this step is mostly language-specific, it must be done separately for each supercompiler.

- general proof that graphs produced by multi-result supercompilation have the same meaning as the trees to which they can be unfolded (Sec. 4). This proof can be reused directly for any supercompiler based on the described algorithm.
- using some assumptions about the folding operation, general proof that any MRSC-produced graph unfolded into a tree can also be produced by driving alone (Sec. 5). This proof can also be reused for any particular supercompiler, only the assumptions about the folding operation must be verified separately in each case. The advantage is that the impact of folding is limited only to checking its specific conditions; other parts of the proof can ignore folding completely.

The whistle is used only for ensuring termination, and has no impact on semantics preservation at all.

The current implementation of the proposed approach is in Coq and we give most definitions and property statements directly in Coq[2]. Understanding the ideas behind most such fragments of Coq source should not require any deep knowledge of that language, as they use familiar constructs from functional programming languages and formal logic, only with slight variations in syntax. We also stress that the approach is not limited in any way to working only with Coq, or even to computer-assisted formal verification in general. Of course, the implementation should be directly portable to other dependently-typed languages such as Agda or Idris. Most importantly, the implementation of the main data structures and algorithms should be easy to port as well to any modern language with good support for functional programming (Haskell, ML, Scala, ...). In the latter case a realistic alternative to formal proofs is the use of the sufficient conditions on basic blocks in conjunction with a property-based testing tool like QuickCheck [2], which can still give high confidence in the correctness of the supercompiler.

2 Preliminaries

Before delving into the main components of the proposed MRSC formalization, let's quickly introduce some preliminary definitions, mostly necessary to make things work in a total dependently-typed language like Coq or Agda.

2.1 Modeling General Recursive Computations

In order to formally prove correctness results, we need first to formalize programming language semantics, in an executable form, in a total language. As any kind of interpreter (big-step, small-step, ...) for a Turing-complete language

[2] All proofs, some auxiliary definitions, and most lemmas are omitted. Interested readers can find all the gory details in the Coq sources accompanying the paper: `https://sites.google.com/site/dkrustev/Home/publications`

is a partial, potentially non-terminating function, we must select some roundabout way to represent such functions. While different partiality monads based on coinduction exist, they all have different advantages and drawbacks [1], and none of them is practical for all possible occasions. So we stick to a very basic representation: monotonic functions of type **nat** → **option** A, where we use the usual ordering for **nat** and an "information ordering" for **option** A (which is explicitly defined in Fig. 1). The **nat** argument serves as a finite limit to the amount of computation performed. If we can complete it within this limit, we return the resulting value wrapped in Some, otherwise we return None. The monotonicity condition simply states, that if we can return a value within some limit, we will always return the same value when given higher limits. Such a representation should be compatible with most kinds of existing partiality monads, which can be equipped with a *run* function of type **nat** → **option** A. We are interested in the extensional equivalence of such computations: if one of 2 computations, given some limit, can return a value, the other also has some (possibly different) limit, after which it will return the same value, and vice versa. This is captured in the definition of EvalEquiv. Note that if converting the definition of MRSC below

```
Definition FunNatOptMonotone {A} (f: nat → option A) : Prop :=
  ∀ n x, f n = Some x → ∀ m, n ≤ m → f m = Some x.
Inductive OptInfoLe {A} : option A → option A → Prop :=
  | OptInfoLeNone: ∀ x, OptInfoLe None x
  | OptInfoLeSome: ∀ x, OptInfoLe (Some x) (Some x).
Definition EvalEquiv {A} (f1 f2: nat → option A) : Prop :=
  ∀ x, (∃ n, f1 n = Some x) ↔ (∃ n, f2 n = Some x).
```

Fig. 1: Model of general recursive computations

to a Turing-complete functional language like Haskell or ML, it would probably be more practical to replace this encoding of potentially non-terminating computations with a representation of lazy values.

2.2 Bar Whistles

Following [4], we use inductive bars [3] as whistles (Fig. 2). Recall that the main job of the whistle is to ensure termination of the supercompiler. The advantage of inductive bars is that the supercompiler definition becomes structurally-recursive on the bar, making it obviously terminating. Different kinds of bars can be built in a compositional way, and we can also build an inductive bar from a decidable almost-full relation – almost-full relations being another constructive alternative to the well-quasi orders classically used in supercompilation [20]. We do not go into further detail here, because the construction of a suitable inductive bar is orthogonal to the correctness issues we study.

```
Inductive Bar {A: Type} (D: list A → Type) : list A → Type :=
  | BarNow: ∀ {h: list A} (bz: D h), Bar D h
  | BarLater: ∀ {h: list A} (bs: ∀ c, Bar D (c :: h)), Bar D h.
Record BarWhistle (A: Type) : Type := MkBarWhistle {
  dangerous: list A → Prop;
  dangerousCons: ∀ (c: A) (h: list A), dangerous h → dangerous (c :: h);
  dangerousDec: ∀ h, {dangerous h} + {¬ (dangerous h)};
  inductiveBar: Bar dangerous nil
}.
```

Fig. 2: Inductive bars for whistles

3 Driving, Trees of Configurations and Their Semantics

3.1 Trees of Configurations, Sets of Trees

Given some abstract type representing *configurations*, we can give a straightforward definition of trees of configurations:

```
Variable Cfg: Type.
CoInductive CfgTree: Type := CTNode: Cfg → FList CfgTree → CfgTree.
```

What is conspicuously missing are *contractions*. We assume – as suggested in [4] – that when present, the contraction of an edge is merged with the configuration below the edge. So we need to only deal with configurations, thus simplifying the formal definition of MRSC. As an obscure technical detail, we use an alternative definition of lists here (FList A) just to avoid some restrictions of Coq's productivity checker [15]. (Readers not particularly interested in such idiosyncrasies of Coq can safely pretend that FList is the same as **list**, drop the "fl" prefix in functions/predicates like flMap, FLExists, ..., and consider list2flist and flist2list as identity functions.)

Multi-result driving will typically produce an infinite list of infinite trees of configurations. It appears hard to explicitly enumerate this list in a total language, as it grows both in width and in height at the same time. As an alternative, we can re-use the trick of Grechanik et al [4] to make a compact representation of the whole set of trees produced by multi-result driving. The meaning of the encoding is probably easiest to grasp in terms of the process of multi-result driving itself, to which we shall come shortly.

```
CoInductive CfgTreeSet: Type :=
  | CTSBuild: Cfg → FList (FList CfgTreeSet) → CfgTreeSet.
```

The only important operation on such sets of trees is membership. Luckily it is definable as a coinductive relation. This definition is best illustrated by a picture (Fig. 3).

```
CoInductive TreeInSet: CfgTree → CfgTreeSet → Prop :=
  | TreeInBuild: ∀ c (ts: FList CfgTree) (tsss: FList (FList CfgTreeSet)),
      FLExists (fun tss ⇒ FLForall2 TreeInSet ts tss) tsss
      → TreeInSet (CTNode c ts) (CTSBuild c tsss).
```

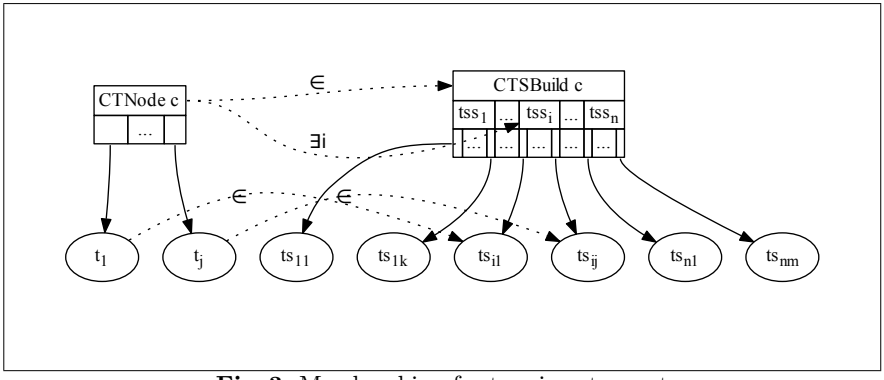


Fig. 3: Membership of a tree in a tree-set

3.2 Driving (+ Generalization)

We explicitly try to follow as closely as possible the Agda formalization of MRSC proposed by Grechanik et al [4] (modulo alpha-equivalence). We assume the same primitive – *mrscSteps* – for performing driving and generalization. Given a current configuration, it will produce a list of results, each of which is in turn a list of new configurations. Any such list of configurations is the result of either a driving step or a generalization step performed on the initial configuration. We can use this primitive to build the whole set of trees corresponding to a given initial configuration. buildCfgTrees is actually the full high-level algorithm for multi-result driving!

```
Variable mrscSteps: Cfg → list (list Cfg).
CoFixpoint buildCfgTrees (c: Cfg) : CfgTreeSet :=
  CTSBuild c (flMap (flMap buildCfgTrees)
    (list2flist (map list2flist (mrscSteps c)))).
```

3.3 Tree Semantics

We first introduce several abstract primitives, related to the semantics of the object language (Fig. 4). The most important one – *evalCfg* – represents a "reference" interpreter for (configurations of) the object language. As it is encoded using the chosen representation for general recursive functions, it must satisfy the corresponding monotonicity condition. The other 3 primitives (*evalNodeXXX*) supply the language-specific parts of the "tree interpreter". The generic algorithm of this tree interpreter – which formally defines the semantics of trees of configurations – is given in Fig. 5. It is easy to deduce the purpose of the 3 primitives from this definition itself: *evalNodeResult* computes (if possible) the final value for the current tree node, while *evalNodeInitEnv* and *evalNodeStep* serve to maintain the evaluation environment (assuming a fixed evaluation order from left to right for subtrees of the node). Note that, while *evalCfg* can be a

general recursive computation, the primitives for the tree interpreter do not have a "fuel" argument – they are expected to be total functions.

```
Variable Val: Type. Variable EvalEnv: Type.
Variable evalCfg: EvalEnv → Cfg → nat → option Val.
Hypothesis evalCfg_monotone: ∀ env c, FunNatOptMonotone (evalCfg env c).
Variable evalNodeInitEnv: EvalEnv → Cfg → EvalEnv.
Variable evalNodeStep: EvalEnv → Cfg → list (option Val) → option Val → EvalEnv.
Variable evalNodeResult: EvalEnv → Cfg → list (option Val) → option Val.
```

Fig. 4: Evaluation primitives

```
Fixpoint evalCfgTree (env: EvalEnv) (t: CfgTree) (n: nat) {struct n} : option Val :=
  match n with
  | 0 ⇒ None
  | S n ⇒ match t with
    | CTNode c ts ⇒
      let stepf (p: EvalEnv × list (option Val)) (t: CfgTree)
        : EvalEnv × list (option Val) :=
        let env := fst p in let ovs := snd p in
        let ov := evalCfgTree env t n in
        (evalNodeStep env c ovs ov, ov::ovs) in
      evalNodeResult env c
        (snd (fold_left stepf (flist2list ts) (evalNodeInitEnv env c, nil)))
    end
  end.
```

Fig. 5: Tree interpreter

The following requirement – *evalCfg_evalCfgTree_equiv* – is the cornerstone for establishing MRSC correctness: we assume that each tree produced by multi-result driving is semantically equivalent to the initial configuration. This assumption permits to easily establish another natural coherence property – that all trees resulting from multi-result driving are semantically equivalent.

```
Hypothesis evalCfg_evalCfgTree_equiv: ∀ env c t,
  TreeInSet t (buildCfgTrees c)
  → EvalEquiv (evalCfg env c) (evalCfgTree env t).
Lemma AllTreesEquiv: ∀ env c t1 t2, let ts := buildCfgTrees c in
  TreeInSet t1 ts → TreeInSet t2 ts →
  EvalEquiv (evalCfgTree env t1) (evalCfgTree env t2).
```

4 Graphs of Configurations, Graph Semantics

4.1 Graph definition

The definition of MRSC graphs of configurations (Fig. 6) is still similar to the Agda formalization of Grechanik et al. [4], the important difference being the treatment of backward nodes resulting from folding: they contain an index determining the upper node + a function that can convert the upper configuration to the lower one. Since a graph having a backward node as root has no sense (and is never created by MRSC), we capture this invariant by a dedicated data type – **TopCfgGraph**.

```
Inductive CfgGraph : Type :=
  | CGBack: nat → (Cfg → Cfg) → CfgGraph
  | CGForth: Cfg → list CfgGraph → CfgGraph.
Inductive TopCfgGraph : Type :=
  | TCGForth: Cfg → list CfgGraph → TopCfgGraph.
Definition top2graph (g: TopCfgGraph) : CfgGraph :=
  match g with
  | TCGForth c gs ⇒ CGForth c gs
  end.
```

Fig. 6: Graphs of configurations

4.2 Converting Graphs to Trees

We can define the unfolding of a graph of configurations into a tree of configurations (Fig. 7). The main work is done in a helper function graph2treeRec, which must maintain several recursion invariants. The parameter *topG* always keeps a reference to the root of the tree, and is used to give a meaning even to incorrect graphs, in which the index of a backward node is too big. In such cases we simply assume the index points to the root of the graph. *gs* contains – in reverse order – all nodes in the path to the root of the graph; it grows when passing through a forward node and shrinks back when passing through a backward node. The parameter *f* is the composition of all configuration transformations of backward nodes, through which we have already passed.

4.3 Graph Semantics

We define the semantics of graphs of configurations by defining a "graph interpreter" (Fig. 8). Again, we use a helper function evalGraphRec, whose parameters *topG*, *gs*, and *f* are used in the same way as in graph2treeRec, in essence performing graph unfolding "on the fly". For the interpretation of each node we reuse the same language-specific primitives we have used for the tree interpreter.

```
CoFixpoint graph2treeRec (topG: TopCfgGraph) (gs: list TopCfgGraph)
  (f: Cfg → Cfg) (g: CfgGraph) : CfgTree :=
  let topGraph2tree (gs: list TopCfgGraph) (f: Cfg → Cfg) (g: TopCfgGraph)
    : CfgTree :=
    match g with
    | TCGForth c gs1 ⇒
        CTNode (f c) (flMap (graph2treeRec topG (g::gs) f) (list2flist gs1))
    end in
  match g with
  | CGBack i f1 ⇒ match nthWithTail i gs with
    | Some (backG, gs1) ⇒ topGraph2tree gs1 (fun c ⇒ f (f1 c)) backG
    | None ⇒ topGraph2tree nil (fun c ⇒ f (f1 c)) topG
    end
  | CGForth c gs1 ⇒ topGraph2tree gs f (TCGForth c gs1)
  end.
Definition graph2tree (g: TopCfgGraph) : CfgTree :=
  graph2treeRec g nil (fun c ⇒ c) (top2graph g).
```

Fig. 7: Unfolding a graph into a tree

```
Fixpoint evalGraphRec (topG: TopCfgGraph) (gs: list TopCfgGraph)
  (f: Cfg → Cfg) (env: EvalEnv) (g: CfgGraph) (n: nat) {struct n} : option Val :=
  match n with
  | 0 ⇒ None
  | S n ⇒
    let evalTopGraph (gs: list TopCfgGraph) (f: Cfg → Cfg) (g: TopCfgGraph)
          : option Val :=
      match g with
      | TCGForth c subGs ⇒
        let stepf (p: EvalEnv × list (option Val)) (g1: CfgGraph)
          : EvalEnv × list (option Val) :=
          let env := fst p in let ovs := snd p in
          let ov := evalGraphRec topG (g::gs) f env g1 n in
          (evalNodeStep env (f c) ovs ov, ov::ovs) in
        evalNodeResult env (f c)
          (snd (fold_left stepf subGs (evalNodeInitEnv env (f c), nil)))
      end in
    match g with
    | CGBack i f1 ⇒
      let g_gs := match nthWithTail i gs with
                    Some p ⇒ p | None ⇒ (topG, nil) end in
      evalTopGraph (snd g_gs) (fun c ⇒ f (f1 c)) (fst g_gs)
    | CGForth c subGs ⇒ evalTopGraph gs f (TCGForth c subGs)
    end
  end.
Definition evalGraph (env: EvalEnv) (g: TopCfgGraph) (n: nat) : option Val :=
  evalGraphRec g nil (fun c ⇒ c) env (top2graph g) n.
```

Fig. 8: Graph interpreter

4.4 Graph semantics correctness

It is not hard to spot that the definition of evalGraphRec has many similarities to the definitions of evalCfgTree and graph2treeRec. Actually evalGraphRec can be seen as a manually fused composition of the other 2 functions. This fact is formally verified by the following theorem, which is one of the 2 key intermediate results used in establishing MRSC correctness:

```
Theorem evalGraph_evalCfgTree: ∀ env g,
  EvalEquiv (evalGraph env g) (evalCfgTree env (graph2tree g)).
```

5 Multi-result Supercompilation

5.1 Definition

We need 2 more primitives (besides *mrscSteps*) in order to define a generic multi-result supercompiler. As we have already explained, we assume a whistle in the form of an inductive bar (Fig. 9). The signature of the folding primitive – *tryFold* – is determined by our decision how to encode backward nodes: if folding is possible, *tryFold* must return:

- a valid index into the list of previous configurations (*tryFold_length*);
- a configuration-transforming function, which will turn the old configuration (higher in the tree) into the current one (*tryFold_funRelatesCfgs*).

The requirement that no folding should be possible with empty history – *tryFold_nil_None* – is quite natural. The last requirement about *tryFold* – *tryFold_funCommutes* – deserves more attention. It states that any configuration transformation, returned by folding, commutes (in a way precisely defined in Fig. 9) with *mrscSteps*. The reason for this assumption is that it permits us to prove that unfolding a mrsc-produced graph will always result into a tree that can also be obtained through driving alone. The latter property is key for enabling modular verification of the different supercompilers produced by the proposed approach. This requirement is further discussed in Sec. 6.

Apart from folding, the rest of the mrsc definition is very similar to the one proposed by Grechanik et al. [4]. The main algorithm is encoded by the recursive function mrscHelper. The top-level function mrsc may seem complicated at first, but it is only because it uses some Coq-specific idioms to convert the final list of results from type **CfgGraph** to type **TopCfgGraph**. If we ignored this conversion, the definition would become:

```
mrsc (c: Cfg) : list CfgGraph := mrscHelper (inductiveBar whistle) c.
```

5.2 Containment of Graphs in Driving Tree Sets

As already hinted, the following containment result is the second key theorem necessary for ensuring mrsc correctness:

```
Theorem graph2tree_mrsc_In_buildCfgTrees: ∀ c g,
  In g (mrsc c) → TreeInSet (graph2tree g) (buildCfgTrees c).
```

It opens the way to replace establishing the semantic correctness of graphs with verifying only the semantic correctness of trees.

```
Definition CommutesWithSteps (f: Cfg → Cfg) :=
  ∀ c, mrscSteps (f c) = map (map f) (mrscSteps c).
Variable tryFold: list Cfg → Cfg → option (nat × (Cfg → Cfg)).
Hypothesis tryFold_nil_None: ∀ c, tryFold nil c = None.
Hypothesis tryFold_length: ∀ h c i f, tryFold h c = Some (i, f) → i < length h.
Hypothesis tryFold_funRelatesCfgs: ∀ h c i f c1 h1,
  tryFold h c = Some (i, f) → nthWithTail i h = Some (c1, h1) → c = f c1.
Hypothesis tryFold_funCommutes: ∀ h c i f,
  tryFold h c = Some (i, f) → CommutesWithSteps f.
Variable whistle: BarWhistle Cfg.
```

Fig. 9: Remaining MRSC primitives

```
Fixpoint mrscHelper (h: list Cfg)
  (b: Bar (dangerous whistle) h) (c: Cfg) {struct b} : list CfgGraph :=
  match tryFold h c with
  | Some (n, f) ⇒ CGBack n f :: nil
  | None ⇒
    match dangerousDec whistle h with
    | left _ ⇒ nil
    | right Hdang ⇒
      match b in (Bar _ h) return (¬ dangerous whistle h → list CfgGraph)
      with
      | BarNow h' bz ⇒ fun (Hdang: ¬ dangerous whistle h') ⇒
          match Hdang bz with end
      | BarLater h' bs ⇒ fun (_: ¬ dangerous whistle h') ⇒
          map (CGForth c) (flat_map (fun css : list Cfg ⇒
              listsProd (map (mrscHelper (bs c)) css)) (mrscSteps c))
      end Hdang
    end
  end.
Definition mrsc (c: Cfg) : list TopCfgGraph :=
  let gs := mrscHelper (inductiveBar whistle) c in
  mapWithInPrf gs
    (fun g Hin ⇒ match g return _ = g → TopCfgGraph with
      | CGBack n f ⇒ fun Heq ⇒
        let Hin' := eq_rect g (fun g ⇒ In g gs) Hin _ Heq in
        match mrscHelper_nil_notBack _ _ _ Hin' with end
      | CGForth c gs ⇒ fun _ ⇒ TCGForth c gs
      end eq_refl).
```

Fig. 10: Big-step multi-result supercompilation

5.3 MRSC Correctness

The next theorem is the main result in this article. Its proof directly relies on the intermediate theorems evalGraph_evalCfgTree and graph2tree_mrsc_In_buildCfgTrees, and also on the key assumption *evalCfg_evalCfgTree_equiv*. The last assumption is the main task left to the user of the approach to verify individually in each particular case.

Theorem mrsc_correct: $\forall$ *env g c*, In *g* (mrsc *c*) $\rightarrow$
EvalEquiv (evalGraph *env g*) (*evalCfg env c*).

6 Current Status and Future Work

The main disadvantage of the current approach is that it has not been field-tested yet on specific supercompilers. We have actually started to build a simple supercompiler for SLL, a basic first-order functional language often used (under different names, and with small variations) in many works on supercompilation [9,17]. Although the verification of this supercompiler – using the proposed approach – is not complete, it has already pointed to some improvements. One of these improvements is already present – evaluation functions take an environment as input (and some return a modified environment). Environment-based evaluation (of trees/graphs of configurations) is useful in typical supercompilers for at least two reasons:

- Contractions in the tree/graph often take the form of patterns, which bind variables inside the corresponding subtree. A successful pattern match will supply values for these bound variables. Environment-based interpreters are a well-known and well-working approach to keep track of such new bindings during the evaluation of subexpressions.
- Generalization is often represented in the tree/graph with let-expressions (or something working in a similar way), whose evaluation by the tree interpreter also involves passing new bindings for the evaluation of sub-trees. Moreover, it is difficult to pass such bindings using a substitution operation, as we bind the let-bound variable not to a configuration, but to a computation, which may yield the value of the corresponding subtree. Environment-based evaluation appears easier to use in this case.

The introduction of evaluation environments as an abstract data type has made impossible to provide general proofs of monotonicity for the tree and for the graph interpreter. Such monotonicity properties can still be very useful, for example when the user proves equivalence between the tree interpreter and the reference interpreter of configurations. We plan to try to recover these general proofs in the future, by postulating some user-defined ordering of environments, and using it to formulate monotonicity requirements for the language-specific building blocks of the tree interpreter.

Another limitation made apparent by the SLL supercompiler involves the precise definition of configuration equality, which is used in several places (in the definition of membership inside our representation of tree sets; in the required properties of the folding primitive, etc.). Currently we use strict syntactic equality, but this may prove too restrictive in many practical cases. For example, if the configuration can bind variables, alpha-equivalence may be a much more useful notion of equality. We plan to fix this deficiency by introducing an abstract configuration equality relation, and basing other definitions on this user-supplied relation instead of the built-in syntactic equality.

We use a general assumption, that if we unfold a graph resulting from supercompilation into a tree, this tree must be among the trees generated by multi-result driving alone. The commutativity condition on configurations produced by folding is imposed exactly in order to make a general proof of this assumption. Both the general assumption and the folding requirement appear to be satisfied in typical cases (such as renaming-based folding, or folding only identical configurations). There are cases, however, where both the assumption and the condition do not hold. Consider folding based on substitution instead of renaming (take some unspecified functional language)[3]. In this case we can have a path fragment in the tree:

$$\ldots f(x) \longrightarrow \ldots \longrightarrow f(5)$$

where folding is possible as there is a substitution converting $f(x)$ into $f(5)$. Assume further that f is defined by pattern matching on its argument. In this case the tree unfolded from the graph cannot be produced by driving alone, because:

- driving $f(5)$ will proceed with a deterministic reduction, giving rise to a single subtree;
- the graph at node $f(x)$ will have 2 subgraphs corresponding to the 2 clauses in the definition of f (assuming Peano representation of natural numbers). Making a copy of this graph will give 2 subtrees at node $f(5)$ as well.

Note, however, that we can achieve similar effect with a suitable combination of generalization and renaming-based folding. The relevant path fragment in this case will be:

$$\ldots f(x) \longrightarrow \ldots \longrightarrow f(5) \longrightarrow \texttt{let}\ y = 5\ \texttt{in}\ f(y) \longrightarrow f(y)$$

Here folding by renaming from $f(x)$ to $f(y)$ is possible. So, ruling out folding by substitution does not lead to an important loss of generality. It remains to be seen if there are other useful kinds of folding ruled out by our assumptions, and, independently, if we can relax the properties required of the folding primitive, while keeping the overall separation of concerns achieved by the current approach.

Another possibility for future improvements concerns the requirement of semantics preservation for the tree interpreter (*evalCfg_evalCfgTree_equiv*). Recall, that this assumption must be verified separately for each supercompiler. This verification step will likely be the most complicated one for specific implementations based on the current approach. So it is interesting to try to find simpler sufficient conditions for the tree interpreter, which can replace this requirement.

Finally, note that we completely omit any formalization of residualization (converting the graph of configurations back into a program in the object language). To complete the proof of correctness of a specific supercompiler, the user must provide a separate proof for the correctness of residualization. Still,

[3] Example suggested by Ilya Klyuchnikov

the proposed approach may offer some help: as we have established the equivalence in the general case between the tree and the graph interpreter (for trees produced by unfolding a graph), it suffices for the user to make a specific proof of equivalence between the residualized program and the tree interpreter – which can be slightly simpler that equivalence w.r.t. the graph interpreter, as folding has no impact on the tree interpreter.

An interesting long-term goal would be to try to apply a similar approach for modular verification to other generalizations of classical supercompilation, such as distillation [5].

7 Related Work

Multi-result supercompilation was introduced by Klyuchnikov et al. [10] and more formally described in later work by the same authors [11], as a generalization of classical [18, 19] and non-deterministic supercompilation. Already in the second work on MRSC, there is a clear separation between the high-level method of multi-result supercompilation, which can be described in a completely language-neutral way, and the set of language-specific basic operations needed to obtain a complete working supercompiler. The recent Agda formalization of "big-step" MRSC [4] is based on an even simpler set of basic operations encapsulating the language-specific parts of the supercompiler. Our work is directly based on this Agda formalization, with some changes in the treatment of folding necessitated by our different goals. We do not use a compact representation for the set of graphs produced by MRSC, but reuse the same idea to represent the set of trees obtained by multi-result driving. It should be easy to merge the 2 formalizations for use cases that may need both an efficient way to represent and manipulate the result set of MRSC and a setting for verifying the correctness of these results.

A similar generic framework for implementing and verifying classical-style supercompilers has been proposed by the author [13]. The current work can be seen as extending that framework to cover the case of multi-result supercompilation. The current formalization is actually simpler, despite the fact that it covers a more general method. This is partly due to the inherent simplicity of MRSC itself, and also a result of incremental improvements based on experience with the previous framework. In particular, we hope the current approach will provide better treatment for generalization. Our stronger assumption of tree-interpreter semantics preservation permits us to have an unified general proof of both soundness and completeness of MRSC, while [13] deals only with soundness.

Taking a wider perspective – about supercompilation and related techniques in general – there are numerous works describing specific supercompilers, including correctness proofs; too many to attempt to enumerate them here. There are also some more general approaches about establishing supercompiler correctness, which are not tied to specific implementations. Sand's improvement theorem [16], for example, gives a general technique for proving semantics preservation of dif-

ferent program transformations, but only for the case of a functional language used as input.

There exist also a few (mostly) language-neutral descriptions of classical supercompilation as a general technique. Jones presents an abstract formulation of driving [6], with only a small number of assumptions about the object language. Still, some of these assumptions seem geared towards simple imperative or first-order tail-recursive functional languages. Also, termination and generalization are not treated there. Klimov [7] covers the complete supercompilation process, and proves a number of interesting high-level properties. To achieve these results, Klimov assumes a specific object language (first-order functional) and data domain. It seems feasible, however, to generalize this approach by abstracting from the details of the object language.

8 Conclusions

We have described the current state of a general approach for modular verification of arbitrary multi-result supercompilers. The main correctness property we are after is preservation of object language semantics by the supercompiler. One key observation that enables our modular approach is that MRSC can be defined in terms of a reusable top-level algorithm + a set of independent building blocks, which must be implemented anew for each specific supercompiler. We propose to apply a similar kind of modularity for the verification of correctness: general reusable theorems concerning the top-level algorithm, which rely on a set of smaller independent properties concerning the building blocks (driving, folding, ...). Only the latter set of properties must be verified from scratch in each case, the general theorems can be reused. Another, more specific idea concerning verification modularization is to consider the unfolding of MRSC-produced graphs of configurations back into trees of configurations. If we can show (as we do, under certain assumptions), that the unfolded graph will always belong to the set of trees produced by driving (+ generalization) alone, we can then ignore completely graphs and folding during the verification process. What is needed in this case is to only verify semantics preservation for the set of trees produced by multi-result driving.

We stress again, that although the current implementation of the approach is inside a proof assistant (Coq), and we speak of formal verification, the approach can be equally useful for verification by testing. Most modern languages already feature property-based testing tools mostly inspired by the Haskell QuickCheck library [2]. The set of correctness assumptions we have identified is perfectly suitable as a starting point of such property-based testing. So, even without doing formal computer-checked proofs, implementers of multi-result supercompilers can use the proposed approach to gain confidence in the correctness of their software.

We must still warn that we are reporting the current state of a work in progress. As we test the approach on specific supercompilers, we shall likely find

further opportunities for improving the framework and making it more easily applicable.

Acknowledgments The author would like to thank Sergei Romanenko and Ilya Klyuchnikov for the insightful discussions related to the topic of this article.

References

1. Chlipala, A.: Certified Programming with Dependent Types. MIT Press (2013), http://adam.chlipala.net/cpdt/
2. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Odersky, M., Wadler, P. (eds.) ICFP. pp. 268–279. ACM (2000)
3. Coquand, T.: About Brouwer's fan theorem. Revue Internationale de Philosophie 230, 483–489 (2004)
4. Grechanik, S.A., Klyuchnikov, I.G., Romanenko, S.A.: Staged multi-result supercompilation: filtering before producing. preprint 70, Keldysh Institute (2013)
5. Hamilton, G.W.: Distillation: extracting the essence of programs. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 61–70. ACM (2007)
6. Jones, N.D.: The essence of program transformation by partial evaluation and driving. In: Bjoner, D., Broy, M., Zamulin, A. (eds.) Perspectives of System Informatics'99. Lecture Notes in Computer Science, vol. 1755, pp. 62–79. Springer Berlin Heidelberg (2000)
7. Klimov, A.V.: A program specialization relation based on supercompilation and its properties. In: Turchin, V. (ed.) International Workshop on Metacomputation in Russia, META 2008 (2008)
8. Klimov, A.V., Klyuchnikov, I.G., Romanenko, S.A.: Automatic verification of counter systems via domain-specific multi-result supercompilation. In: Klimov, A.V., Romanenko, S.A. (eds.) Proceedings of the Third International Workshop on Metacomputation (META 2012). pp. 112–141 (2012)
9. Klyuchnikov, I.: The ideas and methods of supercompilation. Practice of Functional Programming (7) (2011), in Russian
10. Klyuchnikov, I., Romanenko, S.A.: Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In: Clarke, E.M., Virbitskaite, I., Voronkov, A. (eds.) Ershov Memorial Conference. Lecture Notes in Computer Science, vol. 7162, pp. 210–226. Springer (2011)
11. Klyuchnikov, I.G., Romanenko, S.A.: Formalizing and implementing multi-result supercompilation. In: Klimov, A.V., Romanenko, S.A. (eds.) Proceedings of the Third International Workshop on Metacomputation (META 2012). pp. 142–164 (2012)
12. Krustev, D.: A simple supercompiler formally verified in Coq. In: Nemytykh, A.P. (ed.) Proceedings of the Second International Workshop on Metacomputation in Russia (META 2010). pp. 102–127 (2010)
13. Krustev, D.: Towards a framework for building formally verified supercompilers in Coq. In: Loidl, H.W., Peña, R. (eds.) Trends in Functional Programming, Lecture Notes in Computer Science, vol. 7829, pp. 133–148. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-40447-4_9

14. Mendel-Gleason, G.: Types and verification for infinite state systems. PhD thesis, Dublin City University, Dublin, Ireland (2011)
15. Picard, C.: Représentation coinductive des graphes. These, Université Paul Sabatier - Toulouse III (Jun 2012), http://tel.archives-ouvertes.fr/tel-00862507
16. Sands, D.: Proving the correctness of recursion-based automatic program transformations. Theor. Comput. Sci. 167(1&2), 193–233 (1996)
17. Sørensen, M.H.: Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation. Master's thesis, Københavns Universitet, Datalogisk Institut (1994)
18. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Hatcliff, J., Mogensen, T., Thiemann, P. (eds.) Partial Evaluation: Practice and Theory. Lecture Notes in Computer Science, vol. 1706, pp. 246–270. Springer-Verlag (1999)
19. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (July 1986)
20. Vytiniotis, D., Coquand, T., Wahlstedt, D.: Stop when you are almost-full: Adventures in constructive termination. In: Beringer, L., Felty, A. (eds.) Interactive Theorem Proving. Lecture Notes in Computer Science, vol. 7406, pp. 250–265. Springer Berlin Heidelberg (2012)

A Method of a Proof of Observational Equivalence of Processes

Andrew M. Mironov

The Institute of Informatics Problems of the Russian Academy of Sciences
amironov66@gmail.com

Abstract. In the paper we introduce a concept of a process with message passing and present a method of checking observational equivalence of processes with message passing. The presented method is illustrated by an example of a verification of sliding window protocol.

Keywords: processes, message passing, verification, sliding window protocol

1 Introduction

The problem of formal representation and verification of discrete processes is one of the most important problems in computer science. There are several approaches to this problem, the main of them are: CCS and π-calculus [1], [2], CSP and its generalizations [3], temporal logic and model checking [4], Petri nets [5], process algebras [6], communicating finite-state machines [7].

In the present paper we introduce a new model of discrete processes, which is a synthesis of Milner's model of processes [1] and the model of communicating finite-state machines [7]. Discrete processes are represented in our model as graphs, edges of which are labelled by operators. These operators consist of internal actions and communication actions. Proofs of correctness of processes are represented by sets of formulas, associated with pairs of states of analyzed processes. This method of verification of processes is a synthesis of Milner's approach related on the concept of an observational equivalence [1] and Floyd's inductive assertion method [8]. For a simplification of an analysis of processes we introduce a simplification operation on processes. With use this operation it is possible to reduce a complexity of verification of processes. We illustrate an advantage of the proposed model and the verification method on the example of verification of a two-way sliding window protocol.

2 Motivation, advantages of the proposed approach and its comparison with other works

2.1 Motivation of the proposed approach

The main disadvantage of modern methods of verification of discrete processe is their large complexity. More precisely,

- the main disadvantage of verification methods based on model checking approach is a high computational complexity related to the state explosion problem, and
- disadvantages of methods based on theorem proving approach are related with a high complexity of construction of corresponging theorems and their proofs, and also with an understanding of these proofs.

For example, in recent paper [9] a complete presentation of proofs of theorems related to verification of two-way sliding window protocol takes a few dozen pages of a complex mathematical text.

The main motivation for the proposed approach to modeling and verification of discrete systems by checking of observational equivalence of corresponded processes with message passing is to simplify and make more obvious the following aspects of modeling and analysis of discrete systems: representation of mathematical models of analyzed systems, construction of proofs of correctness of the systems, and understanding of these proofs by any who is not a strong expert in the mathematical theory of verification of discrete systems.

2.2 Advantages of the proposed approach

The proposed mathematical model of processes with message passing allows to construct such mathematical models of analysed systems that are very similar to an original description of these systems on any imperative programming language. In section 10 we give an example of such model that corresponds to a C-program describing a sliding window protocol using go back n (the program was taken from book [10], section 3.4.2).

The main advantage of the proposed approach is a possibility to use a simplification operation of models of analyzed systems, that allows essentially simplify the problem of verification of these models. In section 10 we present a result of such simplification for the above model of a sliding window protocol: this model can be simplified to a model with only one state. It should be noted also that the simplified models allow more clearly understand main features of analyzed systems, and facilitate a construction of correctness proofs for analyzed systems.

If an analyzed property of a system has the form of a behavior which is described by some process, for example, in the case when

- an analyzed system is a network protocol, and
- a property of this system is a description of an external behavior of this protocol (related to its interaction with a higher-level protocol)

then a proof of a correctness of such system in this model is a set of formulas associated with pairs of states, the first of which is a state of the analyzed system, and the second is a state of a a process which describes a property of the analyzed system.

In section 10 we give an example of such proof, which is a small set of simple formulas. These formulas can be naturally derived from a simplified model of an analyzed protocol.

Another advantage of the proposed approach is a possibility to verify systems with unbounded sets of states. One of examples of such systems is the above sliding window protocol using go back n.

2.3 Comparison with other works

In this section we present an overview of papers related to verification of message passing systems, which are most relevant to the present paper.

The paper [9] deals with modeling and manual verification in the process algebraic language μCRL. Authors use the theorem prover PVS to formalize and to mechanically prove the correctness of a protocol using selective repeat (a C-program describing this protocol is presented in section 3.4.3 of the book [10]). The main disadvantage of this work is a large complexity of proofs of theorems related to verification of this protocol. This protocol can be verified more simply with use of the approach proposed in the present paper.

There are a lot of works related to verification of systems with message passing based on temporal logic and model checking approach. Most relevant ones to the present paper are [11], [12], [13], [14], [15], [16], [17]. The most deficiency of all of them is restricted abilities: these methods allow verify only finite state systems.

Among other approaches it should be noted approaches with use of first order logic and assertional verification: [18], [19], and approaches with use of process algebra: [20], [21], [22], [23]. The most deficiency of these approaches is a high complexity of construction of proofs of correctness of analyzed systems.

3 Auxiliary concepts

3.1 Terms

We assume that there are given a set $\mathcal{X}$ of **variables**, a set $\mathcal{D}$ of **values**, a set $\mathcal{C}$ of **constants**, and a set $\mathcal{F}$ of **function symbols**. Any constant from $\mathcal{C}$ is interpreted by a value from $\mathcal{D}$, and any function symbol from $\mathcal{F}$ is interpreted by an operation on $\mathcal{D}$.

We assume that $\mathcal{C}$ contains constants 0 and 1, and $\mathcal{F}$ contains boolean function symbols $\wedge, \vee, \rightarrow$, which correspond to standard boolean operations on $\{0, 1\}$.

The set $\mathcal{E}$ of **terms** is defined in the standard way. Variables and constants are terms. Other terms have the form $f(e_1, \ldots, e_n)$, where $f \in \mathcal{F}$, and $e_1, \ldots, e_n$ are terms. For each $e \in \mathcal{E}$ a set of all variables occurring in e is denoted by X_e.

If $X \subseteq \mathcal{X}$, then a **valuation** of variables of X is a correspondence ξ, that associates each variable $x \in X$ with a value $x^\xi \in \mathcal{D}$. We denote by the record $X^\bullet$ the set of all valuations of variables from X. For each $e \in \mathcal{E}$, each $X \supseteq X_e$ and each $\xi \in X^\bullet$ the record e^ξ denotes an object called a **value** of e on ξ and defined in the standard way. We assume that terms e_1 and e_2 are equal iff $\forall \xi \in (X_{e_1} \cup X_{e_2})^\bullet \;\; e_1^\xi = e_2^\xi$.

A term e is a **formula** if $\forall \xi \in X_e^\bullet$ the value e^ξ is 0 or 1. The set of all formulas is denoted by $\mathcal{B}$. The symbols $\top$ and $\bot$ denote true and false formula

respectively. We shall write formulas of the form $\wedge(b_1, b_2)$, $\vee(b_1, b_2)$, etc. in a more familiar form $b_1 \wedge b_2$, $b_1 \vee b_2$, etc.

3.2 Atomic operators

We assume that there is given a set $\mathcal{N}$, whose elements are considered as names of objects that can be sent or received by processes.

An **atomic operator (AO)** is an object o of one of three forms presented below. Each pair (o, ξ), where o is an AO, and ξ is a valuation of variables occurred in o, corresponds to an action o^ξ, informally defined below.

1. An **input** is an AO of the form $\alpha?x$, where $\alpha \in \mathcal{N}$ and $x \in \mathcal{X}$. An action $(\alpha?x)^\xi$ is a receiving from another process an object named α, with a message attached to this object, this message is assigned to the variable x.
2. An **output** is an AO of the form $\alpha!e$, where $\alpha \in \mathcal{N}$ and $e \in \mathcal{E}$. An action $(\alpha!e)^\xi$ is a sending to another process an object named α, to which a message e^ξ is attached.
3. An **assignment** is an AO of the form $x := e$, where $x \in \mathcal{X}$, $e \in \mathcal{E}$. An action $(x := e)^\xi$ is an assigning the variable x with the value e^ξ.

Below we use the following notations.

- For each AO o the record X_o denotes the set of all variables occurred in o.
- If $e \in \mathcal{E}$, and o is an assignment, then the record $o(e)$ denotes a term defined as follows: let o has the form $(x := e')$, then $o(e)$ is obtained from e by a replacement of all occurrences of the variable x by the term e'.
- If o is an assignment, and $\xi \in X^\bullet$, where $X_o \subseteq X \subseteq \mathcal{X}$, then the record $\xi \cdot o$ denotes a valuation from $X^\bullet$, defined as follows: let $o = (x := e)$, then $x^{\xi \cdot o} = e^\xi$ and $\forall\, y \in X \setminus \{x\}\ \ y^{\xi \cdot o} = y^\xi$.

It is easy to prove that if o is an assignment and $e \in \mathcal{E}$, then for each $\xi \in X^\bullet$, where $X_o \cup X_e \subseteq X \subseteq \mathcal{X}$, the equality $o(e)^\xi = e^{\xi \cdot o}$ holds. This equality is proved by an induction on the structure of the term e.

3.3 Operators

An **operator** is a record O of the form $b\,[o_1, \ldots, o_n]$, where b is a formula called a **precondition** of O (this formula will be denoted as $\langle O \rangle$), and $o_1, \ldots, o_n$ is a sequence of AOs (this sequence will be denoted as $[O]$), among which there is at most one input or output. The sequence $[O]$ may be empty ($[\,]$).

If $[O]$ contains an input (or an output) then O is called an **input operator** (or an **output operator**), and in this case the record N_O denotes a name occurred in O. If $[O]$ does not contain inputs and outputs, then we call O an **internal operator**.

If $\langle O \rangle = \top$, then such precondition can be omitted in a notation of O.

Below we use the following notations.

1. For each operator O a set of all variables occurred in O is denoted by X_O.

2. If O is an operator, and $b \in \mathcal{B}$, then the record $O \cdot b$ denotes an object, which either is a formula or is not defined. This object is defined recursively as follows. If $[O]$ empty, then $O \cdot b \stackrel{\text{def}}{=} \langle O \rangle \wedge b$. If $[O] = o_1, \ldots, o_n$, where $n \geq 1$, then we shall denote by the record $O \setminus o_n$ an operator obtained from O by a removing of its last AO, and
   - if $o_n = \alpha?x$, then $O \cdot b \stackrel{\text{def}}{=} (O \setminus o_n) \cdot b$, if $x \notin X_b$, and is undefined otherwise
   - if $o_n = \alpha!e$, then $O \cdot b \stackrel{\text{def}}{=} (O \setminus o_n) \cdot b$
   - if $o_n = (x := e)$, then $O \cdot b \stackrel{\text{def}}{=} (O \setminus o_n) \cdot o_n(b)$.
3. If O is an internal operator, and $\xi \in X^\bullet$, where $X_O \subseteq X \subseteq \mathcal{X}$, then the record $\xi \cdot O$ denotes a valuation from $X^\bullet$, defined as follows: if $[O]$ is empty, then $\xi \cdot O \stackrel{\text{def}}{=} \xi$, and if $[O] = o_1, \ldots, o_n$, where $n \geq 1$, then $\xi \cdot O \stackrel{\text{def}}{=} (\xi \cdot (O \setminus o_n)) \cdot o_n$.

It is easy to prove that if O is internal and $b \in \mathcal{B}$, then for each $\xi \in X^\bullet$, where $X_O \cup X_b \subseteq X \subseteq \mathcal{X}$, such that $\langle O \rangle^\xi = 1$, the equality $(O \cdot b)^\xi = b^{\xi \cdot O}$ holds. This equality is proved by an induction on a lenght of $[O]$.

3.4 Concatenation of operators

Let O_1 and O_2 be operators, and at least one of them is internal.

A **concatenation** of O_1 and O_2 is an object denoted by the record $O_1 \cdot O_2$, that either is operator or is undefined. This object is defined iff $O_1 \cdot \langle O_2 \rangle$ is defined, and in this case $O_1 \cdot O_2 \stackrel{\text{def}}{=} (O_1 \cdot \langle O_2 \rangle)[[O_1], [O_2]]$. It is easy to prove that

- if operators O_1, O_2 and formula b are such that objects in both sides of the equality $(O_1 \cdot O_2) \cdot b = O_1 \cdot (O_2 \cdot b)$ are defined, then this equality holds, and
- if operators O_1, O_2, O_3 are such that all objects in both sides of the equality $(O_1 \cdot O_2) \cdot O_3 = O_1 \cdot (O_2 \cdot O_3)$ are defined, then this equality holds.

4 Processes with a message passing

4.1 A concept of a process with a message passing

A **process with a message passing** (also called more briefly a **process**) is a 4-tuple P of the form

$$P = (S_P, s_P^0, T_P, I_P) \tag{1}$$

components of which have the following meanings.

- S_P is a set of **states** of the process P.
- $s_P^0 \in S_P$ is an **initial state** of the process P.
- T_P is a set of **transitions** of the process P, each transition from T_P has the form $s_1 \stackrel{O}{\to} s_2$, where $s_1, s_2 \in S_P$ and O is an operator.
- $I_P \in \mathcal{B} \setminus \{\bot\}$ is a **precondition** of the process P.

A transition $s_1 \stackrel{O}{\to} s_2$ is called an **input**, an **output**, or an **internal** transition, if O is an input operator, an output operator, or an internal operator, respectively.

For each process P

- the record X_P denotes the set consisting of
  - all variables occurred in any of the transitions from T_P, or in I_P, and
  - a variable at_P, which is not occurred in I_P, and in transitions from T_P, the set of values of at_P is S_P
- the record $\langle P \rangle$ denotes the formula $(at_P = s_P^0) \wedge I_P$.

For each transition $t \in T_P$ the records O_t, $\langle t \rangle$, $start(t)$ and $end(t)$ denote an operator, a formula and states defined as follows: if t has the from $s_1 \stackrel{O}{\to} s_2$, then

$$O_t \stackrel{\text{def}}{=} O, \quad \langle t \rangle \stackrel{\text{def}}{=} (at_P = s_1) \wedge \langle O \rangle, \quad start(t) \stackrel{\text{def}}{=} s_1, \quad end(t) \stackrel{\text{def}}{=} s_2.$$

If t is an input or an output, then the record N_t denotes the name N_{O_t}.

A set X_P^s of **essential variables** of P is a smallest (w.r.t. inclusion) set satisfying the following conditions.

- X_P^s contains all variables contained in preconditions and outputs in operators O_t, where $t \in T_P$.
- If P contains an AO $x := e$ and $x \in X_P^s$, then X_P^s contains all variables occurred in e.

A process P is associated with a graph denoted by the same symbol P. Vertices of this graph are states of P, and its edges correspond to transitions of P: each transition $s_1 \stackrel{O}{\to} s_2$ corresponds to an edge from s_1 to s_2 with a label O.

4.2 Actions of processes

An **action of a process** (or, briefly, an **action**) is a record of one of the following three forms.

- $\alpha?d$, where $\alpha \in \mathcal{N}$ and $d \in \mathcal{D}$. An action of this form is called a **receiving** of an object named α with the attached message d.
- $\alpha!d$, where $\alpha \in \mathcal{N}$ and $d \in \mathcal{D}$. An action of this form is called a **sending** of an object named α with the attached message d.
- τ. An action of this form is called a **silent action**.

A set of all actions is denoted by $\mathcal{A}$.

4.3 An execution of a process

An **execution** of a process (1) is a walk on the graph P starting from s_P^0, with an execution of AOs occurred in labels of traversed edges. At each step $i \geq 0$ of this walk there is defined a current state $s_i \in S_P$ and a current valuation $\xi_i \in X_P^\bullet$. We assume that $s_0 = s_P^0$, $\langle P \rangle^{\xi_0} = 1$, and for each step i of this walk $at_P^{\xi_i} = s_i$.

An execution of P on step i is described informally as follows. If there is no transitions in T_P starting at s_i, then P terminates, otherwise

- P selects a transition $t \in T_P$, such that $\langle t \rangle^{\xi_i} = 1$, and if t is an input or an output, then at the current moment P can receive or send respectively an object named N_t (i.e. at the same moment there is another process that can send to P or receive from P respectively an object named N_t). If there is no such transition, then P suspends until at least one such transition will appear, and after resumption its execution P selects one of such transitions,
- after a sequential execution of all AOs occurred in the operator O_t of the selected transition t, P moves to the state $end(t)$.

An execution of each AO o occurred in $[O_t]$ consists of a performing of an action $a \in \mathcal{A}$ and a replacement the current valuation ξ on a valuation ξ', which is considered as a current valuation after an execution of the AO o. An execution of an AO o is as follows:

- if $o = \alpha?x$, then P performs an action of the form $\alpha?d$, and $x^{\xi'} \stackrel{\text{def}}{=} d$, $\forall\, y \in X_P \setminus \{x\} \quad y^{\xi'} \stackrel{\text{def}}{=} y^\xi$
- if $o = \alpha!e$, then P performs the action $\alpha!(e^\xi)$, and $\xi' \stackrel{\text{def}}{=} \xi$
- if $o = (x := e)$, then P performs τ, and $x^{\xi'} \stackrel{\text{def}}{=} e^\xi$, $\forall\, y \in X_P \setminus \{x\} \quad y^{\xi'} \stackrel{\text{def}}{=} y^\xi$.

5 Operations on processes

In this section we define some operations on processes which can be used for a construction of complex processes from simpler ones. These operations are generalizations of corresponded operations on processes defined in Milners's Calculus of Communicating Systems [1].

5.1 Parallel composition

The operation of parallel composition is used for building processes, composed of several communicating subprocesses.

Let $P_i = (S_i, s_i^0, T_i, I_i) \quad (i = 1, 2)$ be processes, such that $S_1 \cap S_2 = \emptyset$ and $X_{P_1} \cap X_{P_2} = \emptyset$. A **parallel composition** of P_1 and P_2 is a process $P = (S, s^0, T, I)$, where

$$S \stackrel{\text{def}}{=} S_1 \times S_2, \quad s^0 \stackrel{\text{def}}{=} (s_1^0, s_2^0), \quad I \stackrel{\text{def}}{=} I_1 \wedge I_2$$

and T consists of the following transitions:

- for each transition $s_1 \stackrel{O}{\to} s_1'$ of the process P_1, and each state s of P_2 the process P has the transition $(s_1, s) \stackrel{O}{\to} (s_1', s)$
- for each transition $s_2 \stackrel{O}{\to} s_2'$ of the process P_2, and each state s of the process P_1 the process P has the transition $(s, s_2) \stackrel{O}{\to} (s, s_2')$
- for each pair of transition of the form $\begin{cases} s_1 \stackrel{O_1}{\to} s_1' \in T_{P_1} \\ s_2 \stackrel{O_2}{\to} s_2' \in T_{P_2} \end{cases}$ where one of the operators O_1, O_2 has the form $(O_1' \cdot [\alpha?x]) \cdot O_1''$, and another operator has the form $(O_2' \cdot [\alpha!e]) \cdot O_2''$, the process P has the transition $(s_1, s_2) \stackrel{O}{\to} (s_1', s_2')$, where $\langle O \rangle = \langle O_1 \rangle \wedge \langle O_2 \rangle$ and $[O] = \Big((O_1' \cdot O_2') \cdot [x := e]\Big) \cdot (O_1'' \cdot O_2'')$.

A parallel composition of P_1 and P_2 is denoted by the record $P_1 \,|\, P_2$.

If $S_1 \cap S_2 \neq \emptyset$ or $X_{P_1} \cap X_{P_2} \neq \emptyset$, then before a construction of the process $P_1 \,|\, P_2$ it is necessary to replace states and variables occuring in both processes on new states or variables respectively.

For any tuple $P_1, P_2, \ldots, P_n$ of processes their parallel composition $P_1 \,|\, \ldots \,|\, P_n$ is defined as the process $((P_1 \,|\, P_2) \,|\, \ldots) \,|\, P_n$.

5.2 Restriction

Let $P = (S, s^0, T, I)$ be a process, and L be a subset of the set $\mathbf{N}$.

A **restriction** of P with respect to L is the process $P \setminus L = (S, s^0, T', I)$ which is obtained from P by removing of those transitions that have labels with the names from L, i.e.

$$T' \stackrel{\text{def}}{=} \Big\{ (\, s \xrightarrow{\quad O \quad} s' \,) \in R \,\Big|\, [O] = [\,], \text{ or } N_O \notin L \Big\}$$

6 Renaming

The last operation is called a **renaming**: for any mapping $f : \mathcal{N} \to \mathcal{N}$ and any process P the record $P[f]$ denotes a process which is called a renaming of P and is obtained from P by changing of names occurred in P: any name α occurred in P is changed on $f(\alpha)$.

If the mapping f acts non-identically only on the names $\alpha_1, \ldots, \alpha_n$, and maps them to the names $\beta_1, \ldots, \beta_n$ respectively, then the process $P[f]$ can be denoted also as $P[\beta_1/\alpha_1, \ldots, \beta_n/\alpha_n]$.

6.1 An example of a process defined with use of parallel composition and restriction

In this subsection we describe a process which is defined with use of the above operations. This process is an implementation of a distributed algorithm of separation of sets. The problem of separation of sets has the following form. Let U, V be a pair of finite disjoint sets, with each element $x \in U \cup V$ is associated with a number $weight(x)$, called a **weight** of this element. It is need to convert this pair to a pair of sets U', V', such that

- $|U| = |U'|, \quad |V| = |V'|$
  (for each finite set M the notation $|M|$ denotes a number of elements in M)
- $\forall\, u \in U', \forall\, v \in V' \quad weight(u) \leq weight(v)$.

Below we shall call U and V as the left set and the right set, respectively.

The problem of separation of sets can be solved by an execution of several sessions of exchange elements between these sets. Each session consists of the following actions:

- find an element mx with a maximum weight in the left set
- find an element mn with minimum weight in the right set
- transfer
  - mx from the left set to the right set, and
  - mn from the right set to the left set.

To implement this idea it is proposed a distributed algorithm, defined as a process of the form

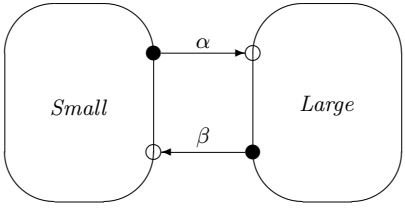
$$(Small \mid Large) \setminus \{\alpha, \beta\} \tag{2}$$

where

- the process $Small$ executes operations associated with the left set, and
- the process $Large$ executes operations associated with the right set.

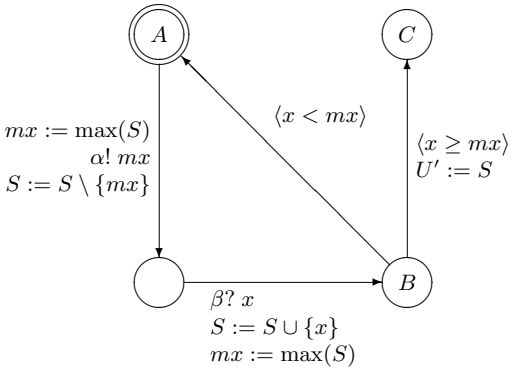
The restriction of the actions with names α and β in (2) means that a transmission of objects with names α and β can be executed only between the subprocesses $Small$ and $Large$, i.e. such objects can not be transmitted outside the process (2).

A flow graph (i.e. a relation between components) corresponded to this process has the form



Below we shall use the following notations: for each subset $W \subseteq U \cup V$ the records $\max(W)$ and $\min(W)$ denote an element of W with maximum and minimum weight, respectively. A similar meaning have the records $\max(W)$ and $\min(W)$, where W is a a variable whose values are subsets of $U \cup V$.

The process $Small$ has the following form:

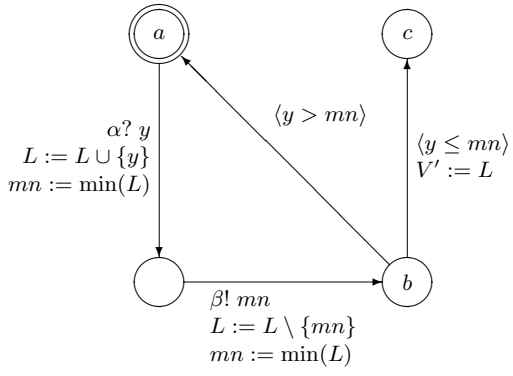


(3)

(a double circle denotes an initial state).

An initial condition of the process *Small* is $(S = U)$.

The process *Large* has the following form:



(4)

An initial condition of the process *Large* is $(L = V)$.

7 Realizations of processes

7.1 Realizations of AOs and sequences of AOs

A **realization of an AO** o is a triple (ξ, a, ξ'), such that

- $\xi, \xi' \in X^\bullet$, where $X_o \subseteq X \subseteq \mathcal{X}$, and $a \in \mathcal{A}$
- if $o = \alpha?x$, then $a = \alpha?(x^{\xi'})$ and $\forall\, y \in X \setminus \{x\} \quad y^{\xi'} = y^{\xi}$
- if $o = \alpha!e$, then $a = \alpha!(e^{\xi})$ and $\xi' = \xi$
- if $o = (x := e)$, then $a = \tau$ and $\xi' = \xi \cdot o$.

Let $o_1, \ldots, o_n$ be a sequence of AOs which contains at most one input or output. A **realization of** $o_1, \ldots, o_n$ is a triple (ξ, a, ξ'), such that

- $\xi, \xi' \in X^\bullet$, where $X \subseteq \mathcal{X}$ and $a \in \mathcal{A}$

– if $n = 0$, then $\xi' = \xi$ and $a = \tau$, otherwise there exists a sequence

$$(\xi_0, a_1, \xi_1),\ (\xi_1, a_2, \xi_2),\ \ldots, (\xi_{n-1}, a_n, \xi_n) \tag{5}$$

where $\xi_0 = \xi$, $\xi_n = \xi'$, $\forall\, i = 1, \ldots, n$ (ξ_{i-1}, a_i, ξ_i) is a realization of o_i, and $a = \tau$, if each a_i in (5) is equal to τ, otherwise a coincides with that a_i, which is different from τ.

7.2 Realization of transitions

Let P be a process of the form (1), and $t \in T_P$.

A **realization of** t is a triple (ξ_1, a, ξ_2), where $\xi_1, \xi_2 \in X_P^\bullet$ and $a \in \mathcal{A}$, such that $\langle t \rangle^{\xi_1} = 1$ and $(\xi_1 \cdot (at_P := end(t)), a, \xi_2)$ is a realization of $[O_t]$.

The following properties hold.

– If a transition t is internal or is an output, then for each $\xi \in X_P^\bullet$, such that $\langle t \rangle^{\xi} = 1$, there exist a unique $\xi' \in X_P^\bullet$ and a unique $a \in \mathcal{A}$, such that (ξ, a, ξ') is a realization of t. We shall denote such ξ' by $\xi \cdot t$.
– If a transition t is an input, then for each $\xi \in X_P^\bullet$, such that $\langle t \rangle^{\xi} = 1$, and each $d \in \mathcal{D}$ there exists a unique $\xi' \in X_P^\bullet$, such that $(\xi, N_t?d, \xi')$ is a realization of t. We shall denote such ξ' by $\xi \cdot t^d$.

7.3 Realizations of processes

A **realization of a process** P is a graph P^r having the following components.

– The set S_P^r of vertices of P^r is the disjoint union $X_P^\bullet \cup \{P^0\}$.
– The set T_P^r of edges of P^r consists of the following edges:
  • for each realization (ξ_1, a, ξ_2) of any $t \in T_P$ the graph P^r has an edge from ξ_1 to ξ_2 with a label a, and
  • for each $\xi \in X_P^\bullet$, such that $\langle P \rangle^{\xi} = 1$, and each edge of P^r from ξ to ξ' with a label a the graph P^r has an edge from P^0 to ξ' with a label a.

We shall use the following notations: for any pair v, v' of vertices of P^r

– the record $v_1 \stackrel{a}{\to} v_2$ denotes an edge from v_1 to v_2 with a label a
– $v \stackrel{\tau^*}{\to} v'$ means that either $v = v'$ or $\exists\, v_0, v_1, \ldots, v_n : \forall\, i = 1, \ldots, n$ the graph P^r has an edge $v_{i-1} \stackrel{\tau}{\to} v_i$, and $v_0 = v$, $v_n = v'$.
– $v \stackrel{\tau^* a \tau^*}{\longrightarrow} v'$ (where $a \in \mathcal{A}$) means that $\exists\, v_1, v_2$: the graph P^r has an edge $v_1 \stackrel{a}{\to} v_2$, and $v \stackrel{\tau^*}{\longrightarrow} v_1$, $v_2 \stackrel{\tau^*}{\longrightarrow} v'$.

8 Observational equivalence of processes

8.1 A concept of observational equivalence of processes

Processes P_1 and P_2 are said to be **observationally equivalent** if P_1^r and P_2^r are observationally equivalent in Milner's sense [1], i.e. there exists $\mu \subseteq S_{P_1}^r \times S_{P_2}^r$, such that

1. $(P_1^0, P_2^0) \in \mu$
2. if $(v_1, v_2) \in \mu$ and $v_1 \stackrel{\tau}{\to} v_1'$, then $\exists\, v_2' : v_2 \stackrel{\tau^*}{\to} v_2'$, $(v_1', v_2') \in \mu$,
   if $(v_1, v_2) \in \mu$ and $v_2 \stackrel{\tau}{\to} v_2'$, then $\exists\, v_1' : v_1 \stackrel{\tau^*}{\to} v_1'$, $(v_1', v_2') \in \mu$
3. if $(v_1, v_2) \in \mu$ and $v_1 \stackrel{a}{\to} v_1'$, where $a \neq \tau$, then $\exists\, v_2' : v_2 \stackrel{\tau^* a \tau^*}{\longrightarrow} v_2'$, $(v_1', v_2') \in \mu$,
   if $(v_1, v_2) \in \mu$ and $v_2 \stackrel{a}{\to} v_2'$, where $a \neq \tau$, then $\exists\, v_1' : v_1 \stackrel{\tau^* a \tau^*}{\longrightarrow} v_1'$, $(v_1', v_2') \in \mu$

The record $P_1 \approx P_2$ means that P_1 and P_2 are observationally equivalent.

A lot of problems related to verification of discrete systems can be reduced to the problem to prove that $P_1 \approx P_2$, where the process P_1 is a model of a system being analyzed, and P_2 is a model of some property of this system. In section 10 we consider an example of a proof that $P_1 \approx P_2$, where P_1 is a model of the sliding window protocol, and P_2 is a model of its external behavior.

8.2 A method of a proof of observational equivalence of processes

In this section we present a method of a proof of observational equivalence of processes. This method is based on theorem 1. To formulate and prove this theorem, we introduce auxiliary concepts and notations.

1. Let P be a process, and $s, s' \in S_P$. A **composite transition (CT)** from s to s' is a sequence T of transitions of P of the form

$$s = s_0 \stackrel{O_1}{\to} s_1, \quad s_1 \stackrel{O_2}{\to} s_2, \quad \ldots \quad s_{n-1} \stackrel{O_n}{\to} s_n = s' \tag{6}$$

   such that there is at most one input or output operator among $O_1, \ldots, O_n$, and there are defined all concatenations in the expression

$$(\ldots(O_1 \cdot O_2) \cdot \ldots) \cdot O_n \tag{7}$$

   Sequence (6) may be empty, in this case $s = s'$. If CT T is not empty and has the form (6), then the record O_T denotes a value of the expression (7). If CT T is empty, then $O_T \stackrel{\text{def}}{=} [\,]$.
   We shall use for CTs the same concepts and notation as for ordinary transitions ($start(T)$, $end(T)$, N_T etc.). A CT T is said to be an input, an output, or an internal iff O_T is an input operator, an output operator, or an internal operator, respectively.
   A concept of a realization of a CT is defined by analogy with the concept of a realization of a transition (see section 7.2). This concept has properties similar to properties of a realization of a transition, in particular:
   (a) if a CT T is internal or is an output, then for each $\xi \in X_P^\bullet$, such that $\langle T \rangle^\xi = 1$, there is a unique $\xi' \in X_P^\bullet$ and a unique $a \in \mathcal{A}$, such that (ξ, a, ξ') is a realization of T, we shall denote such ξ' by the record $\xi \cdot T$
   (b) if a CT T is an input, then for each $\xi \in X_P^\bullet$, such that $\langle T \rangle^\xi = 1$, and each $d \in \mathcal{D}$ there is a unique $\xi' \in X_P^\bullet$, such that $(\xi, N_T?d, \xi')$ is a realization of T, we shall denote such ξ' by the record $\xi \cdot T^d$.

2. If b and b' are formulas, then the record $b \leq b'$ is a brief notation of the proposition that the formula $b \to b'$ is true.
3. If O_1, O_2 are operators, and $b \in \mathcal{B}$, then the record $(O_1, O_2) \cdot b$ denotes a formula defined by a recursive definition presented below. In this definition we use records of the form $O \setminus o$ and $o(b)$, which denote an operator and a formula respectively, defined in section 3.3.
   Let $[O_1] = o_1, \ldots, o_n$ and $[O_2] = o'_1, \ldots, o'_m$, then the formula

$$(O_1, O_2) \cdot b \tag{8}$$

   is defined as follows:
   (a) $\langle O_1 \rangle \wedge \langle O_2 \rangle \wedge b$, if $n = m = 0$
   (b) $(O_1 \setminus o_n, O_2) \cdot o_n(b)$, if o_n is an assignment
   (c) $(O_1, O_2 \setminus o'_m) \cdot o'_m(b)$, if o'_m is an assignment
   (d) $((O_1 \setminus o_n), (O_2 \setminus o'_m)) \cdot b(z/x, z/y)$, if $o_n = \alpha?x,\ o'_m = \alpha?y$, and $b(z/x, z/y)$ is a formula obtained from b replacing all occurrences of x and y on a fresh variable z (i.e. z is not occurred in O_1, O_2 and b)
   (e) $((O_1 \setminus o_n), (O_2 \setminus o'_m)) \cdot ((e_1 = e_2) \wedge b)$, if $o_n = \alpha!e_1$ and $o'_m = \alpha!e_2$
   (f) $\bot$, otherwise.

Theorem 1

Let $P_i = (S_{P_i}, s^0_{P_i}, T_{P_i}, \langle P_i \rangle)\ \ (i = 1, 2)$ be processes such that $S_{P_1} \cap S_{P_2} = \emptyset$ and $X_{P_1} \cap X_{P_2} = \emptyset$. Then $P_1 \approx P_2$, if there exist a set $\{b_{s_1 s_2} \mid s_i \in S_{P_i}\ (i = 1, 2)\}$ of formulas with variables from $(X_{P_1} \cup X_{P_2}) \setminus \{at_{P_1}, at_{P_2}\}$, such that

1. $\langle P_1 \rangle \wedge \langle P_2 \rangle \leq b_{s^0_{P_1} s^0_{P_2}}$
2. $\forall\, (s_1 \stackrel{O}{\to} s'_1) \in T_{P_1},\ \forall\, s_2 \in S_{P_2}$ there exists a set $\{s_2 \stackrel{T_i}{\to} s^i_2 \mid i \in \Im\}$ of CTs of P_2 such that $b_{s_1 s_2} \wedge \langle O \rangle \leq \bigvee\limits_{i \in \Im} (O, O_{T_i}) \cdot b_{s'_1 s^i_2}$
3. $\forall\, (s_2 \stackrel{O}{\to} s'_2) \in T_{P_2},\ \forall\, s_1 \in S_{P_1}$ there exists a set $\{s_1 \stackrel{T_i}{\to} s^i_1 \mid i \in \Im\}$ of CTs of P_1 such that $b_{s_1 s_2} \wedge \langle O \rangle \leq \bigvee\limits_{i \in \Im} (O_{T_i}, O) \cdot b_{s^i_1 s'_2}$

Proof.

Since $X_{P_1} \cap X_{P_2} = \emptyset$, then there is a natural bijection between $X^\bullet_{P_1} \times X^\bullet_{P_2}$ and $(X_{P_1} \cup X_{P_2})^\bullet$. Below we identify these two sets.

We define the relation $\mu \subseteq S^r_{P_1} \times S^r_{P_2}$ as follows:

$$\mu \stackrel{\text{def}}{=} \{(\xi_1, \xi_2) \in X^\bullet_{P_1} \times X^\bullet_{P_2} \mid b^{(\xi_1, \xi_2)}_{at^{\xi_1}_{P_1} at^{\xi_2}_{P_2}} = 1\} \cup \{(P^0_1, P^0_2)\}.$$

We prove that μ satisfies the conditions from section 8.1.

1. The condition $(P^0_1, P^0_2) \in \mu$ follows from the definition of μ.
2. Let $(v_1, v_2) \in \mu$ and $v_1 \stackrel{\tau}{\to} v'_1$. We must prove that

$$\exists\, v'_2 : v_2 \stackrel{\tau^*}{\to} v'_2,\ (v'_1, v'_2) \in \mu \tag{9}$$

We consider separately the cases $v_1 = P_1^0$ and $v_1 \neq P_1^0$.
If $v_1 = P_1^0$, then $v_2 = P_2^0$, and according to definition of the graph P_1^r (section 7.3), $\exists\, \xi_1 \in X^{\bullet}_{P_1} : \langle P_1 \rangle^{\xi_1} = 1$ and the graph P_1^r has the edge $\xi_1 \stackrel{\tau}{\to} \xi_1' = v_1'$, i.e. (ξ_1, τ, ξ_1') is a realization of a transition $s^0_{P_1} \stackrel{O_1}{\to} s_1'$ from T_{P_1}, where O_1 is an internal operator.
According to item 2 in the theorem, there exists a set $\{s^0_{P_2} \stackrel{T_i}{\to} s_2^i \mid i \in \Im\}$ of CTs of process P_2, such that

$$b_{s^0_{P_1} s^0_{P_2}} \wedge \langle O_1 \rangle \leq \bigvee_{i \in \Im} (O_1, O_{T_i}) \cdot b_{s_1' s_2^i} \tag{10}$$

Since $\langle P_2 \rangle \neq \bot$, then $\exists\, \xi_2 \in X^{\bullet}_{P_2}$: $\langle P_2 \rangle^{\xi_2} = 1$, so

$$1 = \langle P_1 \rangle^{\xi_1} \wedge \langle P_2 \rangle^{\xi_2} = (\langle P_1 \rangle \wedge \langle P_2 \rangle)^{(\xi_1, \xi_2)} \leq b^{(\xi_1, \xi_2)}_{s^0_{P_1} s^0_{P_2}} \tag{11}$$

(the last inequality holds according to property 1 in the statement of the theorem).
According to the definition of a realization of a transition, the equality $\langle O_1 \rangle^{\xi_1} = 1$ holds. This equality, (10) and (11), imply that there is $i \in \Im$ such that

$$\Big((O_1, O_{T_i}) \cdot b_{s_1' s_2^i}\Big)^{(\xi_1, \xi_2)} = 1 \tag{12}$$

It is easy to prove that the equality

$$\Big((O_1, O_{T_i}) \cdot b_{s_1' s_2^i}\Big)^{(\xi_1, \xi_2)} = b_{s_1' s_2^i}{}^{(\xi_1 \cdot O_1, \xi_2 \cdot O_{T_i})} \tag{13}$$

holds. This equality is an analogue of the equality in the end of section 3.3, and is proved by induction on the total number of AOs in $[O_1]$ and $[O_2]$.
(12) and (13) imply that

$$b_{s_1' s_2^i}{}^{(\xi_1 \cdot O_1, \xi_2 \cdot O_{T_i})} = 1 \tag{14}$$

By the definition of μ and ξ_2, the statement (9) in this case ($v_1 = P_1^0$) follows from the statement

$$\exists\, \xi_2' : \xi_2 \stackrel{\tau^*}{\to} \xi_2',\ b^{(\xi_1', \xi_2')}_{at^{\xi_1'}_{P_1} at^{\xi_2'}_{P_2}} = 1 \tag{15}$$

Define $\xi_2' \stackrel{\text{def}}{=} (\xi_2 \cdot (at_{P_2} := s_2^i)) \cdot O_{T_i}$. Since $at^{\xi_1'}_{P_1} = s_1'$, and $\xi_1' = (\xi_1 \cdot (at_{P_1} := s_1')) \cdot O_1$, then (15) follows from the statements

$$\xi_2 \stackrel{\tau^*}{\to} (\xi_2 \cdot (at_{P_2} := s_2^i)) \cdot O_{T_i} \tag{16}$$

$$b^{((\xi_1 \cdot (at_{P_1} := s_1')) \cdot O_1, (\xi_2 \cdot (at_{P_2} := s_2^i)) \cdot O_{T_i})}_{s_1' s_2^i} = 1 \tag{17}$$

(16) follows from the definitions of concepts of a CT and a concatenation of operators and from the statements $at^{\xi_2}_{P_2} = s^0_{P_2}$ and $\langle O_{T_i} \rangle^{\xi_2} = 1$. The first

of these statements follows from the equality $\langle P_2\rangle^{\xi_2} = 1$, and the second is justified as follows. The definition of formulas of the form $(O_1, O_2)\cdot b$ implies that the statement (12) can be rewritten as

$$\Big(\langle O_1\rangle \wedge \langle O_{T_i}\rangle \wedge b\Big)^{(\xi_1,\xi_2)} = 1 \tag{18}$$

where b is some formula. Since $X_{P_1} \cap X_{P_2} = \emptyset$, then (18) implies the desired statement $\langle O_{T_i}\rangle^{\xi_2} = 1$.
(17) follows from (14) and from the assumption that at_{P_1} and at_{P_2} do not occur in $b_{s'_1 s^i_2}$, O_1 and O_{T_i}.
Thus, in the case $v_1 = P_1^0$ the property (9) holds.
In the case $v_1 \neq P_1^0$ the property (9) can be proved similarly.

3. Let $(v_1, v_2) \in \mu$ and $v_1 \stackrel{a}{\to} v'_1$, where $a \neq \tau$. We must prove that

$$\exists\, v'_2 : v_2 \stackrel{\tau^* a \tau^*}{\longrightarrow} v'_2,\ (v'_1, v'_2) \in \mu \tag{19}$$

(a) At first consider the case $v_1 = P_1^0$ and $a = \alpha?d$.
If $v_1 = P_1^0$, then $v_2 = P_2^0$, and according to the definition of the graph P_1^r (section 7.3), $\exists \xi_1 \in X^{\bullet}_{P_1} : \langle P_1\rangle^{\xi_1} = 1$ and the graph P_1^r has the edge $\xi_1 \stackrel{a}{\to} \xi'_1 = v'_1$, i.e. (ξ_1, a, ξ'_1) is a realization of a transition t of the form $s^0_{P_1} \stackrel{O_1}{\to} s'_1$ from T_{P_1}, where O_1 is an input operator. Using the notation introduced at the end of section 7.2, we can write $\xi'_1 = \xi_1 \cdot t^d$.
Just as in the preceding item, we prove that $\exists\, \xi_2 \in X^{\bullet}_{P_2}$: $\langle P_2\rangle^{\xi_2} = 1$, and there exists a CT $s^0_{P_2} \stackrel{T_i}{\to} s^i_2$ of the process P_2, such that the equality

$$\Big((O_1, O_{T_i})\cdot b_{s'_1 s^i_2}\Big)^{(\xi_1,\xi_2)} = 1 \tag{20}$$

holds, which should be understood in the following sense: for each of valuation $\xi \in (X_{P_1} \cup X_{P_2} \cup \{z\})^{\bullet}$ (where z is a variable, referred in the item 3d of the definition from section 8.2, we can assume that $z \notin ((X_{P_1} \cup X_{P_2}))$, coinciding with ξ_i on X_{P_i} $(i = 1, 2)$, the equality $\Big((O_1, O_{T_i})\cdot b_{s'_1 s^i_2}\Big)^{\xi} = 1$ holds. In particular, (20) implies that O_{T_i} is an input operator, and $N_{O_{T_i}} = N_{O_1} = \alpha$.
Define $\xi'_2 \stackrel{\text{def}}{=} \xi_2 \cdot T_i^d$. It is easy to prove that $\xi_2 \stackrel{\tau^* a \tau^*}{\to} \xi'_2$, and the statement (19) in the case $v_1 = P_1^0$ follows from the equality

$$b^{(\xi_1\cdot t^d, \xi_2 \cdot T_i^d)}_{s'_1 s^i_2} = 1 \tag{21}$$

which is justified as follows.
In this case O_1 and O_{T_i} can be represented as concatenation of the form

$$O_1 = (O'_1 \cdot [\alpha?x]) \cdot O''_1, \quad O_{T_i} = (O'_{T_i} \cdot [\alpha?y]) \cdot O''_{T_i}$$

Definition of formulas of the form (8) implies that

$$\begin{aligned}&(O_1, O_{T_i}) \cdot b_{s'_1 s^i_2} = \\ &= \Big((O'_1 \cdot [\alpha?x]) \cdot O''_1, (O'_{T_i} \cdot [\alpha?y]) \cdot O''_{T_i}\Big) \cdot b_{s'_1 s^i_2} = \\ &= \Big(O'_1 \cdot [\alpha?x], O'_{T_i} \cdot [\alpha?y]\Big) \cdot \Big((O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2}\Big) = \\ &= (O'_1, O'_{T_i}) \cdot \Big(((O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2})(z/x, z/y)\Big)\end{aligned} \tag{22}$$

(20) and (22) imply the equality

$$\Big(((O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2})(z/x, z/y)\Big)^{(\xi_1 \cdot O'_1, \xi_2 \cdot O'_{T_i})} = 1$$

Its special case is the equality

$$\Big(((O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2})(d/x, d/y)\Big)^{(\xi_1 \cdot O'_1, \xi_2 \cdot O'_{T_i})} = 1$$

The last equality can be rewritten as

$$\Big((O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2}\Big)^{(\xi_1 \cdot O'_1 \cdot (x:=d), \xi_2 \cdot O'_{T_i} \cdot (y:=d))} = 1$$

whence it follows that

$$\Big(b_{s'_1 s^i_2}\Big)^{(\xi_1 \cdot O'_1 \cdot (x:=d) \cdot O''_1, \xi_2 \cdot O'_{T_i} \cdot (y:=d) \cdot O''_{T_i})} = 1 \tag{23}$$

It is easy to see that the left side of (23) coincides with the left side of the equality (21).
Thus, in the case $v_1 = P^0_1$ and $a = \alpha?d$ the property (19) is proven.
In the case $v_1 \neq P^0_1$ and $a = \alpha?d$ the property (19) can be proved similarly.

(b) Now we prove (19), when $a = \alpha!d$. As in the previous item, we consider only the case $v_1 = P^0_1$.
If $v_1 = P^0_1$, then $v_2 = P^0_2$, and
- $\exists \xi_1 \in X^{\bullet}_{P_1} : \langle P_1 \rangle^{\xi_1} = 1$ and the graph P^r_1 has the edge $\xi_1 \stackrel{a}{\to} \xi'_1 = v'_1$, i.e. (ξ_1, a, ξ'_1) is a realization of a transition $t \in T_{P_1}$ of the form $s^0_{P_1} \stackrel{O_1}{\to} s'_1$, where O_1 is an output operator
- $\exists \xi_2 \in X^{\bullet}_{P_2}$: $\langle P_2 \rangle^{\xi_2} = 1$, and there exists a CT $s^0_{P_2} \stackrel{T_i}{\to} s^i_2$ of the process P_2, such that

$$\Big((O_1, O_{T_i}) \cdot b_{s'_1 s^i_2}\Big)^{(\xi_1, \xi_2)} = 1 \tag{24}$$

(24) implies that O_{T_i} is an output operator, and $N_{O_{T_i}} = N_{O_1} = \alpha$.
Define $\xi'_2 \stackrel{\text{def}}{=} \xi_2 \cdot T_i$. For a proof of (19) it is enough to prove the statements

$$\xi_2 \stackrel{\tau^* a \tau^*}{\to} \xi'_2 \tag{25}$$

$$b_{s'_1 s^i_2}^{(\xi_1 \cdot t, \xi_2 \cdot T_i)} = 1 \tag{26}$$

In this case O_1 and O_{T_i} can be represented as concatenations of the form

$$O_1 = (O'_1 \cdot [\alpha!e_1]) \cdot O''_1 \tag{27}$$

$$O_{T_i} = (O'_{T_i} \cdot [\alpha!e_2]) \cdot O''_{T_i} \tag{28}$$

The definition of formulas of the form (8) implies that

$$\begin{array}{l} (O_1, O_{T_i}) \cdot b_{s'_1 s^i_2} = \\ = \left((O'_1 \cdot [\alpha!e_1]) \cdot O''_1, (O'_{T_i} \cdot [\alpha!e_2]) \cdot O''_{T_i}\right) \cdot b_{s'_1 s^i_2} = \\ = \left(O'_1 \cdot [\alpha!e_1], O'_{T_i} \cdot [\alpha!e_2]\right) \cdot \left((O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2}\right) = \\ = (O'_1, O'_{T_i}) \cdot \left\{ \begin{array}{l} e_1 = e_2 \\ (O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2} \end{array} \right\} \end{array} \tag{29}$$

(24) and (29) imply the equality

$$\left\{ \begin{array}{l} e_1 = e_2 \\ (O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2} \end{array} \right\}^{(\xi_1 \cdot O'_1, \xi_2 \cdot O'_{T_i})} = 1$$

from which it follows that

$$e_1^{\xi_1 \cdot O'_1} = e_2^{\xi_2 \cdot O'_{T_i}} \tag{30}$$

$$\left((O''_1, O''_{T_i}) \cdot b_{s'_1 s^i_2}\right)^{(\xi_1 \cdot O'_1, \xi_2 \cdot O'_{T_i})} = 1 \tag{31}$$

By assumption, $(\xi_1, \alpha!d, \xi'_1)$ is a realization of the transition $s^0_{P_1} \stackrel{O_1}{\to} s'_1$. From the representation of O_1 as a concatenation (27) it follows that $d = e_1^{\xi_1 \cdot O'_1}$, whence, according to (30) we get the equality $d = e_2^{\xi_2 \cdot O'_{T_i}}$. From this and from a representation of O_{T_i} as a concatenation (28) it follows that $(\xi_2, \alpha!d, \xi_2 \cdot T_i)$ is a realization of the CT T_i. Since $\xi_2 \cdot T_i = \xi'_2$ and $\alpha!d = a$, then it follows that we are justified the statement (25).
The statement (26) follows from (31).
Thus, in the case $v_1 = P^0_1$ and $a = \alpha!d$ the property (19) is proven.
In the case $v_1 \neq P^0_1$ and $a = \alpha!d$ the property (19) can be proved similarly

The symmetrical conditions on the relation μ (i.e., second parts of the conditions on μ, presented in second and third items in section 8.1) can be proved similarly. ■

9 Simplification of processes

9.1 A concept of a simplification of processes

The concept of a simplification of processes is intended to reduce the problem of verification of processes.

A **simplification** of a process P is a sequence of transformations of this process, each of which is performed according to one of the rules set out below. Each of these transformations (except the first) is performed on the result of previous transformation. A **result** of a simplification is a result of last of these transformations.

Simplification rules are defined as follows. Let P be a process.

Rule 1 (removing of states).
If $s \in S_P \setminus \{s_P^0\}$, and

- $s_1 \stackrel{O_1}{\to} s$, ..., $s_n \stackrel{O_n}{\to} s$ are all transitions incoming to s
- $s \stackrel{O'_1}{\to} s'_1$, ..., $s \stackrel{O'_m}{\to} s'_m$ are all transitions outgoing from s, and if all these transitions are internal, then $\langle O'_i \rangle \wedge \langle O'_j \rangle = \bot$ if $i \neq j$
- $s \notin \{s_1, \ldots, s_n, s'_1, \ldots, s'_m\}$
- $\forall\, i = 1, \ldots, n,\ \forall\, j = 1, \ldots, m \quad \exists\, O_i \cdot O'_j$

then s and all transitions related to s are removed from P, and the transitions $s_i \stackrel{O_i \cdot O'_j}{\longrightarrow} s'_j$ (where $i = 1, \ldots, n,\ j = 1, \ldots, m$) are added to P.

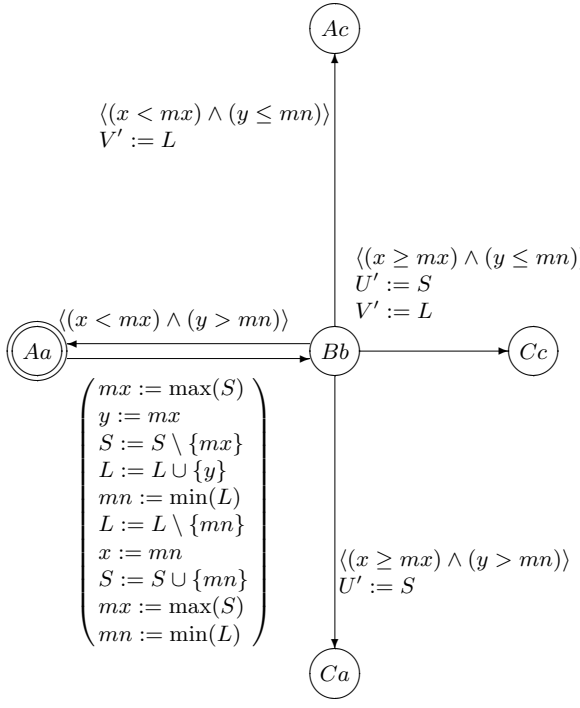
Rule 2 (fusion).
If P has a pair of transitions of the form $s_1 \stackrel{O}{\to} s_2$, $s_1 \stackrel{O'}{\to} s_2$, and $[O] = [O']$, then this pair is replaced by a single transition $s_1 \stackrel{b[O]}{\longrightarrow} s_2$, where $b = \langle O \rangle \vee \langle O' \rangle$.

Rule 3 (elimination of unessential assignments).
If P has an AO $(x := e)$, where $x \notin X_P^s$, then this AO is removed from P.

Theorem 2
If P' is a result of simplification of P, then $P' \approx P$.

9.2 An example of a process obtained by a simplification

In this section we present a process which is obtained by a simplification of the process (2). This process has the following form:



(32)

This simplified process allows to detect some simple flaws of the algorithm of separation of sets, for examle a possibility of a deadlock situation: there are states of the process (32) (namely, Ac and Ca) such that

- there is no transitions starting at these states
- but falling into these states is not a normal completion of the process.

9.3 Another example of a simplification of a process

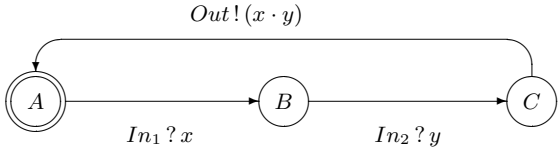
Suppose we have a system "multiplier", which has

- two input ports with the names In_1 and In_2, and
- one output port with the name Out.

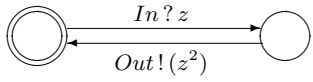
An execution of the multiplier is that it

- receives on its input ports two values, and
- gives their product on the output port.

A behavior of the multiplier is described by the process Mul:



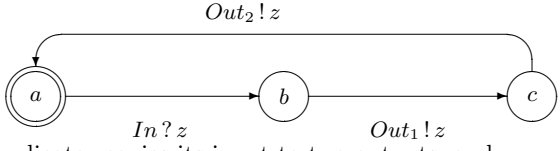
Using this multiplier, we want to build a system "calculator of a square", whose behavior is described by the process $Square\_Spec$:



The desired system is a composition of

– the auxiliary system "duplicator" having
  - an input port In, and
  - output ports Out_1 and Out_2

  behavior of which is described by the process Dup:

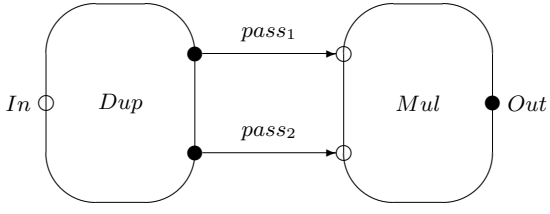


  i.e. the duplicator copies its input to two outputs, and
– the multiplier, which receives on its input ports those values that duplicator gives.

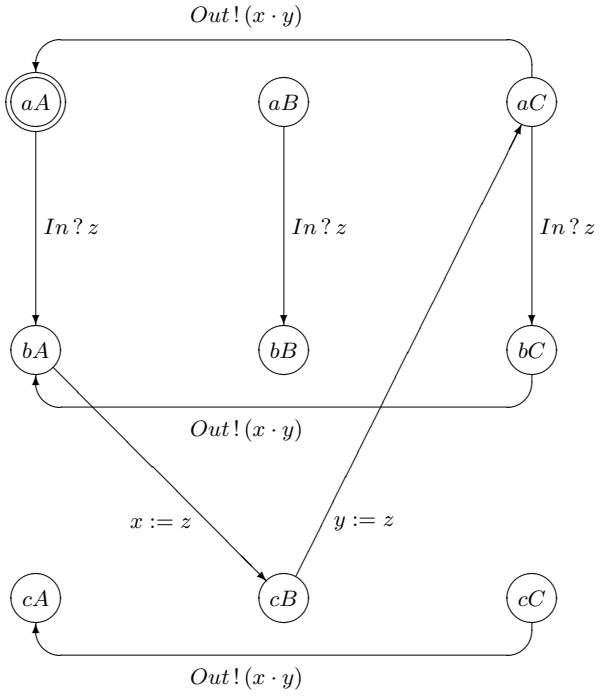
A process $Square$, corresponding to such a composition is defined as follows:

$$Square \stackrel{\text{def}}{=} \stackrel{\text{def}}{=} \left(\begin{array}{l} Dup[pass_1/Out_1, pass_2/Out_2] \\ \mid Mul[pass_1/In_1, pass_2/In_2] \end{array} \right) \setminus \{pass_1, pass_2\}$$

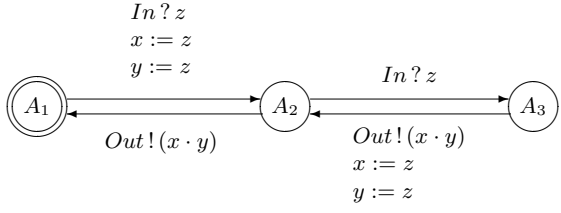
A flow graph of the process $Square$ has the form



However, the process *Square* does not meet the specification *Square_Spec* (i.e. *Square* and *Square_Spec* are not observationally equivalent). This fact is easy to detect by a construction of a graph representation of *Square*, which, by definition of operations of parallel composition, restriction and renaming, is the following:



After a simplification of this process we obtain the process



(33)

which shows that

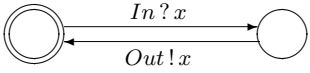
– the process *Square* can execute two input actions together (i.e. without an execution of an output action between them), and

– the process *Square_Spec* can not do that.

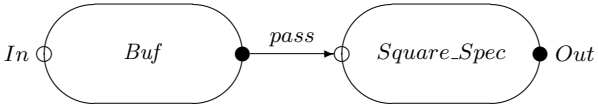
The process *Square* meets another specification:

$$Square\_Spec' \stackrel{\text{def}}{=} \left(\begin{array}{l} Buf[pass/Out] \ \mid \\ \mid \ Square\_Spec[pass/In] \end{array} \right) \setminus \{pass\}$$

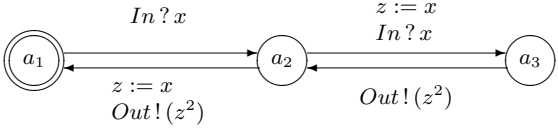
where *Buf* is a buffer which can store one message, whose behavior is represented by the diagram



A flow graph of *Square_Spec′* has the form



A simplified process *Square_Spec′* has the form

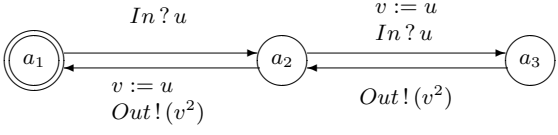


(34)

The statement that *Square* meets the specification *Square_Spec′* can be formalized as

$$(33) \approx (34) \tag{35}$$

We justify (35) with use of theorem 1. At first, we rename variables of the process (34), i.e. instead of (34) we shall consider the process



(36)

To prove (33) $\approx$ (36) with use of theorem 1 we define the formulas b_{A_i,a_j} (where $i, j = 1, 2, 3$) as follows:

- $b_{A_i,a_j} \stackrel{\text{def}}{=} \bot$, if $i \neq j$
- $b_{A_1,a_1} \stackrel{\text{def}}{=} \top$
- $b_{A_2,a_2} \stackrel{\text{def}}{=} (x = y = z = u)$
- $b_{A_3,a_3} \stackrel{\text{def}}{=} (x = y = v) \wedge (z = u)$.

10 An example: verification of a sliding window protocol

In this section we present an example of use of theorem 1 for a verification of a sliding window protocol.

A sliding window protocol ensures a transmission of messages from one agent to another through a medium, in which messages may get distorted or lost. In this section we consider a two-way sliding window protocol, in which the agents can both send and receive messages from each other. We do not present here a detail explanation of this protocol, a reader can find it in section 3.4.2 of the book [10] (a protocol using go back n).

10.1 A structure of the protocol

The protocol is a system consisting of interacting components, including

- components that perform a formation, sending, receiving and processing of messages (such components are called **agents**, and messages sent from one agent to another, are called **frames**), and
- a medium, through which frames are forwarded (such a medium is called a **channel**).

A detailed description of the components and relation between them is represented in the Appendix.

10.2 Frames

Each frame f, which is sent by any of the agents, contains a packet x, and a couple of numbers:

- a number $s \in \mathbf{Z}_n \stackrel{\text{def}}{=} \{0, 1, \ldots, n-1\}$ (where n is a fixed integer), which is associated with the packet x and with the frame f, and
- a number $r \in \mathbf{Z}_n$, which is a number associated with a last received undistorted frame.

To build a frame, a function φ is used, i.e. a frame has the form $\varphi(x, s, r)$.

To extract the components x, s, r from the frame $\varphi(x, s, r)$, the functions $info$, seq and ack are used, these functions have the following properties:

$$info(\varphi(x, s, r)) = x, \quad seq(\varphi(x, s, r)) = s, \quad ack(\varphi(x, s, r)) = r$$

10.3 Window

The set of variables of an agent contains an array $x[n]$. Values of some components of this array are packets which are sent, but not yet acknowledged. A set of components of the array x, which contain such packets at a current time, is called a **window**.

Three variables of the agent are related to the window: b (a lower bound of the window), s (an upper bound of the window), and w (a number of packets in the window). Values of these variables belong to the set $\mathbf{Z}_n$. At the initial moment values of b, s and w are equal to 0. At any moment the window can be empty (if $b = s$), or not empty (if $b \neq s$). In the last case the window consists of elements of x with indices from the set $[b, s[$, where $[b, s[$ denotes the set

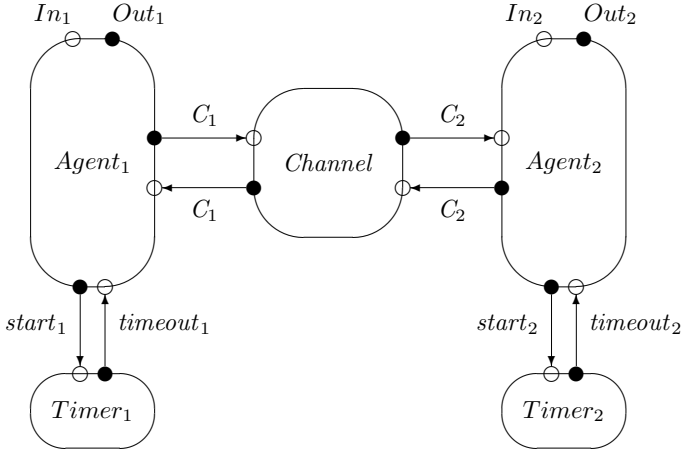
- $\{b, b+1, \ldots, s-1\}$, if $b < s$, and
- $\{b, b+1, \ldots, n\} \cup \{0, 1, \ldots, s-1\}$, if $s < b$.

Adding a new packet to the window is performed by an execution of the following actions: this packet is written in the component $x[s]$, s is increased by 1 modulo n (i.e. a new value of s is assumed to be $s+1$, if $s < n-1$, and 0, if $s = n-1$), and w is increased by 1. Removing a packet from the window is performed by an execution of the following operations: b is increased by 1 modulo n, and w is decreased by 1 (i.e. it is removed a packet whose number is equal to the lower bound of the window).

If an agent received a frame, the third component r of which (i.e. a number of an acknowledgment) is such that $r \in [b, s[$, then all packets in the window with numbers from $[b, r[$ are considered as acknowledged and are removed from the window (even if their acknowledgments were not received).

10.4 Flow graph

A relation between subprocesses of sliding window protocol is represented by the flow graph:



10.5 Timers

Each component $x[i]$ of the array x is associated with a timer, which determines a duration of waiting of an acknowledgement from another agent of a receiving of the packet contained in the component $x[i]$. The combination of these timers is considered as a process $Timer$, which has an array $t[n]$ of boolean variables. The process $Timer$ has one state and transitions which are labeled by the following operators:

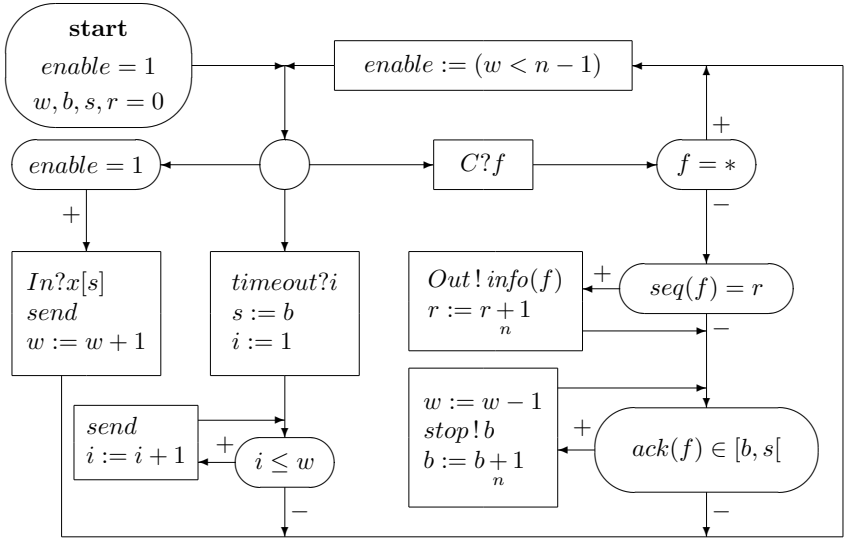
- $[start?i, t\,[i] := 1]$
- $[stop?i,\ t\,[i] := 0]$
- $(t\,[j] = 1)[timeout\,!\,j, t\,[j] := 0]$ (where $j = 0, \ldots, n-1$)

An initial condition is $t = (0, \ldots, 0)$.

If an agent has received an object with a name *timeout* from a timer, then the agent sends again all packets from its window.

10.6 Agents

A behavior of each agent is described by the same process, combining functions of a sender and a receiver. This behavior can be represented by the following flowchart.



where

- *send* is an abbreviation of the list of AOs $\left\{ \begin{array}{l} C\,!\,\varphi(x[s], s, r \underset{n}{-} 1) \\ start\,!\,s \\ s := s \underset{n}{+} 1 \end{array} \right\}$

– $*$ is a special notation for a distorted message, and
– a value of the variable *enable* is 1, if the agent can receive a new packet from his network level (i.e. $w < n-1$), and 0, otherwise.

Processes $Agent_1$ and $Agent_2$ are obtained by a simple transformation of this flowchart, and by an addition of corresponded index (1 or 2) to its variables and names.

10.7 Specification

External actions of the above protocol (i.e. actions which are related to its communication with a network level) have the form $In_1?d$, $In_2?d$, $Out_1!d$ and $Out_2!d$. Assume that we take into account only external actions $In_1?d$ and $Out_2!d$, and ignore other its external actions (i.e. we consider a transmission only in one direction: from the left to the right). We would like to prove that such behavior is equivalent to a behavior of a process B_{n-1}, which is called "a FIFO buffer which can hold at most $n-1$ frames", and is defined as follows:

– variables of B_{n-1} are
  - an array $(x[0], \ldots, x[n-1])$, elements of which have the same type as a type of frames in the above protocol, and
  - variables r, s, u, values of which belong to $\mathbf{Z}_n$, and have the following meaning: at every moment
    * a value of u is equal to a number of frames in the buffer
    * values r and s can be interpreted as lower and upper bounds of a part of the array x, which stores the received frames, which has not yet been issued from the buffer
– B_{n-1} has one state and 2 transitions with labels

$$(u < n-1)\,[In?x[s],\ s := s \underset{n}{+} 1,\ u := u+1]$$
$$(u > 0)\,[Out\,!\,x[r],\ r := r \underset{n}{+} 1,\ u := u-1]$$

where $\forall\, i \in \{0, n-2\}\ \ i \underset{n}{+} 1 \stackrel{\text{def}}{=} i{+}1$ and $(n-1) \underset{n}{+} 1 \stackrel{\text{def}}{=} 0$
– initial condition is $r = s = u = 0$.

10.8 A process corresponded to the protocol

A process that describes a behavior of the protocol with respect to the above specific point of view (where we ignore actions of the form $In_2?d$ and $Out_1!d$) is constructed as a parallel composition of the processes corresponded to components of this procotol, with elimination of atomic operators related to ignored communications.

10.9 Verification

With use of the simplification operations from section 9, we can transform the process corresponded to the protocol (with elimination of atomic operators which are corresponded to ignored actions) to a process P with only one state and with transitions labelled by the following operators:

- $(w < n-1)\ [In?x[s],\ M_1 := M_1 \cdot \varphi(x[s], s, \ldots),\ s := s \underset{n}{+} 1,\ w := w+1]$
- $(M_1 \neq \varepsilon) \wedge (seq(\hat{M}_1) = r)\ [Out\,!\,info(\hat{M}_1),\ r := r \underset{n}{+} 1,\ M_1 := M_1']$
- $(M_2 \neq \varepsilon) \wedge (ack(\hat{M}_2) \in [b, s[)\ [b := ack(\hat{M}_2) \underset{n}{+} 1,\ w := s \underset{n}{-} b,\ M_2 := M_2']$, where $\forall i, j \in \{0, n-1\}\quad i \underset{n}{-} j \stackrel{\text{def}}{=} i-j$, if $i-j \in \{0, n-1\}$, and $n + i-j$, otherwise
- $[M_1 := M_1 \cdot \varphi(x[b], b, \ldots), \ldots\ , M_1 := M_1 \cdot \varphi(x[s \underset{n}{-} 1], s \underset{n}{-} 1, \ldots)]$
- $(M_1 \neq \varepsilon)\ [M_1 := M_1']$
- $(M_2 \neq \varepsilon)\ [M_2 := M_2']$
- $[M_2 := M_2 \cdot \varphi(\ldots, \ldots, r \underset{n}{-} 1)]$

where

dots denote unessential components of expressions, and the symbols M_i, $\hat{M}_i$, M_i', $\cdot$ and ε have the following sense:

- M_1 and M_2 are variables of the process *Channel*, and values of these variables are lists of frames which were received by the process *Channel* (M_i holds frames received from $Agent_i$), every received frame is added to the end of a corresponded list
- $\hat{M}_i$ $(i = 1, 2)$ is an expression, a value of which is equal to the first element of the list M_i
- M_i' $(i = 1, 2)$ is an expression, a value of which is equal to the list M_i without its first element
- $\cdot$ is a function of an addition of a frame to the end of a list
- ε is a constant, a value of which is an empty list.

For a proof that the process P is observationally equivalent to the process B_{n-1}, we define a formula $b_{s_1 s_2}$ where s_1 is a unique state of P and s_2 is a unique state of B_{n-1} as a conjunction of the following formulas:

- $(M_1 \neq \varepsilon) \wedge (seq(\hat{M}) = r) \Rightarrow u > 0$
- $\forall\, f \in M_1\ info(f) = x[seq(f)]$
- $\forall\, f \in M_2\quad ack(f) \in [b \underset{n}{-} 1, r[$
- $[r, s[\ \subseteq\ [b, s[$
- $w = s \underset{n}{-} b \leq n-1$
- $u = s \underset{n}{-} r \leq w$
- if a value of M_2 is $f_1 \cdot \ldots \cdot f_k$, then the sequence $ack(f_1)\quad \ldots\quad ack(f_k)$ is monotonically increasing (mod n) subsequence of $[b \underset{n}{-} 1, r[$

(the last record is not a formula, but can be represented by a formula, we omit this representation).

It is not so diffcult to check that $b_{s_1 s_2}$ satisfies the conditions of theorem 1 and this proves that the process P is observationally equivalent to B_{n-1}.

11 Conclusion

The concept of a process with message passing which is presented in this paper can be considered as a formal model of a communicating program without recursion. In the paper we have established suffcient conditions of observational equivalence of processes. The next steps of investigations in this area can be the following.

- Find necessary and suffcient conditions of observational equivalence of processes with message passing.
- Generalize the proposed concept of a process with message passing for formal modeling of communicating programs with recursion, and find necessary and suffcient conditions of observational equivalence of such processes.

References

1. R. Milner: A Calculus of Communicating Systems. Number 92 in Lecture Notes in Computer Science. Springer Verlag (1980)
2. R. Milner: Communicating and Mobile Systems: the π-calculus. Cambridge University Press (1999)
3. C.A.R. Hoare: Communicating Sequential Processes. Prentice Hall (1985)
4. Clarke, E.M., Grumberg, O., and Peled, D.: Model Checking, MIT Press (1999)
5. C.A. Petri: Introduction to general net theory. In W. Brauer, editor, Proc. Advanced Course on General Net Theory, Processes and Systems, number 84 in LNCS, Springer Verlag (1980)
6. J.A. Bergstra, A. Ponse, and S.A. Smolka, editors: Handbook of Process Algebra. North-Holland, Amsterdam (2001)
7. D. Brand, P. Zafiropulo: On Communicating Finite-State Machines. Journal of the ACM, Volume 30 Issue 2, April 1983, pp. 323-342. ACM New York, NY, USA (1983)
8. R.W. Floyd: Assigning meanings to programs. In J.T. Schwartz, editor, Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science, pages 19-32. AMS (1967)
9. Badban, B. and Fokkink, W.J. and van de Pol, J.C.: Mechanical Verification of a Two-Way Sliding Window Protocol (Full version including proofs). Internal Report TR-CTIT-08-45, Centre for Telematics and Information Technology, University of Twente, Enschede, June 2008. `http://doc.utwente.nl/64845/` (2008)
10. A. Tanenbaum: Computer Networks. Fourth Edition. Prentice Hall (2002)
11. B. Hailpern: Verifying Concurrent Processes Using Temporal Logic. LNCS 129. Springer-Verlag (1982)
12. G. Holzmann: Design and Validation of Computer Protocols. Prentice Hall (1991)
13. G. Holzmann: The model checker Spin. IEEE Transactions on Software Engineering, 23:279-295 (1997)

14. R. Kaivola: Using compositional preorders in the verification of sliding window protocol. In Proc. 9th Conference on Computer Aided Verification, LNCS 1254, pages 48-59 (1997)
15. P. Godefroid and D. Long: Symbolic protocol verification with Queue BDDs. Formal Methods and System Design, 14(3):257-271 (1999)
16. K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen: Divide, abstract, and model-check. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, Proc. 6th SPIN Workshop on Practical Aspects of Model Checking, Lecture Notes in Computer Science 1680, pages 57-76. Springer-Verlag (1999)
17. T. Latvala: Model checking LTL properties of high-level Petri nets with fairness constraints. In J. Colom and M. Koutny, editors, Proc. 21st Conference on Application and Theory of Petri Nets, Lecture Notes in Computer Science 2075, pages 242-262. Springer-Verlag (2001)
18. D. Chkliaev, J. Hooman, and E. de Vink: Verification and improvement of the sliding window protocol. In H. Garavel and J. Hatcliff, editors, Proc. 9th Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 2619, pages 113-127 (2003)
19. A. Schoone: Assertional Verification in Distributed Computing. PhD thesis, Utrecht University (1991)
20. F. Vaandrager: Verification of two communication protocols by means of process algebra. Technical Report Report CS-R8608, CWI (1986)
21. R. Groenveld: Verification of a sliding window protocol by means of process algebra. Technical Report P8701, University of Amsterdam (1987)
22. J. van Wamel: A study of a one bit sliding window protocol in ACP. Technical Report P9212, University of Amsterdam (1992)
23. M. Bezem and J. Groote: A correctness proof of a one bit sliding window protocol in μCRL. The Computer Journal, 37(4):289-307 (1994)

Turchin's Relation and Subsequence Relation on Traces Generated by Prefix Grammars*

Antonina N. Nepeivoda

Program System Institute of RAS
Pereslavl–Zalessky

Abstract. Turchin's relation was introduced in 1988 by V. F. Turchin for loop approximation in supercompilation. The paper studies properties of an analogue of the Turchin relation and properties of the subsequence embedding on a restricted set of traces generated by prefix grammars or by a product of prefix grammars.

Keywords: Higman's lemma, prefix rewriting, well binary relation, computational complexity, termination, supercompilation

1 Introduction

In computer science the homeomorphic embedding is investigated from two completely different points of view, for it is of both theoretical and practical interest.

On the one hand, the embedding showed itself to be useful as a branch termination criterion in constructing tools for program transformation ([14], [3]). What makes the homeomorphic embedding reasonable as a termination criterion is the non-existence of an infinite sequence of finite labeled trees such that no tree in the sequence is embedded into some its derivative (the fact was proved by Kruskal and is called Kruskal's theorem; for an elegant proof of the fact see [8]). Relations with this property are called well-binary relations.

Definition 1. *R,$R \subset S \times S$, is called **a well binary relation**, if every sequence $\{\Phi_n\}$ of elements from S such that $\forall i, j(i < j \Rightarrow (\Phi_i, \Phi_j) \notin R)$ is finite. If R is well binary and transitive it is called a well quasiorder (wqo).*

*A sequence $\{\Phi_n\}$ with the property $\forall i, j(i < j \Rightarrow (\Phi_i, \Phi_j) \notin R)$ is called **a bad sequence** with respect to R. Thus, well-binariness of R can be formulated equivalently as "all bad sequences with respect to R are finite".*

On the other hand, well-binariness of the homeomorphic embedding is shown to be non-provable in the Peano arithmetic with the first-order induction scheme [13], and this fact aroused interest of logicians and computer scientists with background in mathematical logic (a thorough study of the proof-theoretical strength of the fact is in [17]). Studies of the homeomorphic embedding as a termination

* The reported study was partially supported by Russian Foundation for Basic Research project No. 14-07-00133-a.

criterion for term rewriting systems ([12,15]) are located in the middle between these poles of pure theory and practice.

The problem is that since these two domains live their own separate lives, it is not always obvious how to use the theoretical investigations in the practical program transformations. Theoricists study properties of the homeomorphic embedding (and similar relations) on arbitrary (maybe not even computable) sequences of trees, and that can imply somewhat obscure view on practical features of the relations: in particular it was established that the upper bound on a bad sequence length with respect to the homeomorphic embedding dominates every multiple recursive function [13], which looks redundantly from the practical point of view. But in real applications the opposite problem becomes much more frequent: the homeomorphic embedding yields branch termination too early [7,12]. In some algorithms of program analysis this flaw was partially fixed either by making an additional annotation [6] or by intersecting the embedding with other wqos [1].

In this paper we study properties of a special case of the homeomorphic embedding on a restricted set of computable sequences.

Definition 2. *Having two words Φ, Ψ in an alphabet Υ let us say that Φ is embedded in Ψ* ***with respect to the subsequence relation*** *($\Phi \trianglelefteq \Psi$) ($\trianglelefteq$ is also called the scattered subword relation) if Φ is a subsequence of Ψ.*

The subsequence relation is proved to be a well quasiorder by G. Higman [4]. We prove that while applied only to sequences generated by prefix grammars the relation admits bad sequences not more than exponential over a grammar size. If we apply the relation to a direct product of sequences generated by prefix grammars we receive the multiple recursive upper bound found by H. Touzet [15]. Also we show how to make a refinement of the subsequence relation that solves the empty word problem for languages generated by alphabetic prefix grammars and inherits some useful features of Turchin's relation, which was also used in program transformation (in particular, in the supercompiler SCP4 [9]).

The paper is organized as follows. First, we introduce notion of a prefix grammar. Then we give a definition of the Turchin relation and shortly prove its well-binariness. After that we show how to build maximal bad sequences with respect to the Turchin relation and give some discussion on using this relation combined with other well binary relations. Finally, we show how our refinement for the Turchin relation allows to refine the subsequence relation and, using our knowledge about the Turchin relation, we investigate properties of the subsequence relation on traces generated by prefix grammars.

The main contributions of the paper are the following:

1. We outline the concept of Turchin's relation in terms of prefix grammars and investigate properties of the relation.
2. We link Turchin's relation with the subsequence relation and show how to model the former by the latter not using a notion of time for sequences generated by prefix grammars.

3. We determine upper bounds of bad sequence length with respect to both relations for sequences generated by a single prefix grammar and for direct products of two sequences generated by prefix grammars.
4. We show that a minimal natural well binary generalization of Turchin's relation on direct products of sequences generated by prefix grammar is the subsequence relation.

2 Prefix Grammars

We consider a restricted class of generative indeterministic grammars, in which rewriting rules are applied in an arbitrary order.

Definition 3. *A tuple* $\langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$*, where* Υ *is an alphabet,* $\Gamma_0 \in \Upsilon^+$ *is an initial word, and* $\mathbf{R} \subset \Upsilon^+ \times \Upsilon^*$ *is a finite set of rewrite rules*[1]*, is called* ***a prefix grammar*** *if* $R : R_l \to R_r$ *can be applied only to words of the form* $R_l\Phi$ *(where* R_l *is a prefix and* Φ *is a (possibly empty) suffix) and generates only words of the form* $R_r\Phi$*.*

If the left-hand side R_l *of a rule* $R : R_l \to R_r$ *has the length* 1 *(only the first letter is rewritten) then the prefix grammar is called* ***an alphabetic prefix grammar****.*

A trace *of a prefix grammar* $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$ *is a word sequence* $\{\Phi_i\}$ *(finite or infinite) where* $\Phi_1 = \Gamma_0$ *and for all* i $\exists R(R : R_l \to R_r \ \&\ R \in \mathbf{R} \ \&\ \Phi_i = R_l\Theta \ \&\ \Phi_{i+1} = R_r\Theta)$ *(*Θ *is a suffix). In other words, the elements of a trace are derived from their predecessors by applications of rewrite rules from* $\mathbf{G}$*.*

Example 1 *Consider the following prefix grammar* $\mathbf{G_\Lambda}$ *with* $\Upsilon = \{a, b, c\}$ *and the following rewrite rules:*

$$R^{[1]} : \Lambda \to ba \qquad R^{[2]} : b \to \Lambda \qquad R^{[3]} : aac \to \Lambda$$
$$R^{[4]} : aad \to \Lambda$$

We cannot apply the rule $R^{[3]}$ *to* **baacb***, for* **baacb** *starts not by* **aac***. If we apply* $R^{[1]}$ *or* $R^{[2]}$ *to* **baacb** *the only correct results of the applications are* **babaacb** *and* **aacb** *respectively.*

When V. F. Turchin discussed a search of semantic loops in Refal programs he considered a stack model, which resembles a prefix grammar [16]. V. F. Turchin proposed to observe call stack configurations to prevent infinite unfolding of a special sort. He aimed at cutting off branches where a stack top derives a path that ends with the same stack top. If we denote the stack top as Φ, the derivation of Φ with Φ on the top as $\Phi\Psi$, and the part of the initial stack that is not modified as Θ then we can say that a branch is dangerous with respect to Turchin's relation if it contains pairs of the form $\Phi\Theta, \Phi\Psi\Theta$. We can notice that the terms form a pair with respect to the subsequence relation, but V. F. Turchin proposed a stronger

[1] It is usually said that Υ is finite, but this restriction is unnecessary in our case. Only finiteness of $\mathbf{R}$ matters in our study.

relation for more precise identification of such stack configurations in his work [16]. V. F. Turchin used this relation to construct better loop approximations in residual programs, but the relation can be also used to forbid a program transformation process to halt driving on finite computation branches. The last property is analyzed in this paper for prefix-grammar-generated traces.

3 Turchin's Relation

To describe the Turchin relation for grammar-generated traces we use a formalization presented in [9]. The formalization introduces a notion of time indices. The main idea of the formalization is to mark every letter in the trace by a natural number that points to the position in the trace where the letter first appears. The order of words in a trace is from up to down.

The length of Φ is denoted by $|\Phi|$.

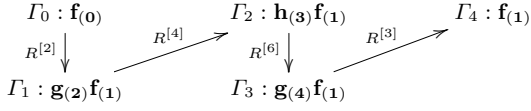
Definition 4. *Consider a trace $\{\Phi_i\}$ generated by a prefix grammar $\mathbf{G}$, $\mathbf{G} = \langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$. Supply letters of Φ_i by numbers that correspond to their **time indices** as follows. The i-th letter of Γ_0 is marked by the number $|\Gamma_0| - i$; if the maximal time index in the trace $\{\Phi_i\}_{i=1}^{k}$ is M and Φ_{k+1} is derived from Φ_k by an application of $R : R_l \to R_r$ then the i-th letter Φ_{k+1} ($i \leq |R_r|$) is marked by $M + |R_r| - i + 1$. Time indices of the other letters of Φ_{k+1} coincide with the corresponding time indices of Φ_k.*

We call such annotation time indexing and we call a trace with the annotation ***a computation***.

Example 2 *Let us consider a grammar $\mathbf{G_{LOG}}$ with $\Upsilon = \{f, g, h\}$ and the following rewrite rules:*

$$\begin{array}{lll} R^{[1]} : f \to \Lambda & R^{[3]} : g \to \Lambda & R^{[5]} : h \to \Lambda \\ R^{[2]} : f \to gf & R^{[4]} : g \to h & R^{[6]} : h \to g \end{array}$$

$\Gamma_0 = f$. A first segment of a computation yielded by the grammar $\mathbf{G_{LOG}}$ can look as:



The time indices are in subscripts, enclosed in brackets. Note that the letter $f_{(0)}$ in Γ_0 is replaced by $f_{(1)}$ in Γ_1, and $f_{(0)} \neq f_{(1)}$.

In the sequel Greek capitals (Γ, Δ, Θ, Ψ, Φ) denote words in a computation (with the time indexing). $\Delta[k]$ denotes the k-th letter of Δ (counting from the beginning).

An equivalence up to the time indices is formally defined as follows. $\Phi \approx \Psi$ if $|\Phi| = |\Psi|$ and $\forall i(i \geq 1 \;\&\; i \leq |\Phi| \Rightarrow (\Phi[i] = a_{(n)} \;\&\; \Psi[i] = b_{(m)} \Rightarrow a = b))$. The definition has the following simple meaning: if we erase time indices of all letters in Φ and Ψ then Φ and Ψ will coincide literally. For instance, in Example 2 $f_{(0)} \neq f_{(1)}$, but $f_{(0)} \approx f_{(1)}$.

Now we are ready to define ***Turchin's relation*** $\Gamma \preceq \Delta$. Loosely speaking, it includes pairs $\langle \Gamma, \Delta \rangle$, where Γ can be presented as `[Top][Context]`, Δ can be presented as `[Top][Middle][Context]`, and the suffix `[Context]` is not modified in the computation segment thar starts from Γ and ends with Δ.

Definition 5. $\Gamma \preceq \Delta \Leftrightarrow \Gamma = \Phi\Theta_0 \;\&\; \Delta = \Phi'\Psi\Theta_0 \;\&\; \Phi' \approx \Phi$. *Pairs* Γ, Δ *such that* $\Gamma \preceq \Delta$ *are called* ***Turchin pairs***[2].

$\preceq$ is not transitive but it is reflexive and antisymmetric up to $\approx$ [11]. Well-binariness of the relation can be proved using the following observation. If a rule R has a non-empty right-hand side R_r, $\Phi \approx R_r$, $\Phi' \approx R_r$, $\Phi\Theta_0$ precedes $\Phi'\Theta_1$, and $\exists i(\Theta_1[i] = \Theta_0[1])$ then $\Phi\Theta_0 \preceq \Phi'\Theta_1$. So the maximal word length in a bad sequence with respect to $\preceq$ is bounded by

$$|\Gamma_0| + \sum(|R_r^{[i]}| - 1)$$

where Γ_0 is the initial word and $\sum(|R_r^{[i]}| - 1)$ runs over the set of different right-hand sides of all rules.

The upper bound is not exact due to the following two limitations. First, not every letter can be rewritten to the chosen right-hand side, i.e. the letter f cannot be rewritten to h in a one step. Second, some rules can accidentally share some letters in their right-hand sides. I.e. the letter g in the right-hand side of the rule $f \to gf$ and the letter g in the right-hand side of the rule $h \to g$ have different nature and the coincidence of the two letters is occasional. In the next section we show how to partly avoid this difficulty.

4 Annotated Prefix Grammars

If in the rules $h \to g$ and $f \to gf$ we write down the corresponding letters as e.g. $g^{[f]}$ and $g^{[h]}$ and say that $g^{[f]} \not\approx g^{[h]}$ then the prefix grammar will generate computations with less number of occasional Turchin's pairs.

Let us give more formal definition of this sort of prefix grammars.

Definition 6. *A prefix grammar* $\mathbf{G}=\langle \Upsilon, \mathbf{R}, \Gamma_0 \rangle$, $\mathbf{R} \subset \Upsilon^+ \times \Upsilon^*$ *is called* ***annotated***[3] *if*

[2] In [9] it is also specified that $|\Phi| > 0$. If a computation is yielded by a grammar only with non-empty left-hand sides of rules then this limitation is unnecessary. Otherwise the condition $|\Phi| > 0$ becomes essential to make the upper bound C'_{Max} constructed with a help of Lemma 1 exact.

[3] This grammar property 2 plays a role only in the construction of a longest bad sequences. So in most propositions grammars with only the properties 1 and 3 are also considered as annotated.

1. *For every two rules* $R : R_l \to R_r$, $R' : R'_l \to R'_r$, *if* $\exists i, j(R_r[i] \approx R'_r[j])$, *then* $R_r \approx R'_r$;
2. *If* $R_l \to R_r \in \mathbf{R}$ *and there is a rule* $R'_l \to R'_r$ *in* $\mathbf{R}$ *then* $R'_l \to R_r \in \mathbf{R}$.
3. *The initial word contains only unique letters:* $\forall i, j, k(R_r^{[k]}[i] \neq \Gamma_0[j])$.

Consider the following algorithm that transforms a prefix grammar $\mathbf{G}$ to an annotated $\mathbf{G}'$.

1. Let $a = R_r^{[n]}[i]$, $a \in \Upsilon$, n be an unique number of the rule with the right-hand side $R_r^{[n]}$. a corresponds to the pair $\langle a, 2^n * 3^{i-1}\rangle$. We set $n = 0$ for the initial word Γ_0 and denote the corresponding tuple of the pairs $\langle \Gamma_0[1], 1\rangle\langle \Gamma_0[2], 3\rangle \ldots \langle \Gamma_0[|\Gamma_0|], 3^{|\Gamma_0|}\rangle$ as Γ'_0.
2. A rewrite rule $R' : R'_l \to \Phi$ of the grammar $\mathbf{G}'$ corresponds the the equivalence class up to left-hand sides of rules $\langle a_i, n_i\rangle \to \Phi$, where Φ is a right-hand side of a rule from $\mathbf{G}$ after the first step, and $\langle a_i, n_i\rangle$ is arbitrary.

If the initial grammar $\mathbf{G}$ yields a bad sequence then the computation by $\mathbf{G}'$ that is derived from Γ'_0 by application of the rules from the equivalence classes that correspond to the right-hand sides of the rules that are applied in the computation by $\mathbf{G}$ is also a bad sequence.

We do not differ rewrite rules with the different left-hand sides in annotated grammars and write them as $x \to R_r$ where x denotes an arbitrary pair sequence of a bounded length.

Example 3 *Let us transform the prefix grammar* $\mathbf{G_{LOG}}$ *from Example 2 into an annotated.*

$\mathbf{G'_{LOG}}$*:*
$\Gamma_0 = \langle f, 1\rangle$ $\quad R^{[2]} : x \to \langle g, 4\rangle$
$R^{[1]} : x \to \langle g, 2\rangle\langle f, 6\rangle$ $\quad R^{[3]} : x \to \langle h, 8\rangle$
$R^{[4]} : x \to \Lambda$

The computation by $\mathbf{G_{LOG}}$ *that corresponds to the computation from Example 2 now begins as follows:*

$$\begin{array}{ccc} \Gamma_0 : \langle \mathbf{f}, \mathbf{1}\rangle_{(\mathbf{0})} & & \Gamma_2 : \langle \mathbf{h}, \mathbf{8}\rangle_{(\mathbf{3})}\langle \mathbf{f}, \mathbf{6}\rangle_{(\mathbf{1})} \\ \Big\downarrow R^{[1]} & \overset{R^{[3]}}{\nearrow} & \Big\downarrow R^{[2]} \\ \Gamma_1 : \langle \mathbf{g}, \mathbf{2}\rangle_{(\mathbf{2})}\langle \mathbf{f}, \mathbf{6}\rangle_{(\mathbf{1})} & & \Gamma_3 : \langle \mathbf{g}, \mathbf{4}\rangle_{(\mathbf{4})}\langle \mathbf{f}, \mathbf{6}\rangle_{(\mathbf{1})} \end{array}$$

Note that now $\Gamma_1 \not\preceq \Gamma_3$.

A useful feature of annotated grammars is their ability to generate longest bad sequences. There are no intersections in the right-hand sides of rewrite rules and thus $\approx$ discerns prefixes that are yielded by distinct rule applications. [4]

[4] This can be very useful if there is a rule R with the left-hand side R_l embedded in the right-hand side R_r. Note that if $R_l \trianglelefteq R_r$ then the subsequence termination criterion is always activated after an application of the rule $R_l \to R_r$. This problem was pointed in [12].

Now we can find the upper bound of a bad sequence length in a computation yielded by a prefix grammar. The proof uses the following lemma.

Lemma 1 *Every computation by an annotated prefix grammar ends either by* Λ *or by a Turchin pair* $\Phi\Theta_0$, $\Phi'\Psi\Theta_0$ *such that there exists a rule* $R_l \to R_r$, *for which* $\Phi \approx R_r$, $\Phi' \approx R_r$, *and* $R_r \neq \Lambda$.

For the proof see Appendix. Note that the proof is for not only alphabetic prefix grammars but for prefix grammars that allow rules of the form $\Phi \to \Psi$. With the help of Lemma 1 we proved that the exact upper bound of a bad sequence length for an annotated prefix grammar is

$$C'_{Max} = |\Gamma_0| * (1 + |R_r^{[0]}| * (1 + |R_r^{[1]}| * (\cdots * (1 + |R_r^{[N]}|) \ldots)))$$

where rules in the sequence $R^{[0]}, R^{[1]}, \ldots, R^{[N]}$ are placed by a non-increasing order with respect to the length of their right-hand sides $|R_r^{[i]}|$ (the proof of this fact is by induction; for details see [11]).

Note that N in the formula $|\Gamma_0|*(1+|R_r^{[0]}|*(1+|R_r^{[1]}|*(\cdots*(1+|R_r^{[N]}|)\ldots)))$ denotes not the cardinality of the set of rewrite rules but the cardinality of the set of the right-hand sides of rewrite rules. Thus when we do the annotation there is no exponential growth of the upper bound.

Example 4 *Let us estimate the length of a longest bad sequence yielded by the grammar* $\mathbf{G'_{LOG}}$ *(Example 3). The length of the initial word is* 1, $|R_r^{[1]}|$ *has the length* 2, *and two rules have the right-hand sides of the length* 1. *The corresponding bad sequence length is* 7.

Now let us build such bad sequence explicitly. For the sake of readability different pairs of the form $\langle$ *letter, number*$\rangle$ *are denoted by different letters (thus* $\langle f, 1\rangle = a$, $\langle g, 2\rangle = c$, $\langle f, 6\rangle = c$, $\langle g, 4\rangle = d$, *and* $\langle h, 8\rangle = e$ *).*

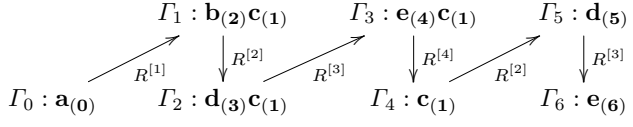
$\mathbf{G'_{LOG}}$*:*

$\Gamma_0 = a$ $\quad$ $R^{[2]} : x \to d$

$R^{[1]} : x \to bc$ $\quad$ $R^{[3]} : x \to e$

$R^{[4]} : x \to \Lambda$

One of the maximal bad sequences is:



Note that the segment Γ_5–Γ_6 *cannot be generated by the initial grammar* $\mathbf{G_{LOG}}$.

If we aim to find embeddings not only in traces generated by single prefix grammars but also in direct products of the traces then usage of $\preceq$ causes some

questions. Namely we must know whether well-binariness is preserved on intersections of the Turchin relation with some wqo. The problem is that the Turchin relation is not well binary on arbitrary computations' subsequences — we only can prove that it is well binary on the whole computations.

Example 5 *Consider the following computation yielded by a prefix grammar.*

$\Gamma_0 : \mathbf{a_{(2)}b_{(1)}c_{(0)}}$	$\Gamma_7 : \mathbf{b_{(5)}c_{(3)}}$
$\Gamma_1 : \mathbf{b_{(1)}c_{(0)}}$	$\Gamma_8 : \mathbf{c_{(3)}}$
$\Gamma_2 : \mathbf{c_{(0)}}$	$\Gamma_9 : \mathbf{b_{(10)}c_{(9)}}$
$\Gamma_3 : \mathbf{b_{(4)}c_{(3)}}$	$\Gamma_{10} : \mathbf{a_{(12)}b_{(11)}c_{(9)}}$
$\Gamma_4 : \mathbf{a_{(6)}b_{(5)}c_{(3)}}$	$\Gamma_{11} : \mathbf{a_{(14)}a_{(13)}b_{(11)}c_{(9)}}$
$\Gamma_5 : \mathbf{a_{(8)}a_{(7)}b_{(5)}c_{(3)}}$	$\Gamma_{12} : \mathbf{a_{(16)}a_{(15)}a_{(13)}b_{(11)}c_{(9)}}$
$\Gamma_6 : \mathbf{a_{(7)}b_{(5)}c_{(3)}}$	$\ldots \quad \ldots$

No two elements of the sequence Γ_0, Γ_5, Γ_{12}, Γ_{21}, ... form a Turchin pair.

The following lemma verifies well-binariness of the intersections (the proof is in Appendix).

Lemma 2 *$\preceq$ contains a wqo T that is well binary on all computations yielded by an annotated prefix grammar.*

T contains Turchin pairs of a special sort. Namely it contains $\langle \Gamma, \Delta \rangle$ such that $\Gamma = \Phi\Theta_0$, $\Delta = \Phi'\Psi\Theta_0$, there exists a rule $R : R_l \to R_r$ with $|R_r| > 0$, $R_r \approx \Phi$, $R_r \approx \Phi'$, and $\Psi[1]$ is never modified in the further computation. So the Turchin relation may not be checked after erasures with no loss of well-binariness.

What is more, existence of T guarantees that the Turchin relation can be intersected with an arbitrary wqo without loss of well-binariness. On the other hand, the idea of intersecting two Turchin relations looks appealing but implies a possible existence of infinite bad sequences with respect to the intersection.

Definition 7. *Let us say that Γ is embedded in Δ with not more than with a single gap if there exist words Φ, Ψ, Θ (maybe empty) such that $\Gamma = \Phi\Theta$, $\Delta = \Phi\Psi\Theta$.*

Let us say that Γ is embedded in Δ ***with not more than with*** *$n+1$* ***gaps*** *if there exist (maybe empty) Φ, Ψ, Θ_1, Θ_2 such that $\Gamma = \Phi\Theta_1$, $\Delta = \Phi\Psi\Theta_2$ and Θ_1 is embedded in Θ_2 with not more than with n gaps.*

Let us give a simple example. **abac** is embedded in **abrac** with not more than a single gap and in **abracadabra** — with not more than two gaps (**abac** is divided into only two parts **ab** and **ac**, but the end of a word is also considered as its part. The end of the word **abracadabra** is not at the same position as the end of **ac**, so the gap between **ac** and the end of the word is also taken into account).

Lemma 3 *If a relation R of word embedding allows only finite number of gaps then it is not well binary on sequences that are yielded by a direct product of prefix grammars $G_1 \times G_2$, even when the grammars are deterministic.*

For the proof see Appendix. So the Turchin relation can be intersected with any relation that is well binary on the whole $\{\Upsilon^*\}$, but not with the other Turchin relation.

5 Turchin's Relation and Subsequence Relation

The Turchin theorem not only guarantees existence of a Turchin pair for every infinite computation but also gives the exponential upper bound of a bad sequence length. In the case of computations yielded by annotated prefix grammars the upper bound of a bad sequence with respect to the subsequence relation coincides with the upper bound of a bad sequence $C'_{Max} = |\Gamma_0| * (1 + |R_r^{[0]}| * (1 + |R_r^{[1]}| * (\cdots * (1 + |R_r^{[N]}|) \ldots)))$ for the Turchin relation. What is more, in a computation yielded by an annotated prefix grammar the lengths of bad sequences with respect to these two relations always coincide.

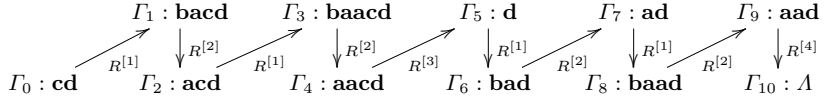
Lemma 4 *The first pair in a computation yielded by an annotated prefix grammar that satisfy the subsequence relation is a Turchin pair.*

For the proof see Appendix.

Lemma 4 may have some practical meaning for systems of program transformation that use the homeomorphic embedding as a branch termination criterion.

Example 6 *The left-hand side of the definition* `f(S(x))=S(f(g(S(x))))` *is embedded in the right-hand side in the sense of the subsequence relation. So an unfolding of the call $f(S(Z))$ yields termination if the homeomorphic embedding is used as a termination criterion. If we use the intersection of this relation with the Turchin relation, the call does not cause the early termination*[5]*. The other way to prevent this too early termination is to use the annotated subsequence relation: it is only enough to annotate function calls to avoid unwanted embeddings with the effect similar to the usage of the Turchin relation intersected with* $\trianglelefteq$.

Then the question emerges if the annotated subsequence relation can prevent too early terminations for every prefix-grammar-generated computation. Namely, whether the annotated subsequence relation allows unfolding to find a trace ending by Λ if Λ is in the language of the prefix grammar. The answer is yes for alphabetic prefix grammars [10] and is negative in the general case when $\mathbf{R} \subset \Upsilon^* \times \Upsilon^*$. To illustrate the last claim, consider the grammar $\mathbf{G_\Lambda}$ of Example 1 if $\Gamma_0 = cd$. Λ belongs to the language of the grammar since



[5] A termination criterion of this type is used in the supercompiler SCP4 [9].

The grammar belongs to the class of annotated grammars. But there are several pairs with respect to $\trianglelefteq$ (and $\prec$) on the trace leading to Λ: e. g. $\Gamma_1 \prec \Gamma_3$. To solve the empty word problem for languages generated by non-alphabetic prefix grammars using the subsequence relation as a termination criterion we need to do some more annotation, which is proved in [10].

Recall that in the case of pairs over $\trianglelefteq$ the upper bound on arbitrary sequences with restricted word length growth is multiple recursive [15]. We show that the upper bound is exact even if the sequences are built by a direct product of two prefix grammars.

Example 7 *Consider the following rewrite grammar (it not necessarily rewrites only prefixes; the grammar is similar to the one described in [15]).*

$R^{[1]} : su \to ss$ $\quad R^{[2]} : tu \to tt$ $\quad R^{[3]} : ts \to tt$
$R^{[4]} : wu \to ws$ $\quad R^{[5]} : tws \to utw$ $\quad R^{[6]} : tw \to ws$
$R^{[7]} : sws \to wt$ $\quad R^{[8]} : sw \to wsss$

If the rules are applied to the initial word $sss \dots swww \dots w$ then the trace of the length $O(B(m,n))$ with no pairs with respect to $triangleleftteq$ is generated.

Now we build a system of two prefix grammars that models the example of H. Touzet (x denotes an arbitrary letter).

1. *The first letter of a word rewritten by the first prefix grammar $\mathbf{G_1}$ represents a current state of the Turing machine.*
2. *The last letter of a word rewritten by the second prefix grammar $\mathbf{G_2}$ represents the end of data $[EOW]$ and is always rewritten into itself.*
3. *The word rewritten by $\mathbf{G_1}$ represents the initial fragment of data which is before the counter of Turing machine. The word rewritten by $\mathbf{G_2}$ represents the final fragment of data which is behind the counter of Turing machine.*
4. *There is an auxiliary set of rules moving the counter to the beginning of the data $\langle State_0 x, y\rangle \to \langle State_0, xy\rangle$.*
5. *There is a rule that starts the rewrite process $\langle State_0, y\rangle \to \langle State_1^F, y\rangle$.*
6. *$R^{[i]} : R_l^{[i]} \to R_r^{[i]}$ are modeled by $\langle State_i^F x, [EOW]\rangle \to \langle State_i^B x, [EOW]\rangle$ (if $i \neq 8$), a rule $\langle State_i^F R_l^{[i]}, x\rangle \to \langle State_0 \Lambda, R_r^{[i]} x\rangle$ and a set of rewrite rules $\langle State_i^F x, y\rangle \to \langle State_i^F xy, \Lambda\rangle$ where x does not coincide with $R_l^{[i]}$.*
7. *The set of rules $\langle State_i^B x, y\rangle \to \langle State_i^B, xy\rangle$ is similar to the one for $State_0$ but the last rule now looks like $\langle State_i^B, y\rangle \to \langle State_{i+1}^F, y\rangle$ instead of $\langle State_0, y\rangle \to \langle State_1^F, y\rangle$.*

If there are two pairs $\langle a_1\Phi, \Psi\rangle$, $\langle a_2\Phi', \Psi'\rangle$ such that $a_1\Phi \trianglelefteq a_2\Phi'$ and $\Psi \trianglelefteq \Psi'$ then $a_1 = a_2$ and $\Phi\Psi \trianglelefteq \Phi'\Psi'$. There can be no such pairs if a_2 is not changed on the trace fragment from $\langle a_1\Phi, \Psi\rangle$ to $\langle a_2\Phi', \Psi'\rangle$ because then $|\Psi'| < |\Psi|$. If a_2 is changed on the trace fragment from $\langle a_1\Phi, \Psi\rangle$ to $\langle a_2\Phi', \Psi'\rangle$ then one of the rules $\langle State_i^F R_l^{[i]}, x\rangle \to \langle State_0 \Lambda, R_r^{[i]} x\rangle$ is applied on the fragment and thus $\Phi\Psi \not\trianglelefteq \Phi'\Psi'$. Therefore the bad sequence length on the trace generated by $\mathbf{G_1} \times \mathbf{G_2}$ with respect to the subsequence relation must be also estimated by $O(B(m,n))$.

Now we can see that transition from only stack transformations to stack-plus-data transformations even in the unary case lifts the computational model from the finite automata up to full power of Turing machines. Thus it becomes interesting to investigate how the popular wqos work on intermediate prefix grammar constructions (as -2 or -1-class prefix grammars [5]), which are widely used in term rewriting theory.

Acknowledgments. The author is grateful to A. P. Nemytykh for fruitful discussions and inspiration on investigating properties of the Turchin relation, and to the anonymous referee for useful remarks that helped to remove vagueness from the paper.

References

1. Albert, E., Gallagher, J., Gomez-Zamalla, M., Puebla, G.: Type-based Homeomorphic Embedding for Online Termination. In Journal of Information Processing Letters, vol. 109(15) (2009), pp. 879–886.
2. Caucal, D.: On the Regular Structure of Prefix Rewriting. Theoretical Computer Science, vol. 106 (1992), pp. 61–86.
3. Bolingbroke, M. C., P. Jones, S. L., Vytiniotis, D.: Termination Combinators Forever. In Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Tokyo, Japan, 2011, pp. 23–34.
4. Higman, G.: Ordering by Divisibility in Abstract Algebras. In Bulletin of London Mathematical Society, vol. 3(2) (1952), pp. 326–336.
5. Jancar, P., Srba, J.: Undecidability Results for Bisimilarity on Prefix Rewrite Systems. In Foundations of Software Science and Computation Structures, LNCS, vol. 3921 (2006), pp.277–291.
6. Klyuchnikov, I.: Inferring and Proving Properties of Functional Programs by Means of Supercompilation. PhD Thesis [In Russian], 2010.
7. Leuschel, M.: Homeomorphic Embedding for Online Termination of Symbolic Methods. In Lecture Notes in Computer Science, vol. 2566 (2002), pp. 379–403.
8. Nash-Williams, C. St. J. A.: On Well-uasi-ordering Infinite Trees. In Proceedings of Cambridge Philosophical Society, vol. 61(1965), pp. 697–720.
9. Nemytykh, A. P.: The SCP4 supercompiler: general structure. Moscow, 2007. 152 p. (in Russian)
10. Nepeivoda, A.: Ping-Pong Protocols as Prefix Grammars and Turchin's Relation, VPT 2013. In Proceedings of First International Workshop on Verification and Program Transformation, EPiC Series, vol. 16, EasyChair, 2013, pp. 74–87.
11. Nepeivoda, A. N.: Turchin's Relation and Loop Approximation in Program Analysis. In Proceedings on the Functional Language Refal. Pereslavl-Zalessky, 2014, pp. 170–192. (in Russian)
12. Puel, L.: Using Unavoidable Set of Trees to Generalize Kruskal's Theorem. In Journal of Symbolic Computation, vol. 8 (1989), pp. 335–382.
13. Simpson, S: Nonprovability of Certain Combinatorial Properties of Finite Trees. In Harvey Friedmans research on the foundations of mathematics, Elsevier Science Publishers, 1985, pp. 87–117.
14. Sørensen, M. H., Glück, R.: An Algorithm of Generalization in Positive Supercompilation. In Logic Programming: Proceedings of the 1995 International Symposium (1995), pp. 465–479.

15. Touzet, H.: A Characterisation of Multiply Recursive Functions with Higman's Lemma. In Information and Computation, vol. 178 (2002), pp. 534–544.
16. Turchin, V. F.: The Algorithm of Generalization in the Supercompiler. In Partial Evaluation and Mixed Computation (1988), pp. 341–353.
17. Weiermann, A.: Phase Transition Thresholds for Some Friedman-Style Independence Results. In Mathematical Logic Quarterly, vol. 53(1) (2007), pp. 4–18.

Appendix

Proof (Lemma 1 on the first Turchin pair).

Let us consider a pair $\Phi_1\Theta_0$, $\Phi_2\Psi\Theta_0$ such that $\Phi_1\Theta_0 \preceq \Phi_2\Psi\Theta_0$, ($\Phi_1 \approx \Phi_2$), and the trace segment ending with $\Phi_2\Psi\Theta_0$ is a bad sequence. According to the properties of annotated grammars, $\Phi_1[1]$ and $\Phi_2[1]$ must be generated by different applications of the same rule $R : x \to R_r$ with $|R_r| > 0$, and if $\Phi_1[1] \approx R_r[i]$ then necessarily $\Phi_2[1] \approx R_r[i]$. Let us denote the prefix $R_r[1]R_r[2]...R_r[i-1]$ as $R^{(i-1)}_{(z)}$ (z is the time index of $R_r[i-1]$). Now turn back to the two applications of R. The result of the former must be of the form $R^{(i-1)}_{(k_1)}\Phi_1\Theta_0$, the result of the latter is of the form $R^{(i-1)}_{(k_2)}\Phi_2\Psi\Theta_0$. They form a Turchin pair and therefore coincide with $\Phi_1\Theta_0$ and $\Phi_2\Psi\Theta_0$.

So $\Phi_1 = R_{r_1}\Phi'_1$, $\Phi_2 = R_{r_2}\Phi'_2$ ($\Phi'_1 \approx \Phi'_2$, and R_{r_1} and R_{r_2} coincide up to the time indices with some right-hand side of a rewrite rule). Let Φ'_1 be non-empty. Then $\exists R', j(\Phi'_1[1] \approx R'_r[j]$ & $\Phi'_2[1] \approx R'_r[j])$, and $\Phi'_1[1] \neq \Phi'_2[1]$. The prefix $R'_r[1]R'_r[2]...R'_r[j-1]$ is denoted as $R'^{(j-1)}_{(z)}$. Now turn back to the R' applications that generate $\Phi'_1[1]$ and $\Phi'_2[1]$. They look as $R'^{(j-1)}_{(l_1)}\Phi'_1\Theta_0$ and $R'^{(j-1)}_{(l_2)}\Phi'_2\Psi\Theta_0$ and form a Turchin pair. This contradicts the choice of $\Phi_1\Theta_0$ and $\Phi_2\Psi\Theta_0$.

Hence $\Phi_1\Theta_0 = R_{r_1}\Theta_0$ and $\Phi_2\Psi\Theta_0 = R_{r_1}\Psi\Theta_0$.

Proof (Existence of a wqo subset of Turchin's relation).

Let $\langle \Upsilon, \mathbf{R}, \Gamma_0\rangle$ be an annotated prefix grammar $\mathbf{G}$. Consider all traces $\{\Phi_i\}_{i=1}^{\infty}$ generated by $\mathbf{G}$ such that $\exists N\, \forall i\, \exists j(i < j$ & $|\Phi_j| \leq N)$.

For every trace J from this set choose the least N that satisfies this property. Due to finiteness of the set $\mathbf{R}$ words generated by the rules from $\mathbf{R}$ can contain finite set of letters. Therefore some word Ψ of the length N must repeat itself (with respect to $\approx$) infinitely in J. The first letter of Ψ is generated by a single rule R with a non-empty right-hand side. Every two results of these applications of R look as $\Delta\Psi$ and $\Delta'\Psi'$ where $\Psi \approx \Psi'$ and $\Delta \approx \Delta'$ for they are same prefixes of the same right-hand side R_r that end at $\Psi[1]$ so $\Delta\Psi$ and $\Delta'\Psi'$ form a Turchin pair.

All other traces $\{\Phi_i\}_{i=1}^{\infty}$ have an infinite growth of the minimal word length: $\forall N\, \exists i_N\, \forall j(j > i_N \Rightarrow |\Phi_j| > N)$. For every such trace and every N choose a minimal i_N such that all successors of Φ_{i_N} never have the length less than N: $\forall j(j < i_N \Rightarrow \exists k(k \geq j$ & $|\Phi_k| < N))$. So $|\Phi_{i_N-1}| < N$, $|\Phi_{i_N}| \geq N$, and Φ_{i_N} is generated from its predecessor by some R with a non-empty right-hand side, $|R_r| \geq 2$: $\Phi_{i_N} = R_{r(l)}\Phi^-_{i_N-1}$ where $\Phi^-_{i_N-1}$ is a suffix of Φ_{i_N-1}. $\Phi^-_{i_N-1}$ stays constant because $|\Phi_{i_N-1}| < N$. All the elements of $\{\Phi_{i_N}\}_{N=1}^{\infty}$ begin with a

non-empty right-hand side of a some rewrite rule, therefore exists an infinite subsequence $\{\Phi_{i_K}\}_{K=1}^{\infty}$ of $\{\Phi_{i_N}\}_{N=1}^{\infty}$ such that all elements of $\{\Phi_{i_K}\}_{K=1}^{\infty}$ begin with the right-hand side of a same rule. Every two elements of $\{\Phi_{i_K}\}_{K=1}^{\infty}$ form a Turchin pair.

The set T of pairs of these two sorts is a wqo on traces generated by an annotated prefix grammar.

Example 8 (Non-well-binariness of N-gaps relation) *Proof. Let us consider a class of grammars* $\mathbf{G}^{[n]}$ *on pairs of words in the alphabet* $\{a_1, \ldots, a_n, A_1, \ldots, A_n, e, E\} \times \{a_1, \ldots a_n, A_1, \ldots, A_n, e, E\}$*, with the initial word* $\langle e, A_1 A_2 \ldots A_n E\rangle$ *and the following rewrite rules:*

$R^{[00]} : \langle e, a_1\rangle \to \langle E, a_1 a_1\rangle$
$R^{[01]} : \langle E, A_1\rangle \to \langle e, A_1 A_1\rangle$
$R^{[02]} : \langle e, A_1\rangle \to \langle a_1 e, \Lambda\rangle$
$R^{[03]} : \langle E, a_1\rangle \to \langle A_1 E, \Lambda\rangle$
$\ldots$
$R^{[i0]} : \langle a_i, a_i\rangle \to \langle \Lambda, a_i a_i\rangle$
$R^{[i1]} : \langle A_i, A_i\rangle \to \langle \Lambda, A_i A_i\rangle$
$R^{[i2]} : \langle a_i, A_i\rangle \to \langle a_i a_i, \Lambda\rangle$
$R^{[i3]} : \langle A_i, a_i\rangle \to \langle A_i A_i, \Lambda\rangle$
$R^{[i4]} : \langle a_i, a_{i+1}\rangle \to \langle \Lambda, a_i a_{i+1} a_{i+1}\rangle$
$R^{[i5]} : \langle A_i, A_{i+1}\rangle \to \langle \Lambda, A_i A_{i+1} A_{i+1}\rangle$
$R^{[i6]} : \langle a_i, A_{i+1}\rangle \to \langle a_{i+1} a_i a_i, \Lambda\rangle$
$R^{[i7]} : \langle A_i, a_{i+1}\rangle \to \langle A_{i+1} A_i A_i, \Lambda\rangle$
$\ldots$
$R^{[n0]} : \langle a_n, e\rangle \to \langle \Lambda, a_n e\rangle$
$R^{[n1]} : \langle A_n, E\rangle \to \langle \Lambda, A_n E\rangle$
$R^{[n2]} : \langle a_n, E\rangle \to \langle a_n a_n, e\rangle$
$R^{[n3]} : \langle A_n, e\rangle \to \langle A_n A_n, E\rangle$

For every N there exists some n such that $\mathbf{G}^{[n]}$ *yields a trace with no pair* $\langle \Phi_1, \Psi_1\rangle$, $\langle \Phi_2, \Psi_2\rangle$ *such that* Φ_1 *is embedded in* Φ_2 *and* Ψ_1 *is embedded in* Ψ_2 *with not more than N gaps.*

Proof (The proof of Lemma 4). Consider Φ_1 and Φ_2 such that $\Phi_1 \trianglelefteq \Phi_2$, there are no pairs with respect to the subsequence relation in the trace before Φ_2, and Φ_1 is embedded into Φ_2 with $n+1$ gaps (up to time indices). Let Θ_0 be their common suffix. Then $\Phi_1 = A_1 A_2 \ldots A_n \Theta_0$ and $\Phi_2 = B_1 A'_1 B_2 A'_2 \ldots B_n A'_n B_{n+1} \Theta_0$, where $A_i \approx A'_i$ for all i from 1 to n. Consider the words where letters $A_n[1]$ and $A'_n[1]$ are generated. The grammar features guarantee that both letters are generated by a same rule $R : x \to R_r$, so the words look as $\Delta A_n \Theta_0$ and $\Delta' A'_n B_{n+1} \Theta_0$. $\Delta \approx \Delta'$ for they are same prefixes of the same right-hand side. This implies that $\Delta A_n \Theta_0 \trianglelefteq \Delta' A'_n B_{n+1} \Theta_0$. In the computation Φ_1 and Φ_2 are the first pair with respect to the subsequence relation, consequently $i = 1 = n$, and $\Phi_1 = A_1 \Theta_0$, and $\Phi_2 = A'_1 B_{n+1} \Theta_0$, so $\Phi_1 \trianglelefteq \Phi_2$.

Algebraic Structures of Programs: First Steps to Algebraic Programming

Nikolai N. Nepeivoda

Program System Institute of RAS
Pereslavl–Zalessky, Russia

Abstract. Basic algebraic notions are introduced which can be used to describe and to transform program-data complexes for traditional languages, restricted computation, non-numeric computation and modeling. General algebraic program systems (GAPS) are free of assumptions what computing (programming) systems have as their particular constructs. First of all we argue why current technique is sometimes not adequate. Then we show how to interpret notions of abstract algebra (groupoid structure) as super-von Neumann computational structure and how to express many useful structures and notions in an algebra. Then GAPS are introduced and basic mathematical results are stated including the precise criterion when a given system of actions over programs can be added to given programming language. Various examples of GAPS are given. And at last we show possible primitives of 'structured algebraic modeling' (programming).

Keywords: program algebras, reversive programs, near-reversivity, algebraic computing

The general sructure of paper

Our introduction gives an informal insight why so abstract algebraic formalism is reasonable for some problems of programming, what are its motivations and main difference from other types of program algebras.

First of all we state basic analogies between program and computation structures and algebraic notions in groupoid (a structure with non-associative binary operation). The main result here is that elements of groupoids can be viewed as well as data, actions, actions over actions and so on. Thus we get a 'functional' programming on entities which are not necessary functions. The side results are that many control and data structures can be expressed purely algebraically without any reference to particular control structures.

Then we define GAPS (Generic algebraic program structure) — a very abstract algebraic description of programs.

Basic properties of GAPS are studied and it is proved that many traditional languages can be represented as GAPS because they model the λ-calculus. There are also example of dully non-traditional computing structures modeled by

GAPS. Two precise criteria when a given language can be enriched or extended by the given system of program transformations are stated.

And at last we define structures which allow to construct near-reversive computations: reversive data types, mirrors, interruptors, crystals, wheels. An example how to 'program' and to design a scheme by these structures is given.

1 Introduction

Reversible computing was the big bang inspiring this investigation. More precisely, it was a cognitive dissonance between brilliant ideas of von Neumann, Landauer, Toffoli, Feynman, Merkle and more than 30 years of stagnation in "applied research". Sorry, it is not a stagnation, but intensive running in a squirrel cage wheel (maybe different for different 'schools'). Usually such effect is induced by some assumption which is so common that it becomes almost invisible. But really it stands on the way, pollutes this way but nobody is brave enough to point to this obstacle (a *sacred cow*, as it is called in [5]).

Here this cow is that computations are binary. Remember that mathematical model of invertible functions is a *group*. Works (including mines) in this direction were based on groups till 2012.

Bennett pointed out that reversivity can beat Landauer limit only if the control system is also based on invertible transformations. By control system we mean here a system of entities organizing the execution of elementary actions as elements of a computing system. These ones can be the statements and structures of programming language, the connectors in a physically realized scheme and so on.

It is necessary to remember the difference between two notions. **Reversive** computing, reversive actions are fully invertible. a^{-1} can be used both before and after a, they are in some sense bijective. **Reversible** ones are retractable ones, they are injective, this action can be undone but not prevented [8]. The store for all intermediate results is a way to provide reversibility but not reversivity. The multiplication of integers by 2 is reversible function but not reversive one.

There are important and fundamental invertible commands which cannot be represented as functions and cannot be embedded into group structure. First of all this is the *absolute mirror* or inversion of a program segment.

We use the list postfix notation for function or action application: $(a\ F)$ where F is an action applied to a.

Definition 1. ***Absolute mirror*** *is the action transforming a preceding action into its inversion:* $(f\ M) = f^{-1}$. *UNDO is the action undoing the last block of actions:* $\{a_1;\ \ldots\ a_n\}$; *UNDO* = *EMPTY.*

Example 1. M and UNDO are non-associative entities and almost never can be embedded into structure of (semi)group:

$$((a \circ b) \circ M) = (b^{-1} \circ a^{-1}) \qquad (a \circ (b \circ M)) = (a \circ b^{-1})$$

$$\{\mathsf{a};\ \mathsf{b}\};\ \mathsf{UNDO} \neq \mathsf{a};\ \mathsf{b};\ \mathsf{UNDO}.$$

There is another aspect of a problem. Program and algorithmic algebras is a classic branch of computer science. They take start at 60'ths. Glushkov [1] defined algorithmic algebras using operators corresponding to base constructions of structured programming, Maurer [2] introduced and studied abstract algebras of computations (program generic algebras, PGA) more like to computer commands, based on semigroups and on operators representing **goto**s. Various kinds of algorithmic and dynamic algebras follow these two basic ideas.

For main results of Glushkov approach we refer to [3, 4]. For current investigations in PGA a good summary are [6, 7]. There are more than hundred works on algebras of programs cited in [11] where a comprehensive survey up to 1996 is done. Almost all these works are devoted to Turing complete systems. But even reversible systems cannot be Turing complete [13, 14].

Full reversivity restricts class of problems more severely. Factorial, multiplication and division of integer numbers are irreversible. All arithmetic operations on standard representation of real numbers are not reversive. If elementary actions are invertible a problem itself can be not invertible.

Example 2. Sorting is irreversive because we forget initial state and it cannot be reconstructed from sorted array. Assembling of Rubik cube is irreversive due to similar reasons.

Even if we go beyond the problem of heat pollution during computations reversivity arises due to development of computer element base. Quantum computations are reversive. Molecular computations are reversive. Superconductor computations often are reversive. Nanocrystal computations are reversive. So studying of computations where majority of operations are reversive is necessary.

Concrete functions are often defined through common λ-notation. This does not mean that all our constructs are based on λ-calculus.

2 Algebraic structures from the point of view of programming

Definition 2. ***Signature*** *σ is a list of symbols: functions (binary functions can be used as infix operations), constants and predicates. There is a metaoperator arity(s) giving for each function or predicate symbol its number of arguments. If there is the predicate $=$ it is interpreted as equality. $s \in \sigma$ where sigma is a signature means that symbol s is in σ.*

Algebraic system **S** *of signature σ is a model of this signature in the sense of classical logic. We have a data type (or nonempty set) S called* ***carrier*** *and second order function ζ such that for function symbols $\zeta(f) \in S^{\mathrm{arity}(f)} \to S$, $\zeta(f) \in S^{\mathrm{arity}(f)} \to \{\mathrm{false}, \mathrm{true}\}$ for predicates, $\zeta(c) \in S$ for constants.*

Definition 3. ***Groupoid*** *is an algebraic system of signature $\langle \circledast, = \rangle$, where $\mathrm{arity}(\circledast) = 2$. Its carrier is often denoted G, symbol for its binary operation can be various for different groupoids. Element e is an* ***identity*** *(or* ***neutral***

element**) if* $\forall x\, x \circledast e = e \circledast x = x$. *Element* 0 *is* ***zero *if* $\forall x\, x \circledast 0 = 0 \circledast x = 0$. ***Left identity*** *is such* a *that* $\forall x\, a \circledast x = x$. ***Left zero*** *is such* a *that* $\forall x\, x \circledast a = a$. *Analogously for right identity and zero.* ***Idempotent*** *is such* a *that* $a \circledast a = a$. *Groupoid is a* ***semigroup*** *if its operation is associative. It is commutative if its operation is commutative. It is* ***injective*** *if*

$$\forall x, y, f\, x \circledast f = y \circledast f \Rightarrow x = y.$$

Semigroup is a ***monoid*** *if it has an identity element. Monoid is a* ***group*** *if there is identity element and for each* x *exists* y *such that* $f \circledast g = g \circledast f = e$. *Bigroupoid is an algebraic system with two binary operations.*

Morphism of algebraic structures is a map of their carriers preserving all functions and constants, and truth (but not necessary falsity) for all predicates. **Isomorphism** is a bijective morphism preserving falsity.

Now we comment this notions from informatic point of view.

Non-associative groupoid gives a set of expressions isomorphic to ordered directed binary trees or Lisp lists.

Example 3. $((a \circledast (b \circledast c)) \circledast (a \circledast d))$ is an expression for Lisp ((a b c) a d) id $\circledast$ is CONS. It defines the binary tree on fig. 1.

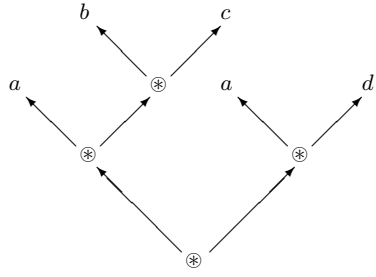


Fig. 1. $((a \circledast (b \circledast c)) \circledast (a \circledast d))$ as a tree

Abstract algebra is extremely valuable for program architects and analysts because each isomorphism gives another representation for data, each morphism gives a structure which can be viewed as approximation for full data or valuable analogy.

If operation is commutative tree becomes inordered and direct analogies with lists vanish. If it is associative list becomes linear and tree becomes a one-dimensional array. Any associative operation can be interpreted as function composition. It follows from classical theorem

Theorem 1. *Each semigroup is isomorphic to semigroup of functions with composition as operation. [10]*

This theorem gives a full criterion when a space of actions can be viewed as a space of functions: associativity. Thus Example 1 shows that *actions not always can be modeled as functions*. Because a semigroup operation always can be viewed as function composition it is usually denoted $\circ$.

The following example shows central role of semigroups in algebraic informatics.

Example 4. Each collection of functions or relations closed under composition is a semigroup. Thus we can treat functions definable by programs as the single semigroup even our language is strongly typized. Programs are simply functions undefined on data not belonging to types of their arguments. Each finite automate can be treated as a finite semigroup and vice versa any finite semigroup can be viewed as a finite automate. Actions in each program language with sequential composition (usually denoted by semicolon) also form a *syntactic* semigroup even their effect cannot be treat as a function.

The last sentence is the second keystone for very abstract notion of program algebra independent from assumptions on concrete operators of language and of functionality of actions.

One example of a commutative non-associative algebra of actions will be used below and is sufficiently simple and expressive to show many peculiarities.

Example 5. A commutative groupoid formalizing a simple game. Its carrier is the set of three elements {well, scissors, paper} (denoted {w,s,p}). Our operation gives for each pair the winner.

$$w \circledast s = w; \quad w \circledast p = p; \quad s \circledast p = s; \quad x \circledast x = x.$$

$(w \circledast s) \circledast p = w \circledast p = p$, $w \circledast (s \circledast p) = w \circledast s = w$. So simultaneous (or independent) actions of two players effect in commutativity of the operation.

So commutativity of a semigroup means that our actions are functional and independent. From the point of view of computations they can be executed in various ways (sequentially in any order; (partially) parallel; and so on). From the logical point of view our actions can be viewed as spending of a money-like resource (Girard's linear logic [9]). We take into account only a total amount of resources in the 'account'. Each operation spends them independently.

3 Groupoid as a computing structure

A groupoid can be viewed as a 'functional'[1] computing structure of 'super-von-Neumann' kind. Each element a of groupoid can be viewed also as the action with the effect $\lambda x.\,(x \circledast a)$. Furthermore it is actions on actions and so on. Data and commands are the same.

So now we look on many interesting elements and properties from the point of view of 'algebraic computer'.

[1] But remember that actions not always are functions!

1. Associativity means that our actions are without 'side effects' and can be viewed as functions.
2. Unity e means 'do nothing'. Moreover each right unity $((x \circledast e) = x$ is 'no operation' command.
3. 0 is the fatal error. Mathematically 0 in semigroups of relations is the empty relations (function which is never defined). If groupoid has more than one element that 0 is not (left, right) identity.
4. Left zero $(z \circledast x) = z$ is an output, the final result of a computation. More precisely if there is the zero then output can be described as $\forall x\,(x \neq 0 \supset (z \circledast x) = z)$ (**left near-zero**).
5. Right zero $(x \circledast z) = z$ is at the same time so called 'quine' (program giving itself) and an input, the initial value overriding all earlier. More precisely if there is the zero then input can be described as $\forall x\,(x \neq 0 \supset (x \circledast z) = z)$ (**right near-zero**).
6. Idempotent $(z \circledast z) = z$ is a pause.
7. Right contraction $(a \circledast f) = (a \circledast g) \supset f = g$ is practically almost useless property: each program acts differently on each elements than any others. But there is an important exception. If our operation is associative and each element has the inversion $a \circ a^{-1} = e$; $a^{-1} \circ a = e$ then our semigroup becomes a group. Each space of bijective functions can be viewed as a group and vice versa. Elementary fully invertible actions on some data type form a group. In a group we can 'prevent' an action not only to **undo** it.
8. If $(x \circledast a) \circledast \tilde{a} = x$ then $\tilde{a}$ is a **weak right inverse** for a. It grants undoing of a.
9. A **one-way pipe** p is such element that

$$(x \circledast p) = y \supset (y \circledast m) = 0.$$

10. A subgroupoid can be viewed as a block of program or construction.
11. Direct product of groupoids means that computation can be decomposed into independent branches corresponding components of direct product.

So we see

Algebraic programming is functional and super-von-Neumann by its essence.

Each groupoid generates the semigroup of actions

Definition 4. ***Action*** *of groupoid element sequence* $f_1, \ldots, f_n$ *is a function*

$$\varphi_{f_1;\ldots;f_n} = \lambda x.\,(\ldots((x \circledast f_1) \circledast f_2)\ldots) \circledast f_n).$$

This semigroup not always fully describes program actions.

Lemma 1. *Actions of groupoid form a semigroup.*

Proof. Composition of $\varphi_{f_1;\ldots;f_n}$ and $\varphi_{g_1;\ldots;g_k}$ is the action corresponding to

$$\varphi_{f_1;\ldots;f_n;g_1;\ldots;g_k}.$$

To make algebraic computation more structured we will partially decompose our groupoid into subsystems. Three main kinds of subsystems are:

1. block;
2. connector;
3. computation-control pair

Block is a subgroupoid. All actions with block elements do not lead out of block. In the simplest case block has inside all its outputs and often also inputs as corresponding algebraic values. To make our algebraic computation structured we demand

If x and y are from different blocks then $(x \circledast y) = 0$.

There is an important transformation of groupoid. *Dual groupoid* $\mathbf{G}'$ to $\mathbf{G}$ is the groupoid with the same carrier and operation $(x * y) = (y \circledast x)$. In both program and technique realizations dual groupoid is implemented by the same structure where arguments (signals) are exchanged. Dual to associative system is associative.

$$(a * (b * c)) = (c \circ b) \circ a; \quad ((a * b) * c) = c \circ (b \circ a).$$

Nevertheless when computing system is modeled as a groupoid dual system to a subsystem is to be represented by another block. In this case we denote the element of dual corresponding to a as a'. $'$ is not an internal operation. In realization a and a' usually will be the same value, element or signal.

Connectors transfer information and control between blocks. They are outside connecting blocks. We demand for connectors c that if $(x \circledast c) = y$ then $(x \circledast y) = 0$. Two most valuable kinds of connectors are mirrors and one-way pipes.

A **mirror** m is such an element that

$$(x \circledast m) = y \equiv (\tilde{y}' \circledast m) = \tilde{x}'.$$

It seems now that mirrors and pipes are to be the only connectors admitted in structured algebraic computations.

A **computation-control pair** is a groupoid decomposed into direct product $G1 \times G2$ where $G1$ has no inputs and outputs. Realizing this pair we are to grant that any output value computed in $G2$ will interrupt process in $G1$.

Example 6. Let a clever, brave and polite black cat is creeping up to mouse in a black room. $G1$ here means a crawling process and $G2$ means a sensor interrupting crawling and initializing attack when the mouse detects the cat. We have the following diagram Crawling and attack easily are described by semigroups (see [18]). Interruption and pipe are non-associative operators. Sensor is almost associative and reversive: its actions form a group in which one element is replaced by output.

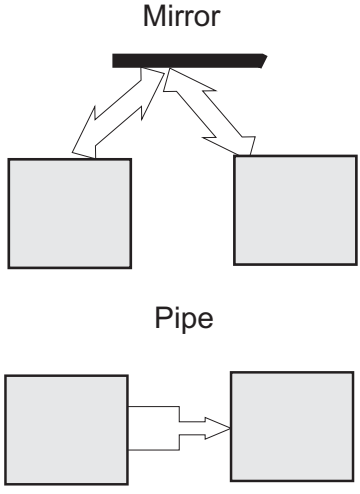


Fig. 2. Connectors

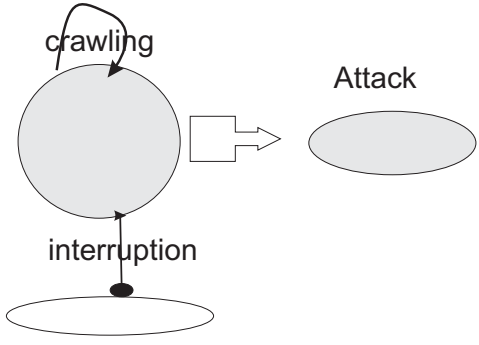


Fig. 3. Black polite cat

Let us consider another interesting possibility arising because high order essences arise in algebraic computing from the very beginning, on level of elementary actions.

It is known that current program systems are like to dinosaurs burdened by gigatons of code. Usually explosive increasing of size of program systems is considered as objective inevitable factor. But there is a side way.

Example 7. High order transformations can shorten programs and sometimes computations in tower of exponents. For example compare two sequences of definitions below

$$\begin{array}{c} \Phi_1 = \lambda f.\, \lambda x.\, ((x\ f)\ f); \\ \Phi_{n+1} = \lambda \Psi_n.\, \lambda \Psi_{n-1}.\, ((\Psi_{n-1}\ \Psi_n)\ \Psi_n). \end{array} \tag{1}$$

$$(x\ (f\ (\Phi_1 \ldots (\Phi_{n-2}\ (\Phi_{n-1}\ \Phi_n^k)) \ldots))) = (x\ \ f^{2^{2^{\cdots^2}}}\ (k \text{ times})) \tag{2}$$

So groupoid induces at least two additional structures: finite order functionals and the semigroup of actions. Higher order functionals need no extra support here. To describe algebraic system effectively it suffices to add an operations converting a system of elements into a single block.

4 General algebraic program structure (GAPS)

Considerations above lead to the formal notion. Let us denote an operation of applying f to x by $x \star f$, an operation composing two elements into the single block $a \circ b$.

Definition 5. ***General algebraic program structure*** *(GAPS) is a bigroupoid where the following holds:*

$$((x \circ y) \circ z) = (x \circ (y \circ z)) \tag{3}$$
$$((x \star f) \star g) = (x \star (f \circ g)) \tag{4}$$

If GAPS has unity and (or) zero they are to satisfy equations

$$(0 \star x) = (x \star 0) = 0 \tag{5}$$
$$(x \star e) = x \tag{6}$$
$$x \circ 0 = 0 \circ x = 0 \tag{7}$$
$$e \circ x = x \circ e = x. \tag{8}$$

So application is not necessary associative (and in fact we don't need composition if it is associative) and composition is associative and gives us possibility to encapsulate a number of elements into the single one. Unity is a right unity.

Lemma 2. *Each groupoid* **G** *can be extended up to GAPS.*

Proof. Consider the following Horn theory. It has constants for all elements of **G**. It includes the diagram of **G** (i. e. the set of true closed elementary formulas), a new constant **B** and the axioms

$$((x \star f) \star g) = (x \star (f \star (g \star \mathbf{B}))) \tag{9}$$

$$(f \star ((g \star (h \star \mathbf{B})) \star \mathbf{B})) = ((f \star (g \star \mathbf{B})) \star (h \star \mathbf{B})) \tag{10}$$

Its initial model is a desired GAPS.

We are sure that GAPS is in algebraic sense a minimal system describing every imaginable collection of elementary actions and programs (constructs) composed from them.

The second operation allows us to formulate many properties more expressively and effectively.

Definition 6. *An element $x \neq 0$ is a divisor of zero if there is such $y \neq 0$ that $x \circ y = 0$. An element y is a right inverse for x if $x \circ y = e$. An element x^{-1} is the inverse of x if $x \circ x^{-1} = x^{-1} \circ x = e$. Two elements are mutually inverse if $a \circ b \circ a = a,\ b \circ a \circ b = b$.*

So each of mutually inverse elements grants undoing (prevention) for another on codomain (domain) of the last one (if they are partial functions).

The next simple theorem shows that every system of functions (every semigroups) can be enriched by every system of total program transformations, so it is called **theorem on abstract metacomputations**. It needs some preliminary discussion.

Remember the notion of enrichment for algebraic systems (its direct analogy for classes in programming is called specialization). Let a signature σ_1 extends a signature σ. An algebra $\mathbb{A}_1$ in σ_1 is enrichment of $\mathbb{A}$ in σ if all carriers, functions, constants and predicates from σ are untouched.

Let there is a morphism $\alpha : \mathbb{G} \to (G \to G)$ of the semigroup $\mathbb{G}$ into the semigroup of maps of its carrier such that $(e\ \alpha) = \lambda x.\, x$, $(0\ \alpha) = \lambda x.\, 0$. From the programmer's point of view it gives interpreter, translator, compiler or supercompiler giving an executable module for a high order program.

An example of such morphism. $(x\ \alpha) = \lambda x.\, 0$ for all divisors of zero. $(x\ \alpha) = \lambda x.\, x$ for others. $x \star y = x \circ y$.

Theorem 2. *Each semigroup $\mathbb{G}$ can be enriched to GAPS such that $(x \star f) = (x\ (f\ \alpha))$.*

Proof. Laws for GAPS hold. Non-trivial is only (10).

$$((x \star f) \star g) = ((x\ (f\ \alpha))\ (g\ \alpha)) = (x\ ((f\ \alpha) \circ (g\ \alpha))) =$$
$$(x\ (f \circ g\ \alpha)) = (x \star (f \circ g))$$

□

Lemma 3. *For each action φ of groupoid there is an element α such that $(x \star \alpha) = (x\ \varphi)$.*

Proof. Let $f_1, \ldots, f_n$ be an arbitrary action. Applying (4) and associativity of $\circ$ get

$$(\ldots((x \circledast f_1) \circledast f_2)\ldots) \circledast f_n) = (x \circledast f_1 \circ f_2 \cdots \circ f_n).$$

□

Example 8. Some 'natural' assumptions destroy GAPS. For example if $\forall f\ (e \star f) = f$ two operations coincide.

$$(x \star y) = ((e \star x) \star y) = (e \star (x \circ y)) = x \circ y.$$

Consider this phenomenon. 0 is the fatal error and we cannot do with it inside of system. e can be interpreted as an empty program but program transformer can generate non-empty code starting from empty data. See example 14 below.

Example 9. Direct application of process to enrich groupoid to GAPS almost always leads to infinite structure. Often it is possible to remove superfluous constructs and get the finite GAPS.

Consider groupoid from the example 5.First of all we write down all different actions of groupoid in the form `shorter result: longer sequence of elements`.

w: wwp, s: wss, p: psp, ws: wws, sw: www, ps: sss, sp: pss, wp: ppp, pw: pwp,wsp: pps, spw: pww, pws: sws.

Semigroup of functions of actions consists of twelve elements and is not commutative though initial groupoid is commutative. Often the groupoid operation $*$ can be extended to GAPS by different ways. For example here there are at least two extensions:

(ws * sw) = wssw = wsw = ws;
(ws * sw) = (ws)*(s*w) = (ws)w = (w*w)(s*w) = ww = w.

Now consider a problem when and how we can join some system of program transformations with a given program system (= a semigroup of program) and to get the single language of metaprogramming. There are three natural subproblems.

1. How to join a language with a system of transformation not changing the language (the semigroup)?
2. How to preserve some sub-language (sub-semigroup) maybe converting other constructions into transformations?
3. How to extend a language to a metalanguage not changing notions inside of the given language?

Definition 7. *Let some system of transformations* $\mathbb{A}$ *is given as system of functions on carrier of a semigroup* $\mathbb{G}$ *described by theory* Th *and desired properties of transformations are written down as an axiomatic theory* Th_1. *It is* ***conservative*** *if there is a GAPS enrichment* $\mathbb{AG}$ *of* $\mathbb{G}$ *such that every* $\varphi \in \mathbb{A}$ *is represented as an action of groupoid* $(x \star \alpha) = (x\ \varphi)$ *for some* α. *It is* conservative over the subgroup $\mathbb{G}_0$ *if it is conservative and in* $\mathbb{AG}$ $a \star b = a \circ b$ *for all elements of* $\mathbb{G}_0$. *It is* ***admissible*** *if there is a semigroup* $\mathbb{G}_1$ *such that* $\mathbb{G} \subseteq \mathbb{G}_1$ *and* $\mathbb{A}$ *is conservative for* $\mathbb{G}_1$.

Let Th_P *is a theory* Th *in which all quantifiers are restricted by unary predicate* P: $\forall x\, A(x)$ *is replaced by* $\forall x\, (P(x) \supset A(x))$; $\exists x\, A(x)$ *is replaced by* $\exists x\, (P(x) \& A(x))$.

$\mathbb{A}$ ***strongly admissible*** *if it is admissible and for resulting algebra theories* Th_1 *and* Th *remain valid.*

Lemma 4. *Collection of actions* $\mathbb{A}$ *is conservative iff its closure is isomorphic to subsemigroup of* $\mathbb{G}$.

Proof. By 3 if enrichment is successful then each element of the semigroup generated by $\mathbb{A}$ represents action of some element of $\mathbb{G}$. Thus closure of $\mathbb{A}$ is embedded into $\mathbb{G}$.

Vice versa, if the closure of $\mathbb{A}$ can be embedded into $\mathbb{G}$ then each action from $\mathbb{A}$ can be represented by its image by this embedding.
□

Example 10. Let there be only one action: inversion of programs **M** such that $((a\ \mathbf{M})\ \mathbf{M}) = a$. To enrich a semigroup od programs by this action is possible iff there is an element of order 2 in this semigroup: $f \neq e \& f \circ f = e$. No matter how this f acts as function.

Lemma 5. *Collection of actions* $\mathbb{A}$ *is conservative over* $\mathbb{G}_0$ *iff there is monomorphism* ψ *of its closure into* $\mathbb{G}$ *such that for each* f *such that* $(f\ \psi) \in \mathbb{G}_0$ $(a\ f) = (a \circ (f\ \psi))$ *holds.*

Example 11. Using this criterion we can test possibility of enrichment up to language of metacomputations considering strings as programs and remain untouched the sublanguage of numerical computations.

The following theorem is proved in [18]. Its proof requires model-theoretic technique, is long and resulting construction is not always algorithmic one.

Theorem 3. *(2012–2014) Let a system of actions* $\mathbb{A}$ *is described by a theory* Th_1, *and* Th *is a theory of semigroup* $\mathbb{G}$ *and* P *is a new unary predicate.*
Then $\mathbb{A}$ *is strongly admissible over* $\mathbb{G}$ *iff there is a partial surjection* $\psi : \mathbb{G} \to \mathbb{A}$ *such that* $(g_1 \circ g_2\ \psi) = (g_1\ \psi) \circ (g_2\ \psi)$ *if all results are defined and the theory* $\mathrm{Th}_1 \cup \mathrm{Th}_P$ *is consistent.*

There is an important consequence of this theorem.

Proposition 1. *Every set of actions* $\mathbb{A}$ *is admissible over* $\mathbb{G}$ *if both theories consist only from facts (true formulas of the form* $[\neg](a\{\circ,*\}b) = c$*).*

Example 12. Consider an additive group $\mathbb{Z}_3$ and add actions of its objects $\{0,1,2\}$ as well, scissors and paper from example 5. Their actions can be described as functions on $\mathbb{Z}_3$ with values 002, 011, 212. Now we extend wsp-groupoid to twelve elements semigroup as in example 9. To conform with it is necessary to add identity which don't belongs to this closure and denote this monoid $\mathbb{C}$. Consider a direct product of $\mathbb{C} \times \mathbb{Z}_3$ and define actions as follows.

$$(\langle c, x\rangle \star \langle d, y\rangle) = \langle c \circ d, (x + y\ d)\rangle\,.$$

This GAPS contains $\mathbb{Z}_3$. Elements of $\mathbb{C}$ can be considered as commands and elements of $\mathbb{Z}_3$ as data. $(x\ d)$ is application of sequence of actions to an element coded by x and coding of the result. Commands transform as a semigroups but their effects as groupoid.

There is another way to define GAPS on the same carrier.

$$(\langle c, x\rangle \star \langle d, y\rangle) = \langle (c \star d), (x + y\ (c \star d))\rangle\,.$$

Thus the problem is when this extension is computable. It can be infinite and non-computable even for finite theories, finite semigroup and collection of actions. Though this theorem is pure one it gives a valuable negative criterion.

A practical consequence. *If a system of program transformations destroys properties of program or forced to make different programs equal it is incorrect.*

There is a particular case when our extension is semi-computable.

Definition 8. ***Horn formula*** *(quasi-identity) is a formula*

$$\forall x_1, \ldots, x_k\,(Q_1 \& \cdots \& Q_n \supset P),$$

where x_i *all its variables, and all* Q_i, P *are predicates.*

If our theories consist of Horn axioms then GAPS can be constructed as the factorization of the free GAPS according to provable identity of terms (the initial model).

Example 13. Because λ-calculus lies in foundations of the modern mathematical theory of Turing-complete program systems (see [11]) it suffices to construct a model of λ-calculus as a GAPS. To do this we take an equivalent representation of λ-calculus as combinatory logic and take its basis $\{\mathbf{I}, \mathbf{B}, \mathbf{C}, \mathbf{S}\}$ [12] described in our terms as

$$(x \star \mathbf{I}) = x$$
$$(x \star (f \star (g \star \mathbf{B}))) = ((x \star f) \star g)$$
$$(x \star (f \star (g \star \mathbf{C}))) = ((x \star g) \star f)$$
$$(x \star (y \star (z \star \mathbf{S})) = ((x \star y) \star (x \star z)).$$

Adding the axiom of associativity of composition **B** (10) and the definition

$$f \circ g = (f \star (g \star \mathbf{B}))$$

we get a model of λ-calculus and using this one we get models for all standard and almost all non-standard programming languages.

To get a model for typed λ-calculus it suffices to add zero 0 and to redefine $\star$ as zero when types are not conforming. To get the identity it suffices to add the following axioms:

$$(\mathbf{I} \star (f \star \mathbf{B})) = f \qquad (f \star (\mathbf{I} \star \mathbf{B})) = f.$$

Example 14. One more example how to turn program system into GAPS. Let we have an algorithmic language in which symbols are operators and concatenation of strings is composition of programs (e.g. Brainfuck Brainfuck). Then empty string is program doing nothing. It can be naturally represented as GAPS. $a \circ f$ is simply concatenation. $a \star f$ will be defined as follows. If f is a correct program then its value on a is $a \star f$. If f yields an error or it is syntactically incorrect then our value is 0.

We see that the result of action over an empty program can be arbitrary.

Example 15. In 1972 one research stopped one little step before GAPS [16].

Consider the alphabet $\{K, S, (,)\}$. Its symbols translate into combinators as

$$(K\ \varphi) = \mathbf{K} \quad (S\ \varphi) = \mathbf{S} \quad (\text{`('}\ \varphi) = \mathbf{B} \quad (\text{`)'}\ \varphi) = \mathbf{I}$$

The result of string translation is defined recursively: (a is a symbol, σ is a string):

$$(a\sigma\ \varphi) = ((\sigma\ \varphi) \star (a\ \varphi)).$$

This interpretation turns the combinatory logic into a Brainfuck-like programming language. But resulting GAPS is not a GAPS for the combinatory logic. It also includes syntactically incorrect constructs like))))**SK**((.

Example 16. If our semigroup is a monoid then universal function in GAPS is trivial $\mathbf{U} = e$:

$$(x \star (f \star \mathbf{U})) = (x \star f),$$

A partial evaluator is not trivial:

$$(f \star (x \star \mathbf{PE})) = (x \star f),$$

A fixed point operator

$$((f \star \mathbf{Y}) \star f) = (f \star \mathbf{Y}),$$

is trivial if there is 0: $\mathbf{Y} = 0$.

GAPS can easily express some functional restrictions on the programs. For example the papers [17, 18] are mathematically describing and investigating GAPS for reversive, reversible and completely non-invertible programs are des.

5 Some tools to compose algebraic programs

Algebraic programming is to be a collection of tools to compose and decompose algebraic substructures of GAPS. Some of these tools were outlined for groupoids in the section 3.

The important tool of (de)composition is the construct used in the example 12. Elements of $\mathbb{Z}_3$ can be considered as data and elements of the semigroup as commands. Now we formulate a general case for this construction.

Definition 9. ***Semidirect product*** *of GAPS. Let there are two GAPS: a GAPS of commands C and a GAPS of data D. Let $\varphi : \mathbf{C} \to \mathrm{Hom}(\mathbf{D}, \mathbf{D})$ is a morphism of the semigroup of commands into the semigroup of morphisms of data semigroup. Then semidirect product $C \rtimes D$ is $C \times D$ with the following operations:*

$$\langle c_1, d_1\rangle \circ \langle c_2, d_2\rangle = \langle c_1 \circ c_2, d_1 \circ (d_2\ (c_2\ \varphi))\rangle$$
$$\langle c_1, d_1\rangle \star \langle c_2, d_2\rangle = \langle c_1 \star c_2, d_1 \star (d_2\ (c_2\ \varphi))\rangle\,.$$

This is a generalization of the semidirect product for semigroups.

Now we introduce data types and their connectors taking into account that each data is also an action and that there will be no direct information or control flow from one type to other type.

Definition 10. ***Type systems*** *on GAPS is a system of subGAPSes T_i such that if $i \neq j$, $a \in T_i$, $b \in T_j$ then $(a \star b) = 0$ and $T_i \cap T_j \supseteq \{0\}$.*

Connector *(between T_i and T_J) is an element c not belonging to any type such that if $(a \star c) = b \neq 0$ then there are $i \neq j$ such that $a \in T_i$, $b \in T_j$.*

Dual *T_i' to type T_i is a GAPS for which there is a bijection to T_i $x \leftrightarrow x^p rime$ such that if $(x \star y) = z$ then $(z' \star y^p rime)e = x'$. Dual is* ***perfect*** *if $(x \circ y)' = (y' \circ x'$.*

Mirror *is a connector m such that if $(a \star m) = b$, $a \in T_i$, $b \in T_j$ then $(b' \star m) = a'$. Mirror is* ***perfect*** *if it is connector between T_i and its perfect dual T_i'.*

Reversive type *is a subGAPS R for which there exists the ideal mirror M such that for each $x, y \in R$*

$$((x \star y)' \star (y \star M)) = x' \quad ((x' \star (y \star M))' \star y) = x.$$

Crystal *is a subGAPS where $\star$ forms a group.*

Pipe *is a connector p such that $(x \star p) = y \neq 0 \supset (y \star p) = 0$.*

Result element *of type T_i is a left zero of T_i according to both operations.*

Interruption structure *is a type with a carrier $T \times \{1, 2\}$ where T is GAPS, there are no results in T except maybe 0 and operations are defined as follows:*

$$(\langle a, 1\rangle \star \langle b, x\rangle) = \langle (a \star b), x\rangle\,; \tag{11}$$
$$(\langle a, 2\rangle \star \langle b, y\rangle) = \langle a, 2\rangle \ \text{ if } b \neq 0 \tag{12}$$

Wheel *is an interruption structure based on group $\mathbb{Z}_n$.*

Interruption controlled type *is a system of type T_i, interruption structure S and the pipe p between T_i and $\{1, 2\}$ such that $(x \star p) = 1$ if x is not a result and $(x \star p) = 2$ if x is a result.*

Interruption controlled type can be easily represented through semidirect product but in this representation the pipe (which is necessary for effective program or physical realization) is hidden and the number of elements grows essentially. Each type can be transformed into a type with a given subset of results not disturbing the actions giving other than a result. Using these elements we sometimes can reduce a very complex GAPS to a composition of types, mirrors, pipes, semidirect products and interruption controls. We need no loops and conditional statements here.

An example below shows how to compute a very large power of an element of a given complex algebraic construct through a precomputed representation of power in Fibonacci system. Almost all actions in this program (system) are completely invertible (reversive) if given algebra of data is reversive. Only the initialization of the system and the final interruption are non-invertible.

Example 17. Let us try to apply the same action a large number of times. This corresponds to computing $a \circ b^{\omega}$ in a group. Then ω is represented in Fibonacci system. This can be easily made by usual computer. Let k be the number of bits in the representation of ω. The two predicates are computed and transferred to a reversive program: `(i fib_odd)`, `(i fib_even)`. The first one is 1 iff i is odd and the corresponding digit is equal to 1. `(i fib_even)` is the same for even indexes.

Type loop is resulted from the additive group of integers making integer 0 its output value. Pipe sends its interrupt to externally implemented (maybe physically) group `tp` which is controlled by two boolean commands exchanging commutation of values during compositions, implemented by mirrors and described as conditional operators. These mirrors use two precomputed arrays of booleans to commute inputs of the next operations.

```
PROGRAM Fibonacci_power
DEFINITIONS
int atom n
GROUP tn: external nooutputs
tp atom var a,b,d
tp atom e
(tp,tp) var c is (a,b)
constant e=E
INTERRUPTOR int loop output (0);
loop atom k
int atom var i [0..k] guarded
PIPE(interruption) p: (k,tn); boolean atom l;
predicate [i] fib_odd, fib_even
END DEFINITIONS

INPUT
read a, k
b← a
```

```
i ← 1
l ← TRUE
d ← E
read fib_odd, fib_even
END INPUT

{i;1},
{l; ⊕ true}
par sync (k,tp,i)
{
tp{
   {c; ∘ if l then (e,a) else (b,e) fi};
   {d; ∘ if (i fib_odd) then
       a else if (i fib_odd) then b else e
   fi fi};
}
k{+ (-1)};
i{+ (1)};
}

OUTPUT
write d
END OUTPUT
```

Rough description of implementation of this program is given on fig. 4

6 Conclusion

The main mathematical result of this paper is Theorem 3. It states that many kinds of programs and other computing schemes can be viewed as GAPS and fully states conditions when a (partially described) system of transformations can be correct for the given system. For example we need no descriptions of usual programming languages here because they were described by the λ-calculus which is GAPS.

The last section shows that complex algebras often can be reduced to structures formed with simpler ones in a way like to analog computers and structured programming. Here is a big amount of open problems because (say) a systematic mathematical theory of finite approximations for infinite algebraic systems do not exists now (even for groups).

Maybe the most valuable property of GAPS is that they can be easily adopted to describe functionally restricted classes of programs and computations. Some advanced results in this direction (for reversive, reversible, completely non-invertible programs and dynamic systems) are presented in [17, 18]. It appears now that algebraic programming and computing is the most general existing concept and it is very flexible.

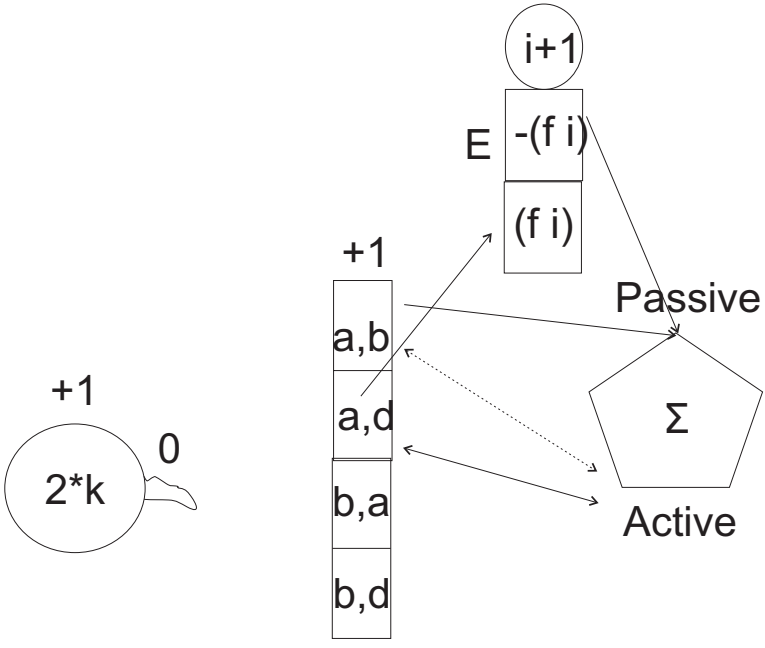


Fig. 4. Program for Fibonacci-based computation

Unfortunately using algebraic models and concepts demands deep knowledge of abstract algebra and category theory and completely another way of thinking then usual Turing-based algorithms and even then Lisp, Refal or Prolog.

Author wishes to thank prof. R. Glück for discussions during which the algebraic concept arose, PSI RAS for support an extremely non-conformist research, his followers A. Nepejvoda and V. Atamanov for their valuable developments in this completely unexplored domain.

References

1. Glushkov, V. M. Abstract Theory of Automata // Cleaver-Hume Press Ltd., 1963.
2. Maurer, W. D. A theory of computer instructions. Journal of the ACM, vol. 13, No 2 (1966) pp. 226–235.
3. Gluschkow W. M., Zeitlin G. E., Justchenko J. L. — Algebra. Sprachen. Programmierung. — Akademie-Verlag, Berlin 1980. — 340 p.
4. Ivanov, P. M. Algebraic modelling of complex systems. — Moscow, 1996. — 274 p.
5. Nepejvoda, N. N. Three-headed Dragon. https://docs.google.com/document /d/1hGzUB3p3j2zYcksoUnxtv7QamHzcgYVYr66iJJiMOhY
6. Alban Ponse and Mark B. van der Zwaag. An Introduction to Program and Thread Algebra. LNCS 3988, 2006, pp 445–488.
7. Bergstra J. A., Bethke I, Ponse A. Program Algebra and Thread Algebra. Amsterdam, 2006, 114p.
8. Nepejvoda N. N. *Reversivity, reversibility and retractability.* Third international Valentin Turchin workshop on metacomputation. Pereslavl: 2012. pp 203–215.
9. Girard J.-Y. *Linear Logic.* Theoretical Computer Science **50**, 1987, 102 pp.
10. Malcev, A.I. Algebraic Systems. Springer-Verlag, 1973, ISBN 0-387-05792-7.
11. Mitchell, J. C. Foundation for programming languages, MIT, 1996.
12. Barenfregt, H. The lambda-calculus. Its syntax and semantics. Elsevier 1984, ISBN 0-444-87508-5.
13. Nepejvoda, N. N. Reversive constructive logics. Logical Investigations, **15**, 150–169 (2008).
14. Axelsen, H. G., Glück, R. What do reversible programs compute? FOCSSACS 2011, LNCS 6604, pp. 42–56, 2011
15. http://www.muppetlabs.com/ breadbox/bf/
16. Böhm, C., Dezani-Ciancaglini, M. Can syntax be ignored during translation? In: Nivat M. (ed.) Automata, languages and programming. North-Holland, Amsterdam, 1972. p.197–207.
17. Nepejvoda, N. N. Abstract algebras of different classes of programs. Proceedings of the 3rd international conference on applicative computation systems (ACS'2012), 103–128.
18. Nepejvoda, N. N. Algebraic approach to control. Control Sciences, 2013, N 6, 2-14.

Author Index

Научное издание

Труды конференции

Сборник трудов Четвертого международного семинара по метавычислениям имени В. Ф. Турчина, г. Переславль-Залесский, 29 июня – 3 июля 2014 г.

Под редакцией А. В. Климова и С. А. Романенко.

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. **14,5**. Усл. печ. л. **14,9**. Подписано к печати **10.06.2014**. Ответственный за выпуск: *С. М. Абрамов.*

Издательство Университета города Переславля

Отпечатано в ООО «Регион». Печать цифровая. Бумага офсетная. Тираж 150 экз. Заказ 3.
152025 Ярославская область, г. Переславль-Залесский, ул. Строителей, д. 41.