Russian Academy of Sciences
Ailamazyan Program Systems Institute

Fifth International Valentin Turchin Workshop on Metacomputation

Proceedings
Pereslavl-Zalessky, Russia, June 27 – July 1, 2016

Издательство Университета города Переславля
Pereslavl-Zalessky

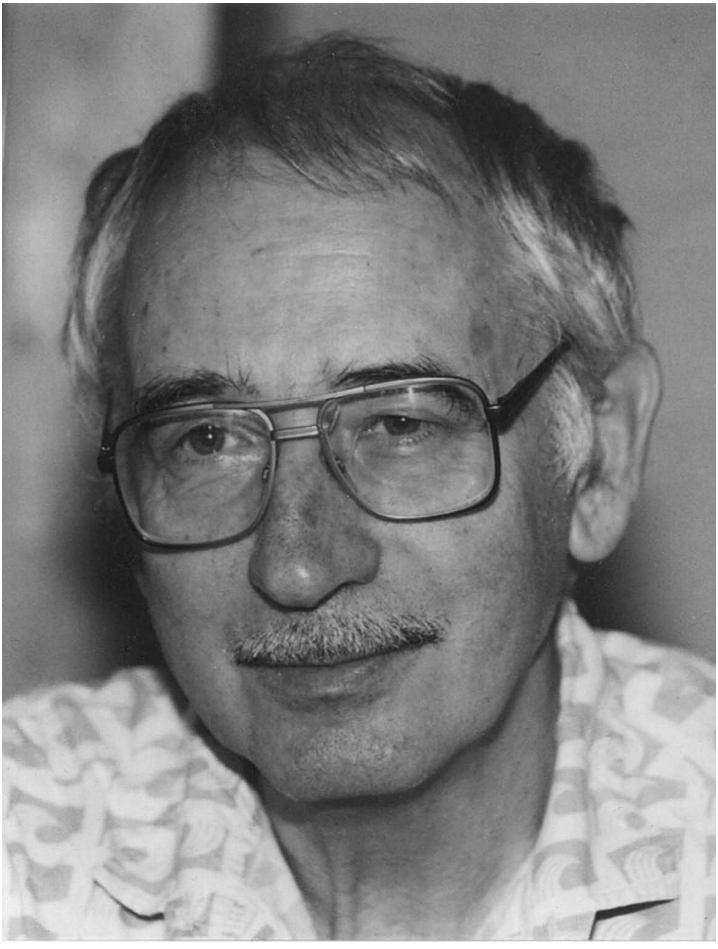
УДК 004.42(063)
ББК 22.18

П-99

Fifth International Valentin Turchin Workshop on Metacomputation // Proceedings of the Fifth International Valentin Turchin Workshop on Metacomputation. Pereslavl-Zalessky, Russia, June 27 – July 1, 2016 / *Edited by A. V. Klimov and S. A. Romanenko.* — Pereslavl Zalessky: Publishing House "University of Pereslavl", 2016, **184** p. — ISBN 978-5-901795-33-0

Пятый международный семинар по метавычислениям имени В. Ф. Турчина // Сборник трудов Пятого международного семинара по метавычислениям имени В. Ф. Турчина, г. Переславль-Залесский, 27 июня – 1 июля 2016 г. / *Под редакцией А. В. Климова и С. А. Романенко.* — Переславль-Залесский: Издательство «Университет города Переславля», 2016, **184** с. — ISBN 978-5-901795-33-0

ISBN 978-5-901795-33-0



Valentin Turchin
(1931–2010)

Preface

The Fifth International Valentin Turchin Workshop on Metacomputation, META 2016, was held on June 27 – July 1, 2016 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

- Robert Glück, Andrei Klimov: *Introduction to Valentin Turchin's Cybernetic Foundation of Mathematics*

Invited Speaker Neil D. Jones

- Neil D. Jones: *On Programming and Biomolecular Computation*
- Daniil Berezun and Neil D. Jones: *Working Notes: Compiling ULC to Lower-level Code by Game Semantics and Partial Evaluation*

Partial evalatuation

- Robert Glück: *Preliminary Report on Polynomial-Time Program Staging by Partial Evaluation*

Program slicing

- Husni Khanfar and Björn Lisper: *Enhanced PCB-Based Slicing*

Distillation

- Venkatesh Kannan and G. W. Hamilton: *Distilling New Data Types*

Program Verification

– Andrew Mironov: *On a Method of Verification of Functional Programs*

Proving Program Equivalence

– Sergei Grechanik: *Towards Unification of Supercompilation and Equality Saturation*
– Dimitur Krustev: *Simple Programs on Binary Trees – Testing and Decidable Equivalence*

Supercompilation

– Dimitur Krustev: *A Supercompiler Assisting Its Own Formal Verification*
– Robert Glück, Andrei Klimov, and Antonina Nepeivoda: *Non-Linear Configurations for Superlinear Speedup by Supercompilation*
– Antonina Nepeivoda: *Complexity of Turchin's Relation for Call-by-Name Computations*

Derivation of parallel programs

– Arkady Klimov: *Derivation of Parallel Programs of Recursive Doubling Type by Supercompilation with Neighborhood Embedding*

Algorithms

– Nikolay Shilov: *Algorithm Design Patterns: Programming Theory Perspective*

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/
– META 2016: http://meta2016.pereslavl.ru/

May 2016

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

Robert Glück, University of Copenhagen, Denmark
Geoff Hamilton, Dublin City University, Republic of Ireland
Dimitur Krustev, IGE+XAO Balkan, Bulgaria
Alexei Lisitsa, Liverpool University, United Kingdom
Neil Mitchell, Standard Charted, United Kingdom
Antonina Nepeivoda, Ailamazyan Program Systems Institute of RAS, Russia
Peter Sestoft, IT University of Copenhagen, Denmark
Artem Shvorin, Ailamazyan Program Systems Institute of RAS, Russia
Morten Sørensen, Formalit, Denmark

Invited Speaker

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

Sponsoring Organizations

Federal Agency for Scientific Organizations (FASO Russia)

Russian Academy of Sciences

Russian Foundation for Basic Research (grant № 16-07-20527-г)

Table of Contents

Working Notes: Compiling ULC to Lower-level Code by Game Semantics and Partial Evaluation

Daniil Berezun[1] and Neil D. Jones[2]

[1] JetBrains and St. Petersburg State University (Russia)
[2] DIKU, University of Copenhagen (Denmark)

Abstract. What: Any expression M in ULC (the untyped λ-calculus) can be compiled into a rather low-level language we call LLL, whose programs contain *none of the traditional implementation devices for functional languages:* environments, thunks, closures, etc. A compiled program is first-order functional and has a fixed set of working variables, whose number is independent of M. The generated LLL code in effect *traverses* the subexpressions of M.
How: We apply the techniques of *game semantics* to the untyped λ-calculus, but take a more operational viewpoint that uses much less mathematical machinery than traditional presentations of game semantics. Further, the untyped lambda calculus ULC is compiled into LLL by *partially evaluating* a traversal algorithm for ULC.

1 Context and contribution

Plotkin posed the problem of existence of a fully abstract semantics of PCF [17]. Game semantics provided the first solution [1–3, 9]. Subsequent papers devise fully abstract game semantics for a wide and interesting spectrum of programming languages, and further develop the field in several directions.

A surprising consequence: it is possible to build a lambda calculus interpreter with **none** of the traditional implementation machinery: β-reduction; environments binding variables to values; and "closures" and "thunks" for function calls and parameters. This new viewpoint on game semantics looks promising to see its operational consequences. Further, it may give a new line of attack on an old topic: *semantics-directed compiler generation* [10, 18].

Basis: Our starting point was Ong's approach to normalisation of the simply typed λ-calculus(henceforth called STLC). Paper [16] adapts the game semantics framework to yield an STLC normalisation procedure (STNP for short) and its correctness proof using the traversal concept from [14, 15].

STNP can be seen as in *interpreter*; it evaluates a given λ-expression M by managing a list of subexpressions of M, some with a single back pointer. These notes extend the normalisation-by-traversals approach to the *untyped* λ-calculus, giving a new algorithm called UNP, for Untyped Normalisation Procedure. UNP correctly evaluates any STLC expression sans types, so it properly extends STNP since ULC is Turing-complete while STLC is not.

Plan: in these notes we start by describing a weak normalisation procedure. A traversal-based algorithm is developed in a systematic, semantics-directed way. Next step: extend this to full normalisation and its correctness proof (details omitted from these notes). Finally, we explain briefly how partial evaluation can be used to implement ULC, compiling it to a low-level language.

2 Normalisation by traversal: an example

Perhaps surprisingly, the normal form of an STLC λ-expression M may be found by simply taking a walk over the subexpressions of M. As seen in [14–16] there is no need for β-reduction, nor for traditional implementation techniques such as environments, thunks, closures, etc. The "walk" is a *traversal*: a sequential visit to subexpressions of M. (Some may be visited more than once, and some not at all.)

A classical example; multiplication of Church numerals[3]

$$\texttt{mul} = \lambda m.\lambda n.\lambda s.\lambda z.m(ns)z$$

A difference between reduction strategies: consider computing $3*2$ by evaluating $\texttt{mul}\ \underline{3}\ \underline{2}$. Weak normalisation reduces $\texttt{mul}\ \underline{3}\ \underline{2}$ only to $\lambda s.\lambda z.\underline{3}(\underline{2}s)z$ but does no computation under the lambda. On the other hand strong normalisation reduces it further to $\lambda s.\lambda z.s^6 z$.

A variant is to use free variables S, Z instead of the bound variables s, z, and to use $\texttt{mul}' = \lambda m.\lambda n.m(nS)Z$ instead of $\texttt{mul}$. Weak normalisation computes all the way: $\texttt{mul}'\ \underline{3}\ \underline{2}$ weakly reduces to $S^6 Z$ as desired. More generally, the Church-Turing thesis holds: a function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is partial recursive (computable) iff there is a λ-expression M such that for any $x_1, \ldots, x_n, x \in \mathbb{N}$

$$f(x_1, \ldots, x_n) = x \ \Leftrightarrow\ M(S^{x_1} Z) \ldots (S^{x_n} Z) \text{ weakly reduces to } \ S^x Z$$

The unique traversal of $\underline{3}\,(\underline{2}\,S)\,Z$ visits subexpressions of Church numeral $\underline{3}$ once. However it visits $\underline{2}$ twice, since in general $x * y$ is computed by adding y together x times. The weak normal form of $\underline{3}\,(\underline{2}\,S)\,Z$ is $S^6 Z$: the core of Church numeral $\underline{6}$.

Figure 1 shows traversal of expression $\underline{2}\,(\underline{2}\,S)\,Z$ in tree form[4]. The labels **1:, 2:** etc. are not part of the λ-expression; rather, they indicate the order in which subexpressions are traversed.

[3] The Church numeral of natural number $\boldsymbol{x}$ is $\underline{\boldsymbol{x}} = \boldsymbol{\lambda s.\lambda z.s^x z}$. Here $\boldsymbol{s^x} = \boldsymbol{s(s(\ldots s(z)\ldots))}$ with $\boldsymbol{x}$ occurrences of $\boldsymbol{s}$, where $\boldsymbol{s}$ represents "successor" and $\boldsymbol{z}$ represents "zero".

[4] Application operators $@_i$ have been made exlicit, and indexed for ease of reference. The two $\underline{2}$ subtrees are the "data'; their bound variables have been named apart to avoid confusion. The figure's "program" is the top part $\_\,(\_\,S)\,Z$.

Computation by traversal can be seen as a game

The traversal in Figure 1 is a game play between program $\lambda m \lambda n.(m@\,(n@S))@Z$ and the two data values (m, n each have λ-expression $\underline{2}$ as value).

Informally: two program nodes are visited in steps 1, 2; then data 1's leftmost branch is traversed from node $\lambda s1$ until variable $s1$ is encountered at step 6. Steps 7-12: data 2's leftmost branch is traversed from node $\lambda s2$ down to variable $s2$, after which the program node S is visited (intuitively, the first output is produced). Steps 13-15: the data 2 nodes $@_6$ and $s2$ are visited, and the program: it produces the second output S. Steps 16, 17: $z2$ is visited, control is finished (for now) in data 2, and control resumes in data 1.

Moving faster now: $@_4$ and the second $s1$ are visited; data 2 is scanned for a second time; and the next two output S's are produced. Control finally returns to $z1$. After this, in step 30 the program produces the final output Z.

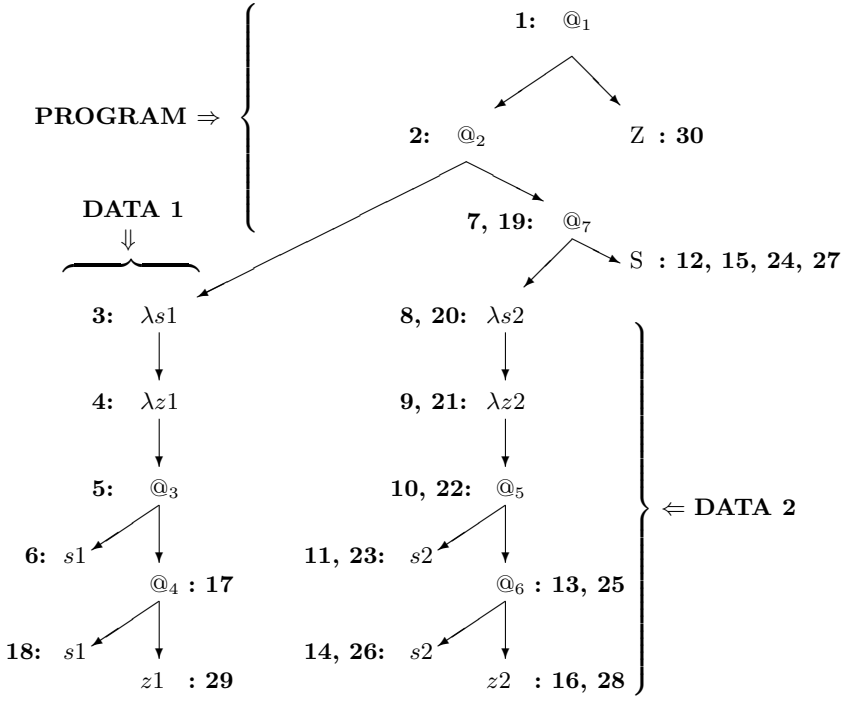


Fig. 1. Syntax tree for `mul` $\underline{2}\,\underline{2}\,S\,Z = \underline{2}(\underline{2}S)Z$. (Labels show traversal order.)

Which traversal? As yet this is only an "argument by example"; we have not yet explained *how to choose* among all possible walks through the nodes of $\underline{2}(\underline{2}S)Z$ to find the correct normal form.

3 Overview of three normalisation procedures

3.1 The STNP algorithm

The STNP algorithm in [16] is deterministic, defined by syntax-directed inference rules.

The algorithm is type-oriented even though the rules do not mention types: it requires as first step the conversion from STLC to η-long form. Further, the statement of correctness involves types in "term-in-context" judgements $\Gamma \vdash M : A$ where A is a type and Γ is a type environment.

The correctness proof involves types quite significantly, to construct program-dependent arenas, and as well a category whose objects are arenas and whose morphisms are innocent strategies over arenas.

3.2 Call-by-name weak normalisation

We develop a completely type-free normalisation procedure for ULC.[5] Semantics-based stepping stones: we start with an environment-based semantics that resembles traditional implementations of CBN (call-by-name) functional languages. The approach differs from and is simpler than [13].

The second step is to enrich this by adding a "history" argument to the evaluation function. This records the "traversal until now". The third step is to simplify the environment, replacing recursively-defined bindings by bindings from variables to positions in the history. The final step is to remove the environments altogether.

The result is a closure- and environment-free traversal-based semantics for weak normalisation.

3.3 The UNP algorithm

UNP yields full normal forms by "reduction under the lambda" using *head linear reduction* [7, 8]. The full UNP algorithm [4] currently exists in two forms:

- An implementation in HASKELL; and
- A set of term rewriting rules. A formal correctness proof of UNP is nearing completion, using the theorem prover COQ [4].

[5] Remark: evaluator nontermination is allowed on an expression with no weak normal form.

4 Weak CBN evaluation by traversals

We begin with a traditional environment-based call-by-name semantics: a *reduction-free* common basis for implementing a functional language. We then eliminate the environments by three transformation steps. The net effect is to replace the environments by traversals.

4.1 Weak evaluation using environments

The object of concern is a pair $e:\rho$, where e is a λ-expression and ρ is an environment[6] that binds some of e's free variables to pairs $e'{:}\rho'$.

Evaluation judgements, metavariables and domains:

$e{:}\rho \Downarrow v$ Expression e in environment ρ evaluates to value v
$v, e{:}\rho \Downarrow v'$ Value v applied to argument e in environment ρ gives value v'
$\rho \in Env = Var \rightharpoonup Exp \times Env$
$v \in Value = \{e : \rho \mid e \text{ does not have form } (\lambda x.e_0)e_1\}$

Note: the environment domain Env is defined recursively.

Determinism The main goal, given a λ-expression M, is to find a value v such that $M{:}[] \Downarrow v$ (if it exists). The following rules may be thought of as an *algorithm* to evaluate M since they are are deterministic. Determinism follows since the rules are *single-threaded*: consider a goal *left* $\Downarrow$ *right*. If *left* has been computed but *right* is still unknown, then at most one inference rule can be applied, so the final result value is uniquely defined (if it exists).

$$\text{(Lam)}\ \frac{}{\lambda x.e{:}\rho \Downarrow \lambda x.e{:}\rho} \qquad \text{(Freevar)}\ \frac{x \text{ free in } M}{x{:}\rho \Downarrow x{:}[]} \qquad \text{(Boundvar)}\ \frac{\rho(x) \Downarrow v}{x{:}\rho \Downarrow v}$$

Abstractions and free variables evaluate to themselves. A bound variable x is accessed using call-by-name: the environment contains an unevaluated expression, which is evaluated when variable x is referenced.

$$\text{(AP)}\ \frac{e_1{:}\rho \Downarrow v_1 \qquad v_1, e_2{:}\rho \ \Downarrow\ v}{e_1@e_2{:}\rho \ \Downarrow\ v}$$

Rule (AP) first evaluates the operator e_1 in an application $e_1@e_2$. The value v_1 of operator e_1 determines whether rule (AP_λ) or ($\text{AP}_{\overline{\lambda}}$) is applied next.

$$\text{(AP}_\lambda\text{)}\ \frac{e' = \lambda x.e'' \qquad \rho'' = \rho'[x \mapsto e_2{:}\rho] \qquad e''{:}\rho'' \Downarrow v}{e'{:}\rho', e_2{:}\rho \ \Downarrow\ v}$$

[6] Call-by-name semantics: If ρ contains a binding $x \mapsto e' : \rho'$, then e' is an as-yet-unevaluated expression and ρ' is the environment that was current at the time when x was bound to e'.

In rule (AP_λ) if the operator value is an abstraction $\lambda x.e'':\rho'$ then ρ' is extended by binding x to the as-yet-unevaluated operand e_2 (paired with its current environment ρ). The body e'' is then evaluated in the extended ρ' environment.

$$(\mathrm{AP}_{\overline{\lambda}}) \quad \frac{e' \neq \lambda x.e'' \qquad e_2{:}\rho \Downarrow e_2'{:}\rho_2' \qquad \mathrm{fv}(e') \cap \mathrm{dom}(\rho_2') = \emptyset}{e'{:}\rho', e_2{:}\rho \;\Downarrow\; (e' @ e_2'){:}\rho_2'}$$

In rule ($\mathrm{AP}_{\overline{\lambda}}$) the operator value $e':\rho'$ is a non-abstraction. The operand e_2 is evaluated. The resulting value is an application of operator value e' to operand value (as long as no free variables are captured).

Rule ($\mathrm{AP}_{\overline{\lambda}}$) yields a value containing an application. For the multiplication example, Rule ($\mathrm{AP}_{\overline{\lambda}}$) yields all of the S applications in result $S@(S@(S@(S@Z)))$.

4.2 Environment semantics with traversal history h

Determinism implies that there exists at most one sequence of visited subexpressions for any $e{:}\rho$. We now extend the previous semantics to accumulate the history of the evaluation steps used to evaluate $e{:}\rho$.

These rules accumulate a list $h = [e_1{:}\rho_1, \ldots, , \ldots, e_n{:}\rho_n]$ of all subexpressions of λ-expression M that have been visited, together with their environments. Call such a partially completed traversal a *history*. A notation:

$$[e_1{:}\rho_1, \ldots, e_n{:}\rho_n] \bullet e{:}\rho = [e_1{:}\rho_1, \ldots, e_n{:}\rho_n, e{:}\rho]$$

Metavariables, domains and evaluation judgements.

$e{:}\rho, h \Downarrow v$ Expression e in environment ρ evaluates to value v
$v, e{:}\rho, h \Downarrow v'$ Value v applied to expression e in environment ρ gives value v'
$\rho \in Env = Var \rightharpoonup Exp \times Env$
$v \in Value = \{e : \rho, h \mid h \in History, e \neq (\lambda x.e_0)e_1\}$
$h \in History = (Exp \times Env)^*$

Judgements now have the form $e : \rho, h \;\Downarrow\; e' : \rho', h'$ where h is the history *before* evaluating e, and h' is the history *after* evaluating e. Correspondingly, we redefine a value to be of form $v = e{:}\rho, h$ where e is not a β-redex.

$$(\mathrm{Lam}) \frac{}{\lambda x.e{:}\rho, h \Downarrow \lambda x.e{:}\rho,\; h{\bullet}(\lambda x.e{:}\rho)}$$

$$(\mathrm{Freevar}) \quad \frac{x \text{ free in } M}{x{:}\rho, h \;\Downarrow\; x{:}[],\; h{\bullet}(x{:}[])} \qquad (\mathrm{Boundvar}) \quad \frac{\rho(x),\; h{\bullet}(x{:}\rho) \;\Downarrow\; v}{x{:}\rho, h \;\Downarrow\; v}$$

$$(\mathrm{AP}) \quad \frac{e_1{:}\rho,\; h{\bullet}(e_1@e_2, \rho) \Downarrow v_1 \qquad v_1, e_2{:}\rho \;\Downarrow\; v}{e_1@e_2{:}\rho, h \;\Downarrow\; v}$$

$$(\mathrm{AP}_\lambda)\quad \frac{e' = \lambda x.e'' \qquad \rho'' = \rho'[x \mapsto e_2{:}\rho] \qquad e''{:}\rho'', h' \;\Downarrow\; v}{e'{:}\rho', h',\ e_2{:}\rho \;\Downarrow\; v}$$

$$(\mathrm{AP}_{\overline{\lambda}})\quad \frac{e' \neq \lambda x.e'' \qquad e_2{:}\rho, h' \Downarrow e_2'{:}\rho_2', h_2' \qquad \mathrm{fv}(e') \cap \mathrm{dom}(\rho_2') = \emptyset}{e'{:}\rho', h',\ e_2{:}\rho \Downarrow (e'@e_2'){:}\rho_2', h_2'}$$

Histories are accumulative It is easy to verify that h is a prefix of h' whenever $e{:}\rho, h \Downarrow e'{;}\rho', h'$.

4.3 Making environments nonrecursive

The presence of the history makes it possible to bind a variable x not to a pair $e{:}\rho$, but instead to the position of a prefix of history h. Thus $\rho \in Env = Var \to \mathbb{N}$. Domains and evaluation judgement:

$e{:}\rho, h \Downarrow v$ Expression e in environment ρ evaluates to value v
$v, e{:}\rho, h \Downarrow v'$ Value v applied to argument e in environment ρ gives value v'
$\rho \in Env = Var \rightharpoonup \mathbb{N}$
$h \in History = (Exp \times Env)^*$

A major difference: environments are now "flat" (nonrecursive) since Env is no longer defined recursively. Nonetheless, environment access is still possible, since at all times the current history includes all previously traversed expressions.

Only small changes are needed, to rules (AP_λ) and (Boundvar); the remaining are identical to the previous version and so not repeated.

$$(\mathrm{AP}_\lambda)\quad \frac{e' = \lambda x.e'' \qquad \rho'' = \rho'[x \mapsto |h|] \qquad e''{:}\rho'', h' \;\Downarrow\; v}{e'{:}\rho', h',\ e_1@e_2{:}\rho, h \;\Downarrow\; v}$$

$$(\mathrm{Boundvar})\quad \frac{nth\ \rho(x)\ h = e_1@e_2{:}\rho' \qquad e_2{:}\rho' \Downarrow v}{x{:}\rho, h \;\Downarrow\; v}$$

In rule (AP_λ), variable x is bound to length $|h|$ of the history h that was current for $(\mathtt{e}_1@\mathtt{e}_2, \rho)$. As a consequence bound variable access had to be changed to match. The indexing function $nth : \mathbb{N} \to History \to Exp \times Env$ is defined by: $nth\ i\ [e_1{:}\rho_1, \ldots, e_n{:}\rho_n] = e_i{:}\rho_i$.

4.4 Weak UNP: back pointers and no environments

This version is a semantics completely free of environments: it manipulates only traversals and back pointers to them. How it works: it replaces an environment by two back pointers, and finds the value of a variable by looking it up in the history, following the back pointers.

Details will be appear in a later version of this paper

5 The low-level residual language LLL

The semantics of Section 4.4 manipulates first-order values. We abstract these into a tiny first-order functional language called LLL: essentially a machine language with a heap and recursion, equivalent in power and expressiveness to the language F in book [11].

Program variables have simple types (not in any way depending on M). A token, or a product type, has a static structure, fixed for any one LLL program. A list type `[tau]` denotes dynamically constructed values, with constructors `[] and :`. Deconstruction is done by `case`. Types are as follows, where Token denotes an atomic symbol (from a fixed alphabet).

```
tau ::=  Token  |  (tau, tau)  |  [ tau ]
```

Syntax of LLL

```
program ::=  f1 x = e1  ...  fn x = en

e  ::=  x     |  f e
    |   token | case e of token1 -> e1 ... tokenn -> en
    |   (e,e) | case e of (x,y) -> e
    |    []   | case e of   [] -> e    x:y -> e

x, y    ::= variables
token   ::= an atomic symbol (from a fixed alphabet)
```

6 Interpreters, compilers, compiler generation

Partial evaluation (see [12]) can be used to specialise a normalisation algorithm to the expression being normalised. The net effect is to compile an ULC expression into an LLL equivalent that contains no ULC-syntax; the target programs are first-order recursive functional program with "cons". Functions have only a fixed number of arguments, independent of the input λ-expression M.

6.1 Partial evaluation (= program specialisation)

One goal of this research is to partially evaluate a normaliser with respect to "static" input M. An effect can be to compile ULC into a lower-level language.

Partial evaluation, briefly A partial evaluator is a *program specialiser*, called *spec*. Its defining property:

$$\forall p \in \mathit{Programs}\ .\ \forall s, d \in \mathit{Data}\ .\ [\![[\![\mathit{spec}]\!](p, s)]\!](d) = [\![p]\!](s, d)$$

The net effect is a *staging transformation*: $[\![p]\!](s, d)$ is a 1-stage computation; but $[\![[\![\mathit{spec}]\!](p, s)]\!](d)$ is a 2-stage computation.

Program speedup is obtained by *precomputation*. Given a program p and "static" input value s, *spec* builds a *residual program* $p_s \stackrel{def}{=} [\![spec]\!](p, s)$. When run on any remaining "dynamic" data d, residual program p_s computes what p would have computed on both data inputs s, d.

The concept is historically well-known in recursive function theory, as the S-1-1 theorem. In recent years partial evaluation has emerged as the practice of engineering the S-1-1 theorem on real programs [12]. One application is compiling. Further, self-application of *spec* can achieve compiler generation (from an interpreter), and even compiler generator generation (details in [12]).

6.2 Should normalisation be staged?

In the current λ-calculus tradition M is self-contained; there is no dynamic data. So why would one wish to break normalisation into 2 stages?

Some motivations for staging The specialisation definition looks almost trivial on a normaliser program NP:

$$\forall M \in \Lambda \,.\, [\![\, [\![spec]\!](\mathrm{NP}, M)]\!]() = [\![\mathrm{NP}]\!](M)$$

An extension: allow M to have separate input data, e.g., the input value $\underline{2}$ as in the example of Section 2. Assume that NP is extended to allow run-time input data.[7] The specialisation definition becomes:

$$\forall M \in \Lambda, d \in Data \,.\, [\![\, [\![spec]\!](\mathrm{NP}, M)]\!](d) = [\![\mathrm{NP}]\!](M, d) =_\beta M@d$$

Is staged normalisation a good idea? Let $\mathrm{NP}_M = [\![spec]\!](\mathrm{NP}, M)$ be the specialiser output.

1. One motivation is that NP_M can be in a much simpler language than the λ-calculus. Our candidate: the "low-level language" LLL of Section 5.
2. A well-known fact: the traversal of M may be much larger than M. By Statman's results [19] it may be larger by a "non-elementary" amount (!). Nonetheless it is possible to construct a λ-free residual program NP_M with $|NP_M| = O(|M|)$, i.e., such that M's LLL equivalent has *size that is only linearly larger* than M itself. More on this in Section 6.3.
3. A next step: consider *computational complexity* of normalising M, if it is applied to an external input d. For example the Church numeral multiplication algorithm runs in time of the order of the product of the sizes of its two inputs.
4. Further, two stages are natural for semantics-directed compiler generation.

[7] Semantics: simply apply M to Church numeral d before normalisation begins.

How to do staging Ideally the partial evaluator can do, at specialisation time, all of the NP operations that depend only on M. As a consequence, NP_M will have *no operations at all* to decompose or build lambda expressions while it runs on data d. The "residual code" in NP_M will contain only operations to extend the current traversal, and operations to test token values and to follow the back pointers.

Subexpressions of M may appear in the low-level code, but are only used as indivisible tokens. They are only used for equality comparisons with other tokens, and so could be replaced by numeric codes – tags to be set and tested.

6.3 How to specialise NP with respect to M ?

The first step is to *annotate* parts of (the program for) NP as either static or dynamic. Computations in NP will be either *unfolded* (i.e., done at partial evaluation time) or *residualised*: Runtime code is generated to do computation in the output program $\text{NP}_{\texttt{mul}}$ (this is p_s as seen in the definition of a specialiser).

Static: Variables ranging over *syntactic objects* are annotated as static. Examples include the λ-expressions that are subexpressions of M. Since there are only finitely many of these for any fixed input M, it is safe to classify such syntactic data as static.

Dynamic: Back pointers are dynamic; so the traversal being built must be dynamic too. One must classify data relevant to traversals or histories as dynamic, since there are unboundedly many.[8]

For specialisation, all function calls of the traversal algorithm to itself that do not progress from one M subexpression to a proper subexpression are annotated as "dynamic". The motivation is increased efficiency: no such recursive calls in the traversal-builder will be unfolded while producing the generator; but *all other calls* will be unfolded.

About the size of the compiled λ-expression (as discussed in Section 6.2).

We assume that NP is *semi-compositional*: static arguments e_i in a function call $f(e_1, \ldots, e_n)$ must either be absolutely bounded, or be substructures of M (and thus λ-expressions).

The size of NP_M will be linear in $|M|$ if for any NP function $f(x_1, \ldots, x_n)$, each static argument is either completely bounded or of BSV; and there is at most one BSV argument, and it is always a subexpression of M.

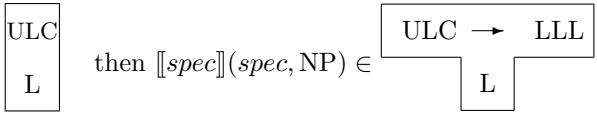
[8] In some cases more can be made static: "the trick" can be used to make static copies of dynamic values that are of BSV, i.e., of *bounded static variation*, see discussion in [12].

6.4 Transforming a normaliser into a compiler

Partial evaluation can transform the ULC (or STNP) normalisation algorithm NP into a program to compute a semantics-preserving function

$$f : \text{ULC} \to \text{LLL} \text{ (or f : STLC} \to \text{LLL)}$$

This follows from the second Futamura projection. In diagram notation of [12]:



Here L is the language in which the partial evaluator and normaliser are written, and LLL of Section 5 is a sublanguage large enough to contain all of the dynamic operations performed by NP.

Extending this line of thought, one can anticipate its use for a *semantics-directed compiler generator*, an aim expressed in [10]. The idea would be to use LLL as a general-purpose intermediate language to express semantics.

6.5 Loops from out of nowhere

Consider again the Church numeral multiplication (as in Figure 1), but with a difference: suppose the data input values for m, n are given separately, at the time when program $\text{NP}_{\texttt{mul}}$ is run.. Expectations:

- Neither `mul` nor the data contain any loops or recursion. However `mul` will be compiled into an *LLL -program* $\text{NP}_{\texttt{mul}}$ *with two nested loops.*
- Applied to two Church numerals m, n, $\text{NP}_{\texttt{mul}}$ computes their product by doing one pass over the Church numeral for m, interleaved with m passes over the Church numeral for n. (One might expect this intuitively).
- These appear as an artifact of the specialisation process. The reason the loops appear: While constructing $\text{NP}_{\texttt{mul}}$ (i.e., during specialisation of NP to its static input `mul`), the specialiser will encounter the same static values (subexpressions of M) more than once.

7 Current status of the research

Work on the simply-typed λ-calculus

We implemented a version of STNP in HASKELL and another in SCHEME. We plan to use the UNMIX partial evaluator (Sergei Romanenko) to do automatic partial evaluation and compiler generation. The HASKELL version is more complete, including: typing; conversion to eta-long form; the traversal algorithm itself; and construction of the normalised term.

We have handwritten STNP-gen in SCHEME. This is the *generating extension* of STNP. Effect: compile from UNC into LLL, so $NP_M = [\![\text{STNP-gen}]\!](M)$. Program STNP-gen is essentially the compiler generated from STNP that could be obtained as in Section 6.4 Currently STNP-gen yields output NP_M as a `scheme` program, one that would be easy to convert into LLL as in Section 5.

Work on the untyped λ-calculus

UNP is a normaliser for UNC. A single traversal item may have two back pointers (in comparison: STNP uses one). UNP is defined semi-compositionally by recursion on syntax of ULC-expression M. UNP has been written in HASKELL and works on a variety of examples. A more abstract definition of UNP is on the way, extending Section 4.4.

By specialising UNP, an arbitrary untyped ULC-expression can be translated to LLL . A correctness proof of UNP is pending. No SCHEME version or generating extension has yet been done, though this looks worthwhile for experiments using UNMIX.

Next steps

More needs to be done towards separating programs from data in ULC (Section 6.5 was just a sketch). A current line is to express such program-data games in a *communicating* version of LLL . Traditional methods for compiling *remote function calls* are probably relevant.

It seems worthwhile to investigate *computational complexity* (e.g., of the λ-calculus); and as well, the *data-flow analysis* of output programs (e.g., for program optimisation in time and space).

Another direction is to study the utility of LLL as an intermediate language for a semantics-directed compiler generator.

References

1. S. Abramsky and G. McCusker. Game semantics. In *Computational Logic: Proceedings of the 1997 Marktoberdorf Summer School*, pages 1–56. Springer Verlag, 1999.
2. Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. In *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, pages 1–15, 1994.
3. Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.
4. Daniil Berezun. UNP: normalisation of the untyped λ-expression by linear head reduction. *ongoing work*, 2016.
5. William Blum and C.-H. Luke Ong. The safe lambda calculus. *Logic Methods in Computer Science*, 5(1), 2009.
6. William Blum and Luke Ong. A concrete presentation of game semantics. In *Galop 2008:Games for Logic and Programming Languages*, 2008.

7. Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of girard's execution formula). In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993*, pages 296–306, 1993.
8. Vincent Danos and Laurent Regnier. Head linear reduction. *unpublished*, 2004.
9. J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.
10. Neil D. Jones, editor. *Semantics-Directed Compiler Generation, Proceedings of a Workshop, Aarhus, Denmark, January 14-18, 1980*, volume 94 of *Lecture Notes in Computer Science*. Springer, 1980.
11. Neil D. Jones. *Computability and complexity - from a programming perspective*. Foundations of computing series. MIT Press, 1997.
12. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice Hall international series in computer science. Prentice Hall, 1993.
13. Andrew D. Ker, Hanno Nickau, and C.-H. Luke Ong. Innocent game models of untyped lambda-calculus. *Theor. Comput. Sci.*, 272(1-2):247–292, 2002.
14. Robin P. Neatherway, Steven J. Ramsay, and C.-H. Luke Ong. A traversal-based algorithm for higher-order model checking. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 353–364, 2012.
15. C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 81–90, 2006.
16. C.-H. Luke Ong. Normalisation by traversals. *CoRR*, abs/1511.02629, 2015.
17. Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.
18. David A. Schmidt. State transition machines for lambda calculus expressions. In Jones [10], pages 415–440.
19. Richard Statman. The typed lambda-calculus is not elementary recursive. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA*, pages 90–94, 1977.

Preliminary Report on Polynomial-Time Program Staging by Partial Evaluation

Robert Glück

DIKU, Dept. of Computer Science, University of Copenhagen, Denmark

Summary

Maximally-polyvariant partial evaluation is a strategy for program specialization that propagates static values as accurate as possible [4]. The increased accuracy allows a maximally-polyvariant partial evaluator to achieve, among others, the Bulyonkov effect [3], that is, constant folding while specializing an interpreter.

The polyvariant handling of return values avoids the monovariant return approximation of conventional partial evaluators [14], in which the result of a call is dynamic if one of its arguments is dynamic. But multiple return values are the "complexity generators" of maximally polyvariant partial evaluation because multiple continuations need to be explored after a call, and this degree of branching is not bound by a program-dependent constant, but depends on the initial static values. In an offline partial evaluator, a recursive call has at most one return value that is either static or dynamic.

A conventional realization of a maximally-polyvariant partial evaluator can take exponential time for specializing programs. The online partial evaluator [10] achieves the same precision in time polynomial in the number of partial-evaluation configurations. This is a significant improvement because no fast algorithm was known for maximally-polyvariant specialization. The solution involves applying a polynomial-time simulation of nondeterministic pushdown automata [11].

For an important class of quasi-deterministic specialization problems the partial evaluator takes linear time, which includes Futamura's challenge [8]: (1) the linear-time specialization of a naive string matcher into (2) a linear-time matcher. This is remarkable because both parts of Futamura's challenge are solved. The second part was solved in different ways by several partial evaluators, including generalized partial computation by employing a theorem prover [7], perfect supercompilation based on unification-based driving [13], and offline partial evaluation after binding-time improvement of a naive matcher [5]. The first part remained unsolved until this study, though it had been pointed out [1] that manual binding-time improvement of a naive matcher could expose static functions to the caching of a hypothetical memoizing partial evaluator.

Previously, it was unknown that the *KMP test* [15] could be passed by a partial evaluator without sophisticated binding-time improvements. Known solutions to the KMP test include Futamura's generalized partial computation utilizing a theorem prover [8], Turchin's supercompilation with unification-based driving [13], and various binding-time-improved matchers [1, 5].

As a result, a class of specialization problems can now be solved faster than before with high precision, which may enable faster Ershov's generating extensions [6,9,12], *e.g.*, for a class similar to Bulyonkov's analyzer programs [2]. This is significant because *super-linear program staging* by partial evaluation becomes possible: the time to run the partial evaluator *and* its residual program is linear in the input, while the original program is not, as for the naive matcher.

This approach provides fresh insights into fast partial evaluation and accurate program staging. This contribution summarizes [10, 11], and examines applications to program staging, and the relation to recursive pushdown systems.

References

1. M. S. Ager, O. Danvy, H. K. Rohde. Fast partial evaluation of pattern matching in strings. *ACM TOPLAS*, 28(4):696–714, 2006.
2. M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21(5):473–484, 1984.
3. M. A. Bulyonkov. Extracting polyvariant binding time analysis from polyvariant specializer. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 59–65. ACM Press, 1993.
4. N. H. Christensen, R. Glück. Offline partial evaluation can be as accurate as online partial evaluation. *ACM TOPLAS*, 26(1):191–220, 2004.
5. C. Consel, O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.
6. A. P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, 1977.
7. Y. Futamura, Z. Konishi, R. Glück. Program transformation system based on generalized partial computation. *New Generation Computing*, 20(1):75–99, 2002.
8. Y. Futamura, K. Nogi. Generalized partial computation. In D. Bjørner, A. P. Ershov, N. D. Jones (eds.), *Partial Evaluation and Mixed Computation*, 133–151. North-Holland, 1988.
9. R. Glück. A self-applicable online partial evaluator for recursive flowchart languages. *Software – Practice and Experience*, 42(6):649–673, 2012.
10. R. Glück. Maximally-polyvariant partial evaluation in polynomial time. In M. Mazzara, A. Voronkov (eds.), *Perspectives of System Informatics. Proceedings*, LNCS 9609. Springer-Verlag, 2016.
11. R. Glück. A practical simulation result for two-way pushdown automata. In Y.-S. Han, K. Salomaa (eds.), *Implementation and Application of Automata. Proceedings*, LNCS 9705. Springer-Verlag, 2016.
12. R. Glück, J. Jørgensen. Generating transformers for deforestation and supercompilation. In B. Le Charlier (ed.), *Static Analysis. Proceedings*, LNCS 864, 432–448. Springer-Verlag, 1994.
13. R. Glück, A. V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. In P. Cousot, et al. (eds.), *Static Analysis. Proceedings*, LNCS 724, 112–123. Springer-Verlag, 1993.
14. N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
15. M. H. Sørensen, R. Glück, N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

Introduction to Valentin Turchin's Cybernetic Foundation of Mathematics

(Summary of Tutorial)

Robert Glück[1] and Andrei Klimov[2⋆]

[1] DIKU, Department of Computer Science, University of Copenhagen
[2] Keldysh Institute of Applied Mathematics, Russian Academy of Sciences

Dedicated to the memory of V.F. Turchin (1931 - 2010) a thinker ahead of his time

Abstract. We introduce an alternative foundation of mathematics developed by Valentin Turchin in the 1980s — *Cybernetic Foundation of Mathematics.* This new philosophy of mathematics as human activity with formal linguistic machines is an extension of mathematical constructivism based on the notion of algorithms that define mathematical objects and notions as processes of a certain kind referred to as *metamechanical.* It is implied that all mathematical objects can be represented as metamechanical processes. We call this Turchin's thesis.

Keywords: constructive mathematics, alternative foundation of mathematics, Cybernetic Foundation of Mathematics, model of a mathematician, mechanical and metamechanical processes, objective interpretability, Turchin's thesis.

1 Introduction: Challenge of Foundation of Mathematics

Valentin Turchin's studies on the foundation of mathematics [1–4] comprise a highly underestimated part of his scientific legacy. Like some other great mathematicians of the 20th century, he was dissatisfied with the solution to the "crisis of mathematics" associated with the likes of David Hilbert and Nicolas Bourbaki - the formal axiomatic method. Its main drawback is that axioms, theorems and other formal texts of theories do not contain real mathematical objects. They refer to mathematical objects by means of variables and jargon, which are interpreted by human thought processes, but have no concrete representation in the stuff of language. Very few mathematical objects have names - constants like *true, false,* digits of numbers, *etc.* This corresponds to the Platonic philosophy of mathematics, an eternal immutable world of mathematical objects and perceived by minds and shaped in mathematical texts.

Valentin Turchin's take on mathematics is akin to *constructivism*: There is no Platonic world of mathematical objects. It is pure imagination. Mathematics is formal linguistic modeling of anything in the world including mathematics. Mathematical objects are abstract constructs represented by a formal language. The world of linguistic models is not static. Sentences in a formal language define processes that are potentially infinite

⋆ Supported by RFBR, research project No. 16-01-00813-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

sequences of states, where the states are also sentences in the language. These processes themselves are (representations of) mathematical objects. Mathematical objects are not a given, but are created by mathematicians in the process of their use of mathematical "machinery". This process of mathematical activity itself can be linguistically modeled, that is included into mathematics as well.

Turchin's thesis: Everything that humans may consider to be a mathematical object can be represented as a formal linguistic process. Valentin Turchin did not express this thesis explicitly. It is our interpretation of what we feel he implied.

The main problem is how to define a notion of linguistic processes to suit Turchin's thesis, that represents all objects mathematicians agree are mathematical. Valentin Turchin was not the first to imagine this kind of solution to the problem of foundation of mathematics. Attempts had been made by mathematicians in the early 20th century which failed. The idea was to equate the processes mentioned in the thesis with *algorithms*. Sophisticated mathematical theories were built on the basis of this idea in the 1950s and 1960s including *constructive mathematical analysis* that dealt with real numbers produced digit by digit by means of algorithms and functions that return their values digit by digit while gradually consuming digits of the arguments.

Unfortunately, such constructive theories were too limited to satisfy Turchin's thesis. For example, all constructive functions of real numbers are continuous; hence, discontinuous functions are excluded from that view of constructive mathematics. Working mathematicians reject this restriction. They consider a great amount of mathematical objects that cannot be represented by algorithms.

Thus mathematical constructivists met a great challenge: *How to define a wider notion of a linguistic process than algorithms, sufficient to represent mathematical objects.*

Valentin Turchin's *Cybernetic Foundation of Mathematics* (CFM) is a solution to this problem.

2 Basics of Cybernetic Foundation of Mathematics

CFM starts from the concept of constructive mathematics based on algorithms. Processes defined by algorithms are included in CFM. The whole of algorithmic and programming intuition works well in CFM. Valentin Turchin uses his favorite programming language Refal to define processes, but any other programming language will do, preferably a functional one in order to keep things as simple as necessary. For example, the main objects of modern mathematics - *sets* - are defined by processes that produce elements one by one. Thus, infinity is only *potential*: each element of an infinite set is produced at some finite moment of time, at no moment are all elements present together.

The next question is what kind of propositions about processes can be formulated and proved. Here Valentin Turchin departs from the solution adopted by the previous constructivists. He demonstrates that only two kinds of elementary propositions are needed:

1. that a given process (with given arguments) terminates at some step, and
2. that a given process never terminates.

All other statements about processes and mathematical objects can be expressed as propositions in this form, regarding processes defined by a mathematician in a chosen formal (programming) language.

This may sound strange (if not impossible) to mathematicians who studied the theory of computability and know about algorithmically undecidable problems.

Here Valentin Turchin introduces in mathematical machinery the concepts that we observe in the scientific activity in natural sciences. Traditionally, mathematics is not included in sciences. We often hear: "Mathematics and sciences". So one could that say Turchin has returned mathematics to the natural sciences.

Scientists operate well with statements they do not know in advance to be true: they propose testable *hypotheses, predictions,* whose characteristic feature is that they can be either confirmed or falsified at some moment in time; they hold the prediction true until falsified; if it is falsified they reconsider and state the negation of the prediction then propose the next hypotheses. Scientists *believe* there exists a trajectory without backtracking, a *process* of gradually increasing knowledge. This belief with respect to linguistic processes is the background of CFM.

The elementary propositions are predictions: The proposition that a given process terminates, cannot be confirmed while the process is running; but when it stops, the proposition is confirmed and becomes certainly true. The proposition that a process does not terminate can never be confirmed, but if it does it is falsified and mathematicians return back and add its negation (saying that the process terminates) to their knowledge.

When we explain the meaning of the propositions, predictions, and truths by referring to the notion of mathematicians, their knowledge, their proposing of predictions, and their process of gaining knowledge by confirmations and falsifications of the predictions, we speak about a *model of a mathematician.* This model is introduced into the mathematical machinery and becomes one of the first-class processes considered in CFM. The previous philosophy of mathematics assumes mathematics is *objective* in the sense that while doing research, mathematicians are outside of mathematics, they are merely observers of eternal mathematical phenomena, observers who do not influence the mathematical machinery but focus their attention on interesting phenomena to discover truths and proofs about them.

Valentin Turchin said that he had introduced mathematicians to mathematics in the same way as modern physics introduced physicists, as observers, to physical theories.

This sounds fantastic. Nevertheless Valentin Turchin has achieved this ambitious goal. In his texts [1–4] are many definitions of mathematical objects including the model of the classic set theory, in terms which modern mathematicians like to define their objects. Hence, he has proved that Turchin's thesis is met at least for objects expressible in set theory.

3 Mechanical and Metamechanical Process

In CFM, proper algorithms are referred to as *mechanical processes.* Processes of general form defined in the programming language of CFM theory can additionally access primitive *cognitive functions* that answer questions like "Does this process terminates or not?":

- if *process p terminates* then ... else ...
- if *process p never terminates* then ... else ...

If the answer is not yet confirmed or falsified for the respective branch of the conditional and we have no reason to consider it contradictory (this notion is also defined in CFM), a truth value can be assigned to it as a prediction.

Processes that use predictions as predicates are referred to as *metamechanical processes.* The collection of predictions that are available at a given moment of time is kept as the *current knowledge of (the model of) the mathematician.* It is considered as an infinite process producing all true predictions like other infinite processes.

Valentin Turchin demonstrated how all essential mathematical notions are defined as metamechanical processes and we, the authors of mathematical theories, should reason about them. Specific mathematical theories can introduce additional cognitive functions in the same style. For example, Valentin Turchin interpreted set theory with the use of the second cognitive primitive: the process that enumerates all sentences that define sets, that is, represents the universe of sets. (Those who remember the famous set-theoretical paradoxes should immediately conclude that this process cannot be a definition of some set and cannot be produced as an element of this collection.)

Reasoning about metamechanical processes is nontrivial if possible at all. Let us consider the most subtle point to lift the veil off CFM a little.

4 Objective Interpretability

Having introduced (the model of) mathematicians into the mathematical machinery by allowing access to their knowledge, we met (the model of) of free will: if the behavior of mathematicians and content of their knowledge are deterministic, than there is nothing essentially new in the notion of a metamechanical process. Many kinds of such generalizations of algorithms have already been considered in mathematics and found to be incomplete. Hence, we must expect that a complete (w.r.t. Turchin's thesis) notion of metamechanical processes inevitably allows us to define *non-deterministic* processes as well. This means that in the hands of one (model of the) mathematician one result may be produced (e.g., some process terminated), while in the hands of another (model of the) mathematician a different result is returned (e.g., the process with the same definition did not terminate).

This is the next strange thing of CFM, considered unacceptable by classic mathematicians. Valentin Turchin did not explain how to deal with non-deterministic processes in general. However, as far as the interpretation of classic mathematics, which all mathematicians believe to be deterministic, is concerned, it is natural to expect that the metamechanical processes used to define classic mathematical notions are deterministic.

In CFM, processes that produce the same results in hands of different instantiations of the model of the mathematician (that is with different initial true knowledge) are referred to as *objectively interpretable* metamechanical processes.

Thus, we, real mathematicians (not models), should use the CFM machinery in the following way:

1. Define some notions of a mathematical theory under study as processes in the programming language of CFM.
2. Prove somehow (in meta-theory, formally or informally, *etc.*) that the defined processes are objectively interpretable.
3. Then freely use these definitions in our mathematical activity and, in particular, in the definitions of next notions and learning truths about these processes.

One may ask: What is the reliability of (meta-) proofs in Item 2? — The answer is: It depends. It must be sufficient for your activities in Item 3. It is the choice of your free will.

Notice that not all definitions are subject to non-trivial proofs in Item 2. There is no problem with new definitions that do not call directly cognitive functions of the model of the mathematician and use only functions that are already proven objectively interpretable. Such definitions are automatically objectively interpretable as well.

5 Interpretation of Set Theory

Valentin Turchin has demonstrated how to define metamechanical processes and prove they are objectively interpretable for the classic mathematical logic and set theory. He considered in turn all axioms of the classic logic and Zermelo–Fraenkel set theory, gave definitions of the corresponding processes and proved their objective interpretability.

The proof techniques ranged from quite trivial, based on programmers' intuition, to rather sophisticated. Almost all Zermel–Fraenkel axioms are existential: they state existence of certain mathematical objects in set theory. Each existential statement is interpreted in the constructive style: by the definition of the process that meets the property formulated in the axiom. For example, the axiom of existence of the empty set is a process that returns the empty list in one step; the axiom of existence of the pair of two arbitrary sets is a process that produces the list of the two elements.

It is intriguing that Valentin Turchin did not manage to find a definition of a process and a proof of its objective interpretability for the axiom schema of replacement, which was the last added by Fraenkel to form the Zermelo–Fraenkel axioms. Its interpretation in CFM remains an open problem.

6 Open Problems

Valentin Turchin laid the foundations for new constructive mathematics. However, these are only the first steps and much remains to do to turn the new foundation of mathematics into a new paradigm. We list some evident open problems to provoke the reader:

– Complete Valentin Turchin's interpretation of the Zermelo–Fraenkel set theory. He defined the corresponding process for a particular case only, where an objectively interpretable function is given in the antecedent of the axiom rather than an objectively interpretable relation.

- Are there any interesting applications of non-deterministic, non-objectively interpretable processes? If yes, how can we apply them in our mathematical activity. CFM does not prohibit this, but Valentin Turchin did not suggest a way to deal with such definitions. He only demonstrated how to exclude them from our discourse.
- Formalize the way of Turchin's reasoning where he proved objective interpretability, in any modern mathematical theory of your choice. Find interesting applications where such formalization is difficult.
- Where does the CFM style of reasoning and thinking do better than the classical one? Find interesting applications where CFM may clarify or explain something better than can classic mathematics and the previous constructivism.
- Examine the correspondence of CFM with existing constructive theories. For example, how to compare CFM, without types and with a language of CFM statically untyped, with the dependent types theory and programming of proofs in a typed functional programming language.
- Develop a proof assistant for CFM, which will help us work with constructive definitions of metamechanical processes and check proofs of objective interpretability.
- It is fair to predict that supercompilation will become an important tool for the manipulation of mathematical definitions in CFM, and extraction and proofs of their properties. Will supercompilers be somehow extended to become effectively applicable to CFM code? Develop and implement a supercompiler to be used as a CFM proof assistant.

Acknowledgement

Valentin Turchin's texts on CFM have been translated into Russian with annotations by Vladimir Balter. We highly appreciate his work and plan to prepare the publication of CFM in English and in Russian.

The authors thank Sergei M. Abramov for generously hosting their joint work on CFM at the Ailamazyan Program Systems Institute, Russian Academy of Sciences. The first author expresses his deepest thanks to Akihiko Takano for providing him with excellent working conditions at the National Institute of Informatics, Tokyo.

References

1. Turchin, V.F.: The Cybernetic Foundation of Mathematics. Technical report, The City College of the City Univeristy of New York (1983), http://pat.keldysh.ru/~roman/doc/Turchin/1983_Turchin__The_Cybernetic_Foundation_of_Mathematics.pdf
2. Turchin, V.F.: The Cybernetic Foundation of Mathematics. i. the concept of truth. Unpublished (1983), http://pat.keldysh.ru/~roman/doc/Turchin/1983_Turchin__The_Cybernetic_Foundation_of_Mathematics_1_The_Concept_of_Truth.pdf
3. Turchin, V.F.: The Cybernetic Foundation of Mathematics. ii. interpretation of set theory. Unpublished (1983), http://pat.keldysh.ru/~roman/doc/Turchin/1983_Turchin__The_Cybernetic_Foundation_of_Mathematics_2_Interpretation_of_Set_Theory.pdf
4. Turchin, V.F.: A constructive interpretation of the full set theory. The Journal of Symbolic Logic 52(1), 172–201 (1987), http://dx.doi.org/10.2307/2273872

Nonlinear Configurations for Superlinear Speedup by Supercompilation

Robert Glück[1], Andrei Klimov[2⋆], and Antonina Nepeivoda[3⋆⋆]

[1] DIKU, Department of Computer Science, University of Copenhagen
[2] Keldysh Institute of Applied Mathematics, Russian Academy of Sciences
[3] Ailamazyan Program Systems Institute, Russian Academy of Sciences

Abstract. It is a widely held belief that supercompilation like partial evaluation is only capable of linear-time program speedups. The purpose of this paper is to dispel this myth. We show that supercompilation is capable of *superlinear* speedups and demonstrate this with several examples. We analyze the transformation and identify the *most-specific generalization* (msg) as the source of the speedup. Based on our analysis, we propose a conservative extension to supercompilation using *equality indices* that extends the range of msg-based superlinear speedups. Among other benefits, the increased accuracy improves the time complexity of the palindrome-suffix problem from $O(2^{n^2})$ to $O(n^2)$.

Keywords: program transformation, supercompilation, unification-based information propagation, most-specific generalization, asymptotic complexity.

1 Introduction

Jones *et al.* reported that partial evaluation can only achieve linear speedups [11, 12]. The question of whether supercompilation has the same limit remains contentious. It has been said that supercompilation is incapable of superlinear (s.l.) speedups (*e.g.*, [13]).

However, in this paper we report that supercompilation is capable of s.l. speedups and demonstrate this with several examples. We analyze the transformation and identify the *most-specific generalization* (msg) as the source of this optimization. Based on our analysis, we propose a straightforward extension to supercompilation using *equality indices* that extends the range of s.l. speedups. Among other benefits, the time complexity of the palindrome-suffix problem is improved from $O(2^{n^2})$ to $O(n^2)$ by our extension. Without our extension the msg-based speedup applies to few "interesting programs".

First, let us distinguish between two definitions of supercompilation:

$$\textit{supercompilation} = \textit{driving} + \textit{folding} + \textit{generalization} \tag{1}$$

and

$$\textit{supercompilation} = \textit{driving} + \textit{identical folding} \tag{2}$$

⋆ Supported by RFBR, research project No. 16-01-00813-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

⋆⋆ Partially supported by RFBR, research project No. 14-07-00133-a, and Russian Academy of Sciences, research project No. AAAA-A16-116021760039-0.

Examples of these have been described, SCP (1) [7, 17, 27, 28, 30] and SCP (2) [4, 25]. In brief, the former use various sophisticated whistle and generalization algorithms to terminate the transformation. In contrast, the latter only fold two configurations that are *identical modulo variable renaming* (α-identical folding) [25]. This means, SCP (2) terminate less often than do SCP (1). Nevertheless, non-trivial optimization can be achieved by SCP (2) due to the unification-based information propagation by driving, such as the optimization of a string matcher [4].

It was reported as proven that supercompilation, like partial evaluation, is not capable of s.l. speedups. However, that proof [23, Ch. 11] applies only to SCP (2) using identical folding without online generalization during supercompilation [23, Ch. 3], it does not apply to SCP (1).[4] We will show that s.l. speedups require folding and most-specific generalization.

We begin by giving an example that demonstrates that supercompilation is capable of s.l. speedups in Section 2. In Section 3, we briefly review driving and generalization of core supercompilation, the latter of which is the key to our main technical result. We characterize the limits of s.l. speedup by supercompilation in Section 4. In Section 5 we present our technique using equality indices which extend the range of superlinear speedups that a core supercompiler can achieve. Section 6 is the conclusion.

We assume that the reader is familiar with the basic notion of supercompilation (*e.g.*, [4, 7, 25]) and partial evaluation (*e.g.*, [12]). This paper follows the terminology [7] where more details and definitions can be found.

2 A Small Example of Superlinear Speedup by Generalization

To dispel the "myth" that supercompilation is only capable of linear speedups, let us compare supercompiler types, SCP(1) and SCP(2), using an example and analyze the transformations with and without generalization. This is perhaps the smallest example that demonstrates that an exponential speedup by supercompilation is possible.

The actual program transformations in this paper were performed by *Simple Supercompiler* (SPSC) [17][5], a clean implementation of a supercompiler with positive driving and generalization. The example programs are written in the object language of that system, a first-order functional language with normal-order reduction to weak head normal form [7]. All functions and constructors have a fixed arity. Pattern matching is only possible on the first argument of a function. For example, the function definition $f(S(x), y) = f(x, y)$ is admissible, but $f(x, S(y)) = f(x, y)$ and $f(S(x), S(y)) = f(x, y)$ are not.

Example 1. Consider a function f that returns its second argument unchanged if the first argument is the nullary constructor Z and calls itself recursively if it has the unary constructor S as the outermost constructor. The function always returns Z as a result. The computation of term $f(x, x)$ in the start function $p(x)$ takes exponential time $O(2^n)$ due to the nested double recursion in the definition of f (here, $n = |x|$ is the number

[4] Similar to partial evaluation, the result of a function call can be generalized by inserting let-expressions in the source program by hand (offline generalization) [2].

[5] SPSC home page http://spsc.appspot.com.

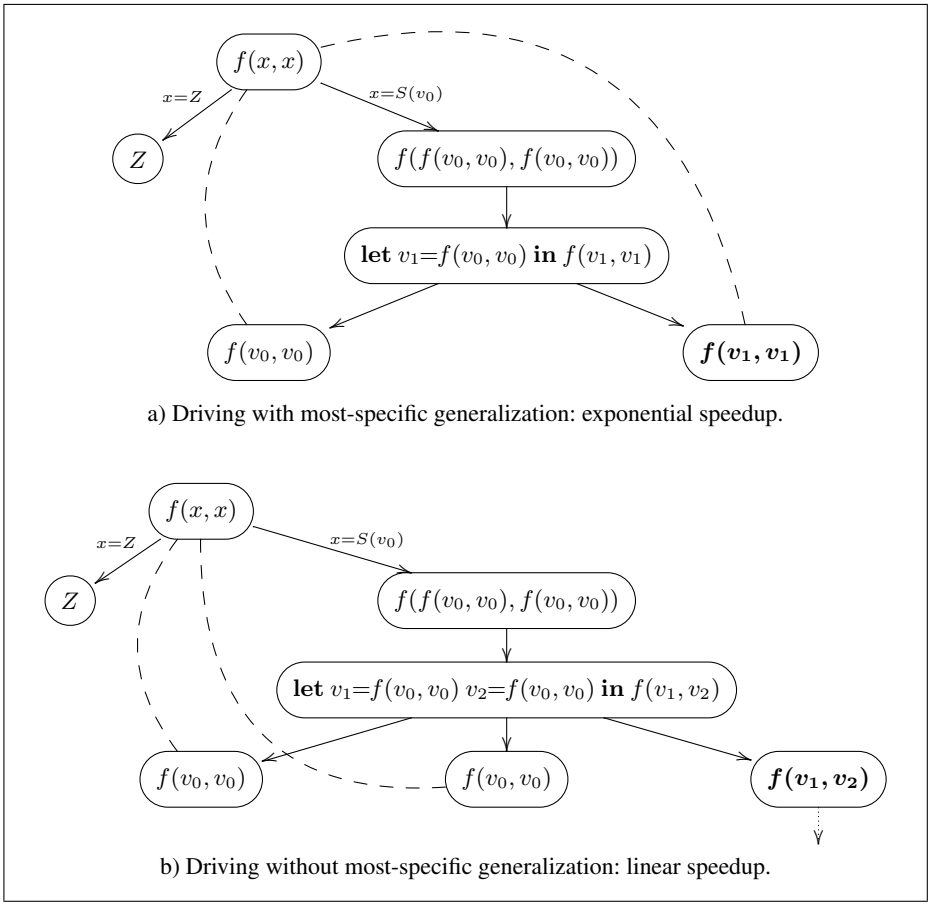


Fig. 1. The partial process trees of a supercompiler with and without generalization.

of S-constructors in the unary number x). The residual program f_1 produced by SPSC takes time $O(n)$ on the same input, that is an *exponential speedup* is achieved.[6]

Source program	Residual program
Start $p(x) = f(x, x)$;	Start $p1(x) = f_1(x)$;
$f(Z, y) = y$; $f(S(x), y) = f(f(x, x), f(x, x))$;	$f_1(Z) = Z$; $f_1(S(x)) = f_1(f_1(x))$;

To understand what causes this optimization, compare the two process trees produced by a supercompiler with and without generalization (Fig. 1a,b). In both cases, driving the initial term $s = f(x, x)$ branches into two subtrees. One for $x{=}Z$ and one for $x{=}S(v_0)$. Driving the left subtree terminates with term Z. In the right subtree each

[6] The speedup is independent of whether the programs are run in CBV, CBN or lazy semantics.

occurrence of x in the right-hand side of f is replaced by $S(v_0)$ and driving leads to the new term $t = f(f(v_0, v_0), f(v_0, v_0))$. The msg of the initial term s and the term t captures the repeated term $f(v_0, v_0)$:

$$\lfloor s, t \rfloor = (f(v_1, v_1), \{v_1 := x\}, \{v_1 := f(v_0, v_0)\}), \tag{3}$$

and leads to the creation of the generalized node in the process tree (a):

$$\textbf{let } v_1{=}f(v_0, v_0) \textbf{ in } f(v_1, v_1). \tag{4}$$

Both of the subterms in this node are instances of the initial term s and can be folded back to it. Hence, the process tree is closed and driving terminates. The linear-time residual program f_1 in Example 1 is obtained from process tree (a). Each function takes as many arguments as there are configuration variables in the corresponding configuration. Now only one variable remains, and the residual function becomes unary. It is the second rule of the msg (defined in Sect. 3) that unifies equal subterms and introduces term sharing in the process tree. We call this mechanism *msg-sharing*.

This optimization cannot be achieved by an SCP (2) supercompiler using identical folding without msg. In the same situation, as shown in process tree (b), such a supercompiler loses the connection between the two arguments in the body of the let-expression: $f(v_1, v_2)$. As a result, the original function definition is rebuilt in the residual program. In fact, without inserting let-expressions in one way or another, a supercompiler with identical folding does not terminate because the configurations continue to grow due to unfolding function calls. Like partial evaluation, s.l. speedup cannot be achieved by SCP (2). That supercompilation with msg can change the asymptotic time complexity was demonstrated above.

The speedup results from core positive supercompilation with msg [7], with no additional transformation techniques. The speedups are achieved both in call-by-name and call-by-value performance.

To illustrate the optimization, we compare side-by-side the runs of the source function and the residual function with unary input $SSSZ$ (abbreviated by 3). Fig. 2 shows the relevant normal-order reduction steps and omits some of the intermediate steps. The reader will notice that one call $f(3, 3)$ in the source-program run (left side) corresponds to one call $f_1(3)$ in the residual-program run (right side), two calls $f(2, 2)$ to one call $f_1(2)$, and four calls of $f(1, 1)$ to one call $f_1(1)$. In general, 2^m calls $f(n-m, n-m)$ correspond to one call $f_1(n-m)$ where n is the number represented by the unary input and $0 \leq m < n$. This shows that s.l. speedup relates to a variable-sized input, where the improvement increases superlinearly as the size of the input grows.

3 The Core Supercompiler

In this section we briefly review supercompilation with generalization. Since supercompilation is formally defined in the literature (*e.g.*, [7]) we will skip the full definition here and only review the algorithm of msg which introduces the sharing of terms.

A supercompiler takes an initial term and a program, and constructs a possibly infinite process tree. If the process tree is finite, a new term and a residual program are generated. The two main components of a supercompiler for developing the process tree are driving and generalization.

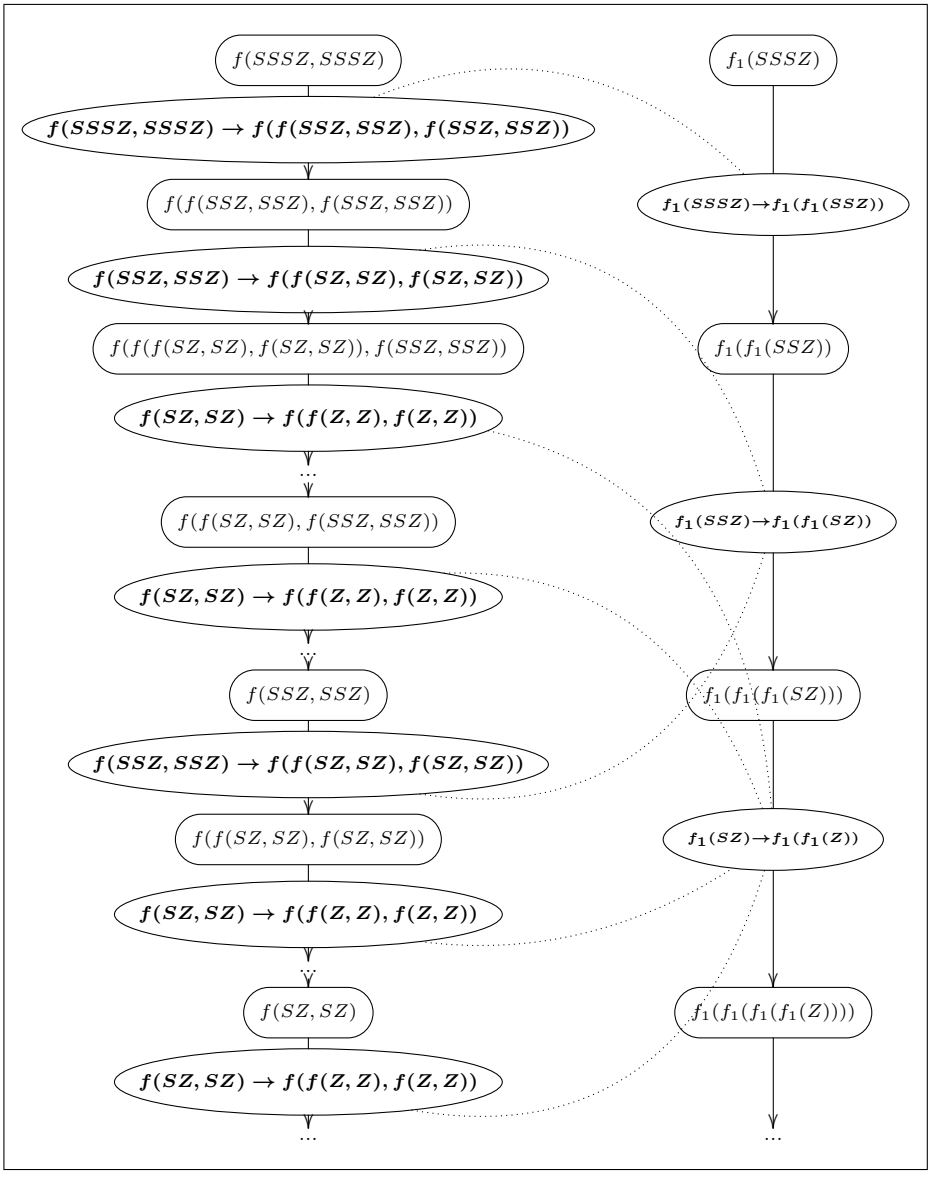


Fig. 2. Exponential- and linear-time runs of the source and residual functions of Example 1.

Driving makes use of *unification-based information propagation* to construct the process tree during supercompilation. Common to all driving methods regardless of their degree of information propagation [4] is the use of *configuration variables* and *non-linear configurations*. These features distinguish driving from the constant-based information propagation of partial evaluation. It is essential that configuration variables may

occur repeatedly in a configuration. This is key to expressing term equality and sharing results of a subcomputation as demonstrated in Example 1.

Generalization To ensure that a finite process tree is constructed from which a residual program can be generated, supercompilation uses the most specific generalization. The result of the msg is used to decide whether to continue driving, fold to an existing configuration, or to generalize two configurations to a new one. There are various strategies to choose which configurations the msg is applied to and how its result is used including the use of transient configurations, upward and downward generalization. The particular strategy is irrelevant to this paper except that different strategies present more or less sharing opportunities to the msg.

We review the definition of msg used in positive supercompilation (taken from [24]). The msg of two terms, $\lfloor s, t \rfloor$, is computed by exhaustively applying the following rewrite rules to the triple $(x, \{x := s\}, \{x := t\})$. The result is the msg $(t_g, \theta_1, \theta_2)$ including a generalized term t_g and two substitutions θ_1 and θ_2 such that $t_g\theta_1 = s$ and $t_g\theta_2 = t$. Any two terms s and t have an msg which is unique up to renaming.

$$\begin{pmatrix} t_g \\ \{x := \sigma(s_1, \ldots, s_n)\} \cup \theta_1 \\ \{x := \sigma(t_1, \ldots, t_n)\} \cup \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g\{x := \sigma(y_1, \ldots, y_n)\} \\ \{y_1 := s_1, \ldots, y_n := s_n\} \cup \theta_1 \\ \{y_1 := t_1, \ldots, y_n := t_n\} \cup \theta_2 \end{pmatrix}$$

$$\begin{pmatrix} t_g \\ \{x := s, y := s\} \cup \theta_1 \\ \{x := t, y := t\} \cup \theta_2 \end{pmatrix} \rightarrow \begin{pmatrix} t_g\{x := y\} \\ \{y := s\} \cup \theta_1 \\ \{y := t\} \cup \theta_2 \end{pmatrix}$$

Take for example the msg of the two configurations in Eq. 3 of Example 1. It identified the term-equality pattern in terms and led to the sharing by a let-expression.

An important property is that msg-based generalization is *semantics preserving*. The branches merged in the process tree are identical. No computation is deleted, and only the result of a computation is shared by a let-expression. The term equality expressed by a let-expression, *e.g.* $\mathbf{let}\ v_1{=}f(v_0, v_0)\ \mathbf{in}\ f(v_1, v_1)$.

The sharing that supercompilation introduces may look like common subexpression elimination, but it is a *dynamic property* that occurs only during driving as the following example illustrates.

Example 2. Consider a variant of Example 1 where the right-hand side of function g contains the permuted terms $f(x, y)$ and $f(y, x)$. As before, given the initial configuration $f(x, x)$, the supercompiler achieves an exponential speedup. The term equality is discovered by the msg because the configuration variables are propagated during driving. Common subexpression elimination will not discover this equality.

Source program	Residual program
Start $p(x) = f(x, x)$;	Start $p1(x) = f_1(x)$;
$f(Z, y) = y$; $f(S(x), y) = g(y, x)$; $g(Z, y) = y$; $g(S(x), y) = f(f(x, y), f(y, x))$;	$f_1(Z) = Z$; $f_1(S(x)) = f_1(f_1(x))$;

4 Limitations of Sharing by Supercompilation

We identified the msg as the mechanism that introduces sharing of equal terms. We demonstrated that s.l. speedups are possible by supercompilation contradicting a common belief, but found few "interesting problems" to which this applies. For example, why does supercompilation not improve the well-known naive, exponential-time Fibonacci function? Some of the limitations that we identified are due to the strategy of developing process trees, *e.g.* identical configurations are not shared across subtrees. Other limitations are due to the unselective application of the msg to configurations, which leads to a loss of term equalities in the process tree. In this section, we examine two important limitations of core supercompilation. For the latter, we propose a more accurate technique of equality indices in the following section.

Limitation 1: No sharing across subtrees The speedups described so far are due to the sharing of computations. Core supercompilation does not identify all equal subterms in a single term (unlike the general form of jungle driving [21]). The msg identifies equal subterms when comparing *two configurations*, each of which contains equal subterms in the *same positions*. Thus, after a configuration is decomposed into subterms, *e.g.* by a let-expression, equal subterms are no longer identifiable as equal by the msg. The msg works *locally* on two given configurations in the process tree. Therefore, well-known examples with obvious opportunities for sharing cannot be improved by msg-sharing. They require a different supercompilation strategy for building process trees. This includes the naive Fibonacci function which requires the sharing of equal terms across subtrees. Let us illustrate this limitation.

Example 3. Consider the naive Fibonacci function:

$$
\begin{array}{l}
f(Z) = S(Z);\\
f(S(x)) = \mathit{ff}(x);\\
\mathit{ff}(Z) = S(Z);\\
\mathit{ff}(S(x)) = add(f(x), f(S(x))).
\end{array}
$$

There are several repeated terms in the process tree of the naive Fibonacci function (Fig. 3), but they occur in separate subtrees. For example, the terms $\mathit{ff}(v_2)$ as well as $\mathit{ff}(v_3)$. The msg alone cannot find equal nodes across different subtrees. To share such terms requires another strategy of building process trees in the supercompiler. Clearly, msg-sharing cannot speed up the naive Fibonacci function, which is a limitation of the supercompilation strategy of building process trees.

Limitation 2: Unselective generalization Two configurations with the same pattern of equal subterms are more likely to appear when a generalization in a branch of the process tree with certain contractions of the variables is considered than when a generalization with the initial configuration is done. But in that case, the general configuration which is a parent of the more specific configuration may be generalized with this configuration resulting in the loss of the subterm equalities. This limitation of the strategy of applying msg-sharing can be avoided by a more accurate generalization as we will show in the next section. Let us illustrate this problem.

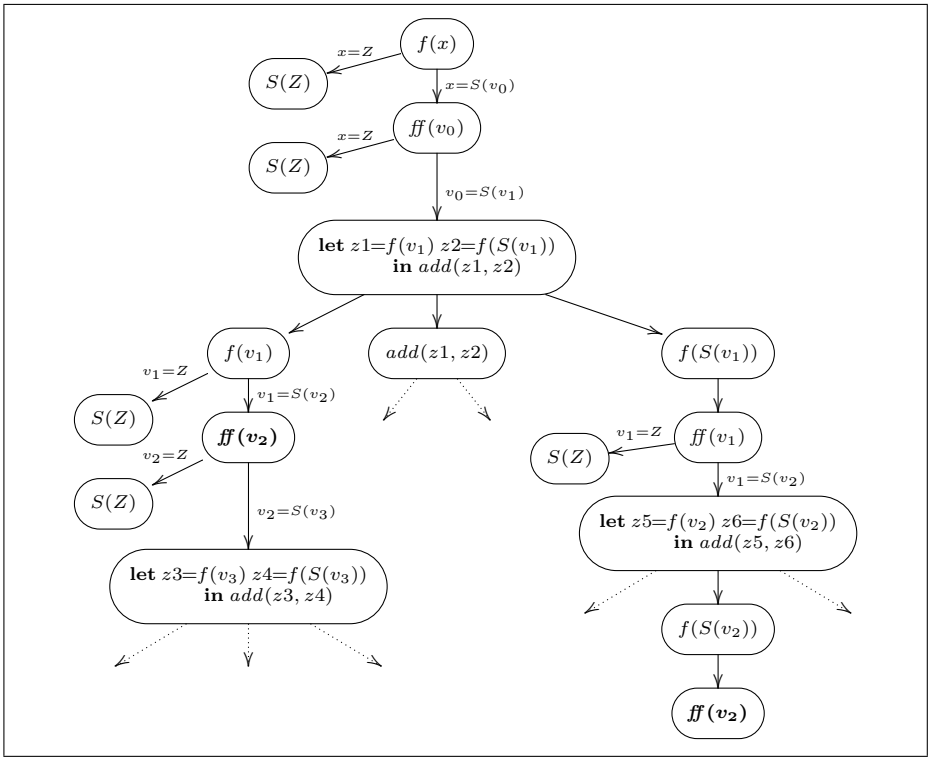


Fig. 3. A fragment of the process trees of the naive Fibonacci function.

Example 4. We replace the initial configuration of Example 1 by $p(x, y) = f(x, y)$. Given the process tree for $p(x, y)$, the configurations $f(x, y)$ and $f(f(x, x), f(x, x))$ are generalized to $f(x, y)$. After such a generalization, no speedup can be achieved by sharing. The residual program replicates the source program and no speedup is achieved. However, the second argument of the initial configuration is significant only for the first evaluation of $f(x, y)$. In all following evaluation steps, the initial value of y does not occur, and all calls of f have equal first and second arguments. The lesson learned from this example is that term equalities should be preserved in configurations. We shall use this observation as the basis for our extension.

Source program	Residual program
Start $p(x, y) = f(x, y)$;	Start $p(x, y) = f_1(x, y)$;
$f(Z, y) = y$; $f(S(x), y) = f(f(x, x), f(x, x))$;	$f_1(Z, y) = y$; $f_1(S(x), y) = f_1(f_1(x, x), f_1(x, x))$;

5 Increasing the Accuracy of Sharing: Equality Indices

Based on our analysis above, namely the generalization of term equalities in Example 4, we propose a new, conservative extension to supercompilation that makes the *applica-*

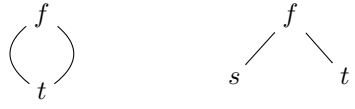
tion of the msg more accurate. We propose *equality indices* to reduces the loss of term equalities. Our goal is not to change the definition of the msg or the driving strategy, but to exploit the msg such that more term equalities can be preserved. We want residual programs to contain as many shared function calls as possible. A technique that assists in this task must distinguish function calls that have different term-equality patterns. Also, the technique must be applicable after each driving step during supercompilation because term equalities in configurations are a dynamic property (*cf.* Example 2).

5.1 How it Works

We examined several examples that show there are more chances for a deep optimization by preserving the term equalities uncovered dynamically during driving. It is more likely that this can be exploited by supercompiling a call with term equalities, *e.g.* $f(t,t)$, than a call without such equalities, *e.g.* $f(s,t)$ where $s \neq t$. We have seen in Example 4 that the loss of these equalities during msg limits the overall optimization by the supercompiler. Example 1 showed that in the residual program the arity of $f(t,t)$ is *reduced* to $f_1(t)$ provided a process graph can be built preserving this equality.

Our goal is to increase the accuracy of sharing by distinguishing calls that contain different patterns of syntactic term equalities. Our approach has two components. The annotation of all calls with equality indices, and the synchronization of annotated terms after driving and before calculating the msg.

First, let us annotate all function names in a configuration with equality indices. For example, the two calls from above are annotated as $f_{[1]}(t,t)$ and $f_{[2]}(s,t)$. Equality indices are textual representations of equality patterns, which can be seen with function calls represented by directed acyclic graphs (dag):



The msg will treat function names with different equality indices as different functions, thereby avoiding the loss of subterm equalities. The form and calculation of the indices, also for the nested case, will be treated in more detail below. For now let it suffice to say that they textually represent different sharing patterns (subterm equalities in calls).

The second issue that we need to approach is that driving can obscure term equalities. For example, driving the following nested call in Example 4 forces the unfolding of the first argument (underlined) and a substitution on the second argument (underlined). This yields in one step a configuration where the term equality disappears (the pattern matching on the first argument in a function definition forces the first argument to be unfolded, but not the second argument).

$$\begin{array}{rcl} f(\underline{f(x,x)}, f(x,x)) & \equiv & f(t,t) \\ \downarrow \text{ driving with } x = S(v_1) & & \\ f(\underline{f(f(v_1,v_1), f(v_1,v_1))}, f(\underline{S(v_1)}, \underline{S(v_1)})) & \equiv & f(t',t''), t' \neq t'' \end{array} \tag{5}$$

Even though both arguments are identical before driving, this is obscured by unfolding the first one and substituting $S(v_1)$ into the second one. In fact, both arguments should be driven at the same time (in lockstep) to preserve the synchronization!

The new configuration should be as follows.

$$f(\underline{f(f(v_1,v_1),f(v_1,v_1))},\underline{f(f(v_1,v_1),f(v_1,v_1))}) \;\equiv\; f(t',t') \tag{6}$$

There are various ways to drive two (or more) terms in lockstep. Our solution is simpler and does not require an extension of the driving strategy. We restore ("synchronize") the argument equality after the driving step using the equality indices. This simple syntactic technique will be made more precise below, also for multiple arguments.

For our extension we need two small algorithms:

1. Annotate every call in a configuration with an equality index, *e.g.* $f(t,t) \overset{ind}{\to} f_{[1]}(t,t)$.
2. Synchronize all arguments in a driven configuration using the equality indices, *e.g.*

$$f_{[1]}(f_{[1]}(t,t),S(t)) \overset{sync}{\to} f_{[1]}(f_{[1]}(t,t),f_{[1]}(t,t))\,.$$

The two operations will be performed after driving and before applying the msg:

$$\text{driven configuration} \to (1)\ \text{index} \to (2)\ \text{synchronize} \to \text{msg} \to \text{drive} \tag{7}$$

A node in the process tree will from then on contain a term annotated with equality indices, t_{Ind}. This amplifies the power of the msg-based sharing mechanism that is present in core supercompilation, and does not require a fundamental change of supercompilation. It is not difficult to add this refinement to an existing system.

5.2 The Equality-Index Algorithm

We now present the technique in more detail. First, for each function call we find the arguments which are equal, and identify them by marking each function name with an equality index. Driving itself ignores the indexes, whereas the msg treats function names with different indices differently. The equality indexes are used after each driving step to restore argument equalities. The two algorithms, indexing and synchronization, are applied after each driving step and before the whistle and msg algorithms are applied.

An *equality index* lists for each argument in a function call the smallest number of the argument to which it is syntactically equal, in particular its own number if all arguments to its left are different. For brevity, the equality index for the first argument is omitted as it is always equal to 1.

For example, a call $f(x,x,z)$ is marked with the equality index $[1,3]$, which means that the second argument x is equal to the first argument x, and the third argument z is not equal to an argument to its left. Not all index combinations are possible, *e.g.* no call can be marked by $[1,2]$. A call $f(t_1,\ldots,t_n)$ in which all arguments differ has the index $[2,\ldots,n]$ and a call $f(t,\ldots,t)$ with identical arguments has the index $[1,\ldots,1]$.

Given a node in the process tree of a program and the driven configuration t in the node, we apply the following two algorithms to update the configuration.

First, we construct an annotated version of the configuration, which differs from the plain configuration only by function call indexes.

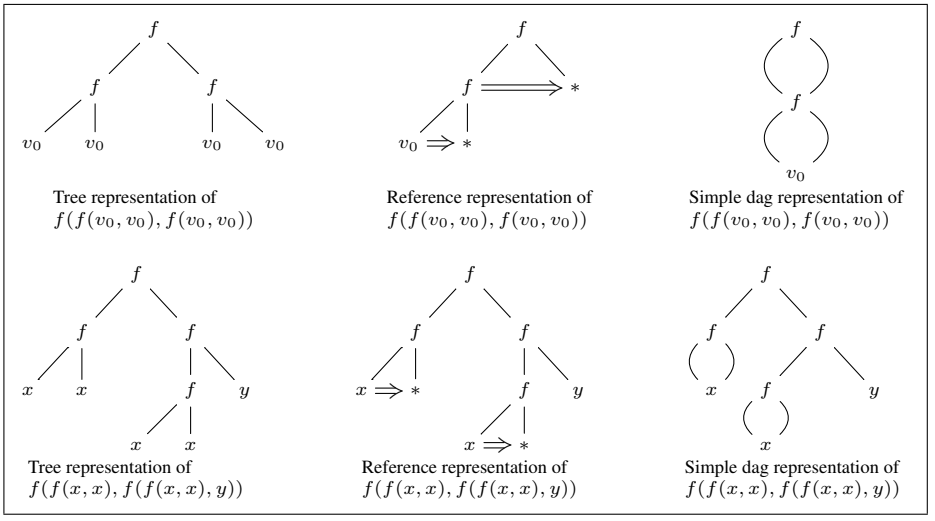


Fig. 4. Tree representations of terms vs. simple dag representations. A double arrow in the reference representation indicates the equality of arguments in a function call.

Algorithm 1 (Assigning equality indexes) *1. Given a configuration t, choose the innermost, leftmost function call in t that is not annotated. Let that call be $f(t_1, \ldots, t_n)$.*

(a) Assign auxiliary values $i = 2$, $j = 1$, $\Gamma = []$ (the empty list).

(b) If $t_i = t_j$ (syntactically), append $[j]$ to Γ, increase i, and set $j = 1$. Otherwise, increase j and repeat step (b). When j reaches i, $t_i = t_j$ is always true. So the number of steps in (b) is finite.

(c) When i reaches $n + 1$, rename this call of f in t to f_Γ. Then mark it as an annotated call and proceed with step (1).

Γ is *the equality index*. The length of the list is equal to the arity of the indexed function minus 1. This index shows which arguments of the call repeat each other. The equality index makes function calls with different argument equalities differ from each other, which preserves the equalities during the msg.

We illustrate how the Algorithm 1 works on term $f(f(v_0, v_0), f(v_0, v_0))$. The innermost, leftmost function call is $f(v_0, v_0)$, which is the first argument of the outer call. Step (a) assigns $i = 2$, $j = 1$, $\Gamma = []$. Step (b) checks whether $t_1 = t_2$ (the two arguments of the call f). Because $v_0 = v_0$, we get $\Gamma = [1]$ and change i to 3. Step (c) applies and the call is renamed to $f_{[1]}(v_0, v_0)$. The term becomes $f(f_{[1]}(v_0, v_0), f(v_0, v_0))$. The three steps are repeated with the second call $f(v_0, v_0)$ and the next term is $f(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0))$. Finally, after repeating the steps with the outermost call, the final term is $f_{[1]}(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0))$. The indices in this term identify for each argument the leftmost identical argument (Fig. 5, left tree).

All functions have a fixed arity, so for each function the set of possible equality indices is finite. The annotation of a binary function can lead to at most two different functions. A ternary function has at most five different equality indices and a function

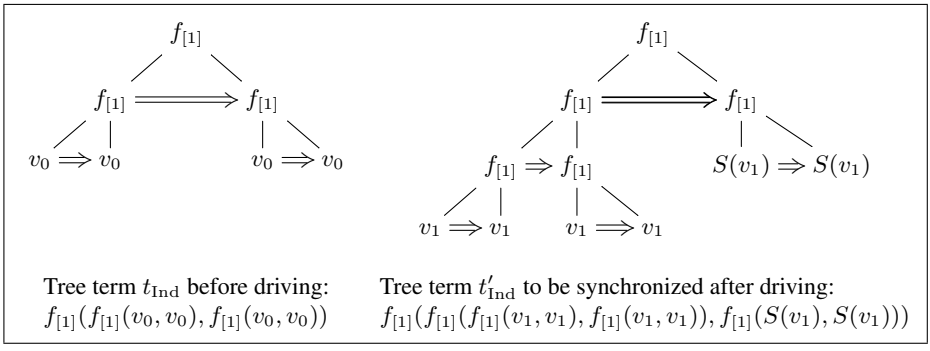


Fig. 5. Equality indexes identify the leftmost identical argument in a function call before driving, and are used to restore the lost argument equalities after driving (bold double arrow, right tree).

with four arguments has at most 15 different equality indices (and it is rare that all 15 arrangements appear during the supercompilation of the same program). Usually, the same function has only a few different equality indices.

The second algorithm restores obscured argument equalities using equality indexes.

Algorithm 2 (Synchronization) *1. Given an indexed term t_{Ind}, choose the innermost, leftmost unsynchronized function call. Let that call be $f_{[k_2,\ldots,k_n]}(t_1, \ldots, t_n)$ and set $j = 2$.*

2. *If $k_j = 1$, replace the k_j-th argument by the first argument. Otherwise, do nothing. Then, in both cases, increase j by 1.*
3. *When $j = n+1$, proceed with step (1) because the synchronization of function call $f_{[k_2,\ldots,k_n]}(t_1, \ldots, t_n)$ is finished. Otherwise, proceed with step (2).*

For example, given the term $f_{[1]}(f_{[1]}(v_0, v_0), S(v_1))$, the innermost unsynchronized call is $f_{[1]}(v_0, v_0)$. Since the equality index contains 1 at the first position, we perform the replacement and obtain the unchanged call $f_{[1]}(v_0, v_0)$. Next, the innermost unsynchronized call is the outermost call $f_{[1]}(\ldots)$. Again, we replace the second argument by the first one. The synchronized term becomes $f_{[1]}(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0))$.

Given calls $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$. If $\forall i, j \,.\, t_i = t_j \Leftrightarrow s_i = s_j$ then the equality indices for $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_n)$ are identical. Hence, if the msg treats function names with different equality indices as different functions, it will never generalize two equal arguments of a call to two unequal terms.

Correctness Why does the synchronization not change semantics of the computation? The answer is that an argument can be copied in the call iff on some step of the computation it is syntactically identical to the other argument. If the call whose argument is copied retains its equality index, and the arguments of the call lose their syntactic equality, it means that the call itself was not driven itself — only its first argument. The source language guarantees that the driven argument is always the first argument — and the first argument is never changed by the synchronization. So, when the term is synchronized, its arguments are replaced only by semantically equal arguments that

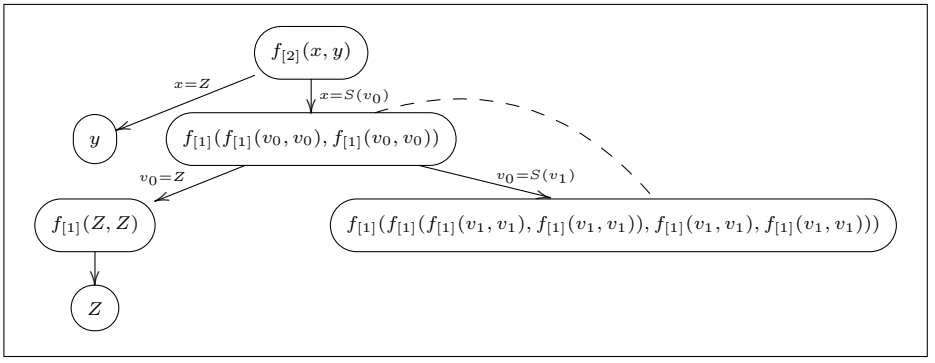


Fig. 6. Unfolding the process tree with equality indices of Example 4.

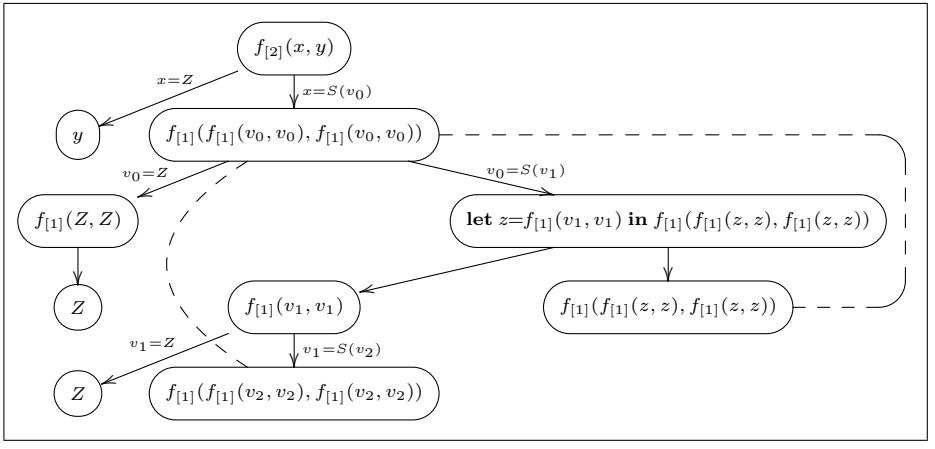


Fig. 7. The closed process tree with equality indices of Example 4.

came through more driving steps. Thus, the termination property of the program is preserved.[7]

Example 5. After introduction of the equality indexes, the process tree of Example 4 is constructed as follows. The first call $f(x, y)$ is annotated by equality index $[2]$ because its two arguments are syntactically different.

Under the assumption that $x = S(v_0)$, driving yields the next configuration

$$f(f(v_0, v_0), f(v_0, v_0))\,. \tag{8}$$

First, the two innermost calls $f(v_0, v_0)$ are annotated by equality index $[1]$ (which means that the second argument is equal to the first). Then the outermost function call

[7] The source language plays a key role in this reasoning. If the language admits patterns with static constructors also in other argument positions, Algorithm 2 must be changed. We must not only copy the first argument, but any argument in which the actual driving step was done.

is also annotated by equality index $[1]$ (its two arguments are equal). We obtain the indexed configuration (Fig. 5, left tree)

$$f_{[1]}(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0)) \,. \tag{9}$$

Although driving this configuration unfolds the call of f in the first argument of the outermost call and substitutes into the second argument due to assumption $v_0 = S(v_1)$,

$$f_{[1]}(f(f(v_1, v_1), f(v_1, v_1)), f_{[1]}(S(v_1), S(v_1))) \,, \tag{10}$$

by which the two arguments become different, the configuration is synchronized to

$$f_{[1]}(f_{[1]}(f_{[1]}(v_1, v_1), f(v_1, v_1)), f_{[1]}(f_{[1]}(v_1, v_1), f(v_1, v_1))) \tag{11}$$

because the equality index $[1]$ of the outermost call of f is unchanged by the driving step (Fig. 5, right tree) and the synchronization algorithm restores the lost syntactic equality of the two arguments (Fig. 5, bold double arrow, right tree). The assumption $v_0 = S(v_1)$, which was used for the substitution into the second argument, remains in the process tree (Fig. 6). Thus, no important driving information is lost by synchronization.

After the last step of driving, the msg of the two configurations (9) and (11),

$$\begin{array}{l} \lfloor f_{[1]}(f_{[1]}(v_0, v_0), f_{[1]}(v_0, v_0)), \\ \; f_{[1]}(f_{[1]}(f_{[1]}(v_1, v_1), f_{[1]}(v_1, v_1)), f_{[1]}(f_{[1]}(v_1, v_1), f_{[1]}(v_1, v_1)))\rfloor \\ = (f_{[1]}(f_{[1]}(z, z), f_{[1]}(z, z)), \theta_1, \theta_2) \,, \end{array} \tag{12}$$

leads to the creation of the following generalization node in the process tree (Fig. 7):

$$\textbf{let } z{=}f_{[1]}(v_1, v_1) \textbf{ in } f_{[1]}(f_{[1]}(z, z), f_{[1]}(z, z)) \,. \tag{13}$$

Finally, after driving $f_{[1]}(v_1, v_1)$ in (13), we obtain the closed process tree in Fig. 7. A linear-time residual program is generated from this tree (shown below). An exponential speedup is achieved by *supercompilation with equality indices*, which was not possible in Example 4 without the generalization precision induced by the equality indices.

Source program	Residual program
Start $p(x, y) = f(x, y)$;	Start $p(x, y) = f_2(x, y)$;
$f(Z, y) = y$; $f(S(x), y) = f(f(x, x), f(x, x))$;	$f_2(Z, y) = y$; $f_2(S(x), y) = \mathit{ff}(x)$; $\mathit{ff}(Z) = Z$; $\mathit{ff}(S(x)) = \mathit{ff}(f_1(x))$; $f_1(Z) = Z$; $f_1(S(x)) = \mathit{ff}(x)$;

In the residual program, the initial function f_2 has the general form, and functions ff and f_1 are generated as a specialization of f having two equal arguments.

We conclude that the equality indices algorithm makes more calls with identical arguments appear in the process tree. This method increases the precision of generalization without changing the driving and msg algorithms of the core supercompiler, but it is not powerful enough to handle old problems such as optimization of the naive Fibonacci function (Sect. 4). Equality indices do not give references to equal terms other than those local in the arguments of a function call, but they are an effective mechanism to preserve important argument equalities in a core supercompiler.

5.3 More Examples of the Equality-Index Algorithm

By using the equality indexes algorithm, we can achieve a superlinear speedup as per the following example.

Example 6. The palindrome-suffix program returns a suffix of either x or y that is a palindrome. If $x = A(A(A(B(Z))))$ and $y = A(B(A(B(Z))))$ then the program returns $B(A(B(Z)))$. The program has run time $O(2^{n^2})$ where $n = |x| + |y|$, that is the number of A- and B-constructors in x and y. Supercompilation without equality indices does not build a superlinear speedup of the program.

Source program	
Start: $p(x, y) = f(x, y)$;	$pal(A(z)) = and(palA(z, Z()), pal(z))$;
$f(x, y) = g(pal(x), pal(y), x, y)$;	$pal(B(z)) = and(palB(z, Z()), pal(z))$;
$g(T, z, x, y) = gt(z, x, y)$;	$palA(A(x), y) = palA(x, A(Z))$;
$gt(T, x, y) = x$;	$palA(B(x), y) = palA(x, B(Z))$;
$gt(F, x, y) = fmin1(y, y)$;	$palA(Z(), y) = ifA(y)$;
$gf(T, x, y) = y$;	$palB(A(x), y) = palB(x, A(Z))$;
$gf(F, x, y) = f(fmin1(x, x), fmin1(y, y))$;	$palB(B(x), y) = palB(x, B(Z))$;
$fmin1(A(x), y) = fmin2(y, x)$;	$palB(Z(), y) = ifB(y)$;
$fmin1(B(x), y) = fmin2(y, x)$;	$ifA(A(x)) = T$;
$fmin2(A(y), x) = f(x, y)$;	$ifA(B(x)) = F$;
$fmin2(B(y), x) = f(x, y)$;	$ifA(Z()) = F$;
$and(T, x) = x$;	$ifB(B(x)) = T$;
$and(F, x) = F$;	$ifB(A(x)) = F$;
	$ifB(Z()) = F$;

Function $fmin1$ that removes the first letter from the list always has equal arguments. However, equality is lost because the function call $f(x, x)$ after execution of $fmin1$ is generalized with $f(x, y)$ which has unequal arguments.

Consider the fragment of the process tree in Fig. 8. The call of $f_{[1]}$ with two equal arguments is generalized only with another call of $f_{[1]}$ with two equal arguments. Thus, the functions are transformed to functions with fewer arguments, and an unary version of f is generated. The residual program with the residual version of $fmin2$ calling the unary version of f, and the unary version of f calling the binary version of g has run time $O(n^2)$ where $n = |x|$.

Example 7. Now consider a negative example where the equality indexes do not help. Let us drive the naive Fibonacci function (Example 3) from an initial configuration $f(x_0)$. When the common order of driving is used by choosing the leftmost redex, repeated function calls are never observed in any configuration. To reveal the repeated calls inherent in the Fibonacci function the following strategy of supercompilation should be used.[8] After each contraction of a configuration variable and making a step of the redex, consider all function calls in the current configuration and make all steps that

[8] To the best of the authors' knowledge, it has not been implemented in any supercompiler.

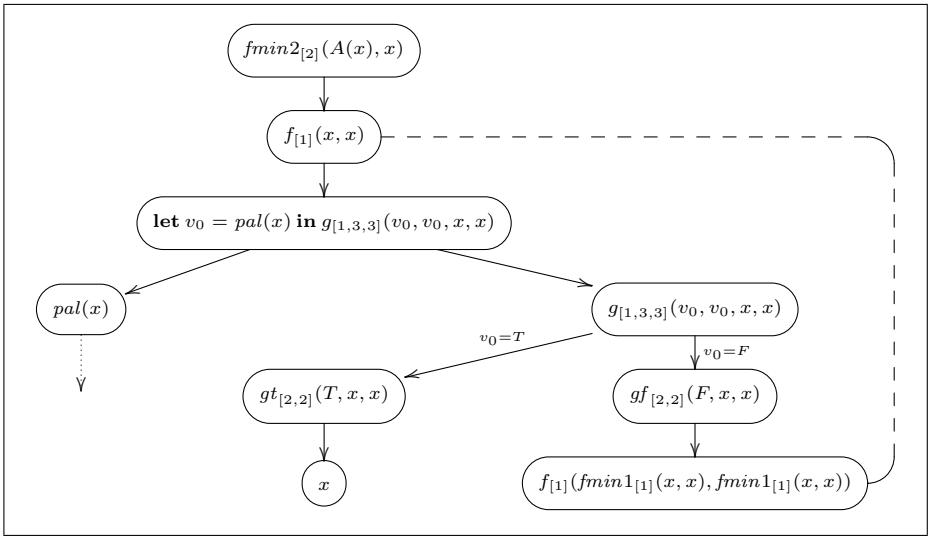


Fig. 8. A fragment of the updated process tree of Example 6.

become possible after substitution of the contraction, namely "normalize" the configuration. Then the repeated call $\mathit{ff}(x_3)$ is found at the fifth step along the following path:

$$\begin{array}{rcl} & & f(x_0) \\ x_0 = S(x_1) & \longrightarrow & \mathit{ff}(x_1) \\ x_1 = S(x_2) & \longrightarrow & add(f(x_2), f(S(x_2)) \\ & \longrightarrow & add(f(x_2), \mathit{ff}(x_2)) \\ x_2 = S(x_3) & \longrightarrow & add(\mathit{ff}(x_3), add(f(x_3), f(S(x_3))) \\ & \longrightarrow & add(\underline{\mathit{ff}(x_3)}, add(f(x_3), \underline{\mathit{ff}(x_3)})) \end{array} \tag{14}$$

The equality indexes do not capture the repeated call $\mathit{ff}(x_3)$ as it occurs as arguments of two different calls to add. Moreover, with the common termination strategy based on homeomorphic embedding, this configuration is not reached since the whistle blows earlier and premature generalization is performed. Hence, to superlinearly speed-up definitions, such as the Fibonacci example a new termination strategy is needed that enables performance of more driving steps before generalization.

5.4 Discussion

The two parts of our method can be considered separately.

Equality indexes and their treatment as function names can be used to construct a whistle that is more precise than the homeomorphic embedding relation used in the standard generalization algorithm [24]. It may be combined together with known refinements of the homeomorphic embedding relation, *e.g.* [20], forcing the relation to distinguish between strict and non-strict substitutions. In simple cases like Example 4,

using the strict embedding alone can also avoid the problem of unselective generalization. In case of comparing other configuration, *e.g.* $f(x, y)$ and $f(g(x, x, y), g(x, x, y))$, none of the refined embedding relations ($\trianglelefteq^*$, $\trianglelefteq_{var}$, $\trianglelefteq^+$) [20] can steer clear of the unselective generalization. Equality indices help to avoid the loss of sharing in this case.

Synchronization preserves term equalities including any syntactically lost due to standard driving. The synchronization feature can be viewed as a very simple version of jungle driving [21] that treats equal terms as one, but due to its simplicity and textual representation, our approach does not require a change of driving.

Partial deduction and driving are transformation techniques that achieve their transformational effects by nonlinear configurations [6]. Thus, our method may be not only useful in supercompilation, but in the context of verification by abstraction-based partial deduction [5] and other advanced program transformers [3].

6 Conclusion

We demonstrated how core supercompilation as it was originally defined by Valentin Turchin and used in many works, is capable of achieving superlinear (up to exponential) speedups on certain programs that repeatedly evaluate some function calls with the same arguments. Although core supercompilation does not check for repeated subterms in configurations explicitly (although it could), it contains an operation that captures equal terms implicitly: *most specific generalization*.

We described two types of supercompilation: SCP (2) using identical folding without online generalization, and SCP (1) with more sophisticated folding and generalization strategies. Identical folding limits SCP (2) to linear speedups similar to the limit of partial evaluation, but performs deeper transformations than partial evaluation due to the unification-based information propagation of driving (*e.g.*, SCP (2) passes the KMP-test [4, 25] which partial evaluation does not [12]).

Folding is a powerful technique which in combination with other transformation techniques can perform deep program optimizations (*e.g.*, improve the naive Fibonacci function [1]). The limitation of SCP (2) lies only in the restriction to identical folding, not in driving. We showed that even a core supercompiler using driving and most-specific generalization could change the asymptotic time complexity of programs, and dispelled the myth that supercompilation is only capable of linear-time program speedups.

A possible reason for the persistence of this myth is that it is sometimes forgotten that generalization without checking the equality of subterms, *e.g.* the generalization of $f(g(x), g(x))$ and $f(h(a, b), h(a, b))$ to $f(x, y)$, is *not* most specific. The most specific in this case being the generalization to $f(x, x)$ with two substitutions $\{x := g(x)\}$ and $\{x := h(a, b)\}$. When these substitutions are residualized as assignments, they evaluate the terms $g(x)$ and $h(a, b)$ only once. We refer to this phenomenon as *msg-sharing*.

Several methods more powerful than core supercompilation have been proposed that achieve superlinear speedup: distillation [10], various forms of higher-level supercompilation [9, 17], including old ideas of walk grammars by Valentin Turchin [26, 29] which are grounds for further exploration.

Nevertheless, gradual extensions of core supercompilation (which may be termed *first-level supercompilation* as they mainly operate on configurations rather than graphs,

walk grammars, *etc.*) are also possible. Here are some ideas which have been investigated or proposed, some of which have been supercompilation folklore for decades:

- Look for repeated subterms in each configuration and restructure the configuration into a let-term if found.
- *Collapsed jungle driving* [21, 22] generalizes the previous idea and formalizes it in terms of dag representation of configurations and operation of collapsing that merges topmost nodes of equal subgraphs.
- Various techniques simpler than dags, which achieve the same effects, but on fewer programs, can be added to existing supercompilers. We demonstrated one such method consisting of the addition of equality indices and the synchronization of terms after driving (Section 5). This method captures repeated subterms occurring in the list of the arguments of the same function call. It is less powerful than the use of dags, but it is much simpler because it deals with terms rather than dags plus it works with the operations of driving, generalization, homeomorphic embedding, whistles, and preserves termination proofs, while the respective operations on dags and the proofs are to be developed afresh.

Future work In preparing this paper the authors found the topic of capturing repeated function calls in first-level supercompilation was undeveloped deserving more attention. Possibly, the only systematic study in the context of supercompilation is [22]. The topic should be revisited in the context of modern research in supercompilation. Another avenue to explore is "old chestnut" problems including the naive Fibonacci function, the sum of factorials, *etc.* Solutions of their superlinear speedup by extended core supercompilation should be presented and used as test cases to develop algorithmic supercompilation strategies that reveal repeated subterms. One of the intriguing questions is the role of multi-result supercompilation [8, 14, 15, 18, 19] in these solutions, and what can be done in the single-result case. First-level methods should be compared with higher-level ones such as distillation, and their relative power and inherent limits should be studied.

Acknowledgments. The authors thank the anonymous reviewers for their constructive feedback. This work benefited greatly from discussions with the participants of the Fourth International Valentin Turchin Workshop on Metacomputation. The first author expresses his deepest thanks to Akihiko Takano for providing him with excellent working conditions at the National Institute of Informatics, Tokyo, and Masami Hagiya, Kanae Tsushima, and Zhenjiang Hu for their invaluable support.

References

1. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the ACM 24(1), 44–67 (1977)
2. Christensen, N.H., Glück, R.: Offline partial evaluation can be as accurate as online partial evaluation. ACM TOPLAS 26(1), 191–220 (2004)
3. Futamura, Y., Konishi, Z., Glück, R.: Program transformation system based on generalized partial computation. New Generation Computing 20(1), 75–99 (2002)

4. Glück, R., Klimov, A.V.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Cousot, P., et al. (eds.) Static Analysis. Proceedings. pp. 112–123. LNCS 724, Springer-Verlag (1993)
5. Glück, R., Leuschel, M.: Abstraction-based partial deduction for solving inverse problems: a transformational approach to software verification. In: Bjørner, D., Broy, M., Zamulin, A.V. (eds.) Perspectives of System Informatics. Proceedings. pp. 93–100. LNCS 1755, Springer-Verlag (2000)
6. Glück, R., Sørensen, M.H.: Partial deduction and driving are equivalent. In: Hermenegildo, M., Penjam, J. (eds.) Programming Language Implementation and Logic Programming. Proceedings. pp. 165–181. LNCS 844, Springer-Verlag (1994)
7. Glück, R., Sørensen, M.H.: A roadmap to metacomputation by supercompilation. In: Danvy, O., Glück, R., Thiemann, P. (eds.) Partial Evaluation. Proceedings. pp. 137–160. LNCS 1110, Springer-Verlag (1996)
8. Grechanik, S.A.: Overgraph representation for multi-result supercompilation. In: Klimov and Romanenko [16], pp. 48–65
9. Grechanik, S.A.: Inductive prover based on equality saturation for a lazy functional language. In: Virbitskaite, I., Voronkov, A. (eds.) Perspectives of System Informatics. Proceedings. pp. 127–141. LNCS 8974, Springer-Verlag (2015)
10. Hamilton, G.W.: Distillation: extracting the essence of programs. In: Partial Evaluation and Program Manipulation. Proceedings. pp. 61–70. ACM Press (2007)
11. Jones, N.D.: Transformation by interpreter specialization. Science of Computer Programming 52(1-3), 307–339 (2004)
12. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall (1993)
13. Jones, N.D., Hamilton, G.W.: Towards understanding superlinear speedup by distillation. In: Klimov, A.V., Romanenko, S.A. (eds.) Fourth International Valentin Turchin Workshop on Metacomputation. Proceedings. pp. 94–109. University of Pereslavl, Russia (2014), http://meta2014.pereslavl.ru/papers/2014_Jones_Hamilton__Towards_Understanding_Superlinear_Speedup_by_Distillation.pdf
14. Klimov, A.V.: Why multi-result supercompilation matters: Case study of reachability problems for transition systems. In: Klimov and Romanenko [16], pp. 91–111
15. Klimov, A.V., Klyuchnikov, I.G., Romanenko, S.A.: Automatic verification of counter systems via domain-specific multi-result supercompilation. In: Klimov and Romanenko [16], pp. 112–141
16. Klimov, A.V., Romanenko, S.A. (eds.): Third International Valentin Turchin Workshop on Metacomputation. Proceedings. University of Pereslavl, Russia (2012)
17. Klyuchnikov, I., Romanenko, S.: SPSC: a simple supercompiler in Scala. In: Bulyonkov, M.A., Glück, R. (eds.) Program Understanding. Proceedings. Ershov Institute of Informatics Systems, Russian Academy of Sciences, Novosibirsk, Russia (2009), http://spsc.googlecode.com/files/Klyuchnikov__Romanenko__SPSC_a_Simple_Supercompiler_in_Scala.pdf
18. Klyuchnikov, I.G., Romanenko, S.A.: Formalizing and implementing multi-result supercompilation. In: Klimov and Romanenko [16], pp. 142–164
19. Klyuchnikov, I.G., 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.) Perspectives of System Informatics. Proceedings. pp. 210–226. LNCS 7162, Springer-Verlag (2012)
20. Leuschel, M.: Improving homeomorphic embedding for online termination. In: Flener, P. (ed.) Logic-Based Program Synthesis and Transformation. Proceedings. pp. 199–218. LNCS 1559, Springer-Verlag (1999)
21. Secher, J.P.: Driving in the jungle. In: Danvy, O., Filinsky, A. (eds.) Programs as Data Objects. Proceedings. pp. 198–217. LNCS 2053, Springer-Verlag (2001)

22. Secher, J.P.: Driving-based Program Transformation in Theory and Practice. Ph.D. thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark (2002)
23. Sørensen, M.H.: Turchin's supercompiler revisited. DIKU Report 94/9, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark (1994)
24. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In: Lloyd, J.W. (ed.) Logic Programming: Proceedings of the 1995 International Symposium. pp. 465–479. MIT Press (1995)
25. Sørensen, M.H., Glück, R., Jones, N.D.: A positive supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)
26. Turchin, V.F.: 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. Turchin, V.F.: The concept of a supercompiler. ACM TOPLAS 8(3), 292–325 (1986)
28. Turchin, V.F.: The algorithm of generalization in the supercompiler. In: Bjørner, D., Ershov, A.P., Jones, N.D. (eds.) Partial Evaluation and Mixed Computation. pp. 531–549. North-Holland (1988)
29. Turchin, V.F.: Program transformation with metasystem transitions. Journal of Functional Programming 3(3), 283–313 (1993)
30. Turchin, V.F.: Supercompilation: techniques and results. In: Bjørner, D., Broy, M., Pottosin, I.V. (eds.) Perspectives of System Informatics. Proceedings. pp. 227–248. LNCS 1181, Springer-Verlag (1996)

Towards Unification of Supercompilation and Equality Saturation

(Extended Abstract)

Sergei Grechanik*

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

Both equality saturation and supercompilation are methods of program transformation. The idea of equality saturation [2] is to infer new equalities about program functions from the initial ones (function definitions). These new equalities can then be used to various ends: a new program may be extracted from these equalities by choosing a representative set of equalities which will constitute the new program's definitions, or properties of the original program may be proved by looking at the set of inferred equalities. Previously we have shown that equality saturation is applicable to functional languages by using transformations (inference rules) borrowed from supercompilation (driving, more precisely), more specifically we used it for the problem of proving equalities [1].

The idea of supercompilation [3] is to build a process tree representing all possible paths of program execution and then transform it into a finite graph which can be easily turned into a new program. Building a process tree is done by using a combination of driving, generalization and folding.

A question might be asked: what is the relationship between equality saturation and supercompilation? Can't they be represented as special cases of something more general? Turns out that they can be (more or less): supercompilation can be seen as the very same equality inference process that underlies equality saturation, the only difference being that in supercompilation this inference is strictly guided by heuristics, whereas in equality saturation transformations are applied simply in breadth-first order. Indeed, a process tree can be represented as a set of equalities between configurations, and process tree building operations just infer new equalities.

Let's consider a simple example of supercompilation in equality inference style. We will use only equalities of the form $f(x_1, ..., x_n) = E$ (i.e. function definitions). Equalities of more general form $E_1 = E_2$ can be represented as multiple equalities of the aforementioned form by introducing auxiliary functions. This representation is actually asymptotically more efficient when we have many equalities.

* Supported by Russian Foundation for Basic Research grant No. 16-01-00813-a and RF President grant for leading scientific schools No. NSh-4307.2012.9.

Consider the following function definitions:

$$(1)\ add(x, y) = \textbf{case}\ x\ \textbf{of}\ \{Z \to y; S(x') \to sadd(x', y)\}$$
$$(2)\ sadd(x, y) = S(add(x, y))$$
$$(3)\ f(x, y, z) = add(add(x, y), z)$$

This is a classic addition associativity example. We will supercompile the function f to get a more optimal definition of the three-number sum function. f may be considered as the root node of our process tree with the right hand side of (3) being its configuration. Rules of supercompilation prescribe performing driving first, which in this case is just unfolding of the *add* function using its definition (1):

$$(4)\ f(x, y, z) = \textbf{case}\ add(x, y)\ \textbf{of}\ \{Z \to z; S(x') \to sadd(x', z)\}$$

To continue driving we should unfold *add* in the scrutinee position using (1) and lift the inner case-of up which is done in one inference step:

$$(5)\ f(x, y, z) = \textbf{case}\ x\ \textbf{of}\ \{Z \to f_1(y, z); S(x') \to f_2(x', y, z)\}$$
$$(6)\ f_1(y, z) = \textbf{case}\ y\ \textbf{of}\ \{Z \to z; S(y') \to sadd(y', z)\}$$
$$(7)\ f_2(x', y, z) = \textbf{case}\ sadd(x', y)\ \textbf{of}\ \{Z \to z; S(x'') \to sadd(x'', z)\}$$

(Auxiliary functions f_1 and f_2 were introduced to split up the complex right hand side of (5)). Now (5) has the form of variable analysis, which means that we are done with f and can move on to the branches f_1 and f_2. f_1 is not interesting and driving it won't actually add new equalities, so let's consider only f_2. In (7) we should unfold *sadd* using (2) and reduce the case-of using the appropriate branch (again, one inference step):

$$(8)\ f_2(x', y, z) = sadd(add(x', y), z)$$

Unfolding of *sadd* leads to

$$(9)\ f_2(x', y, z) = S(f_3(x', y, z))$$
$$(10)\ f_3(x', y, z) = add(add(x', y), z)$$

But the right hand side of (10) is the same as the right hand side of (3) (which is a configuration seen earlier), so we should perform folding. In equality saturation setting this is done by removing (10) and replacing f_3 with f in every definition. Here (9) is the only one we need to modify:

$$(9')\ f_2(x', y, z) = S(f(x', y, z))$$

To get a residual program we should traverse the definitions from f choosing one definition for each function. Actually supercompilation requires us to choose

the last ones (i.e. for f we take (5), not (4) or (3)):

$$
\begin{aligned}
&(5)\ f(x,y,z) = \textbf{case}\ x\ \textbf{of}\ \{Z \to f_1(y,z); S(x') \to f_2(x',y,z)\}\\
&(9')\ f_2(x',y,z) = S(f(x',y,z))\\
&(6)\ f_1(y,z) = \textbf{case}\ y\ \textbf{of}\ \{Z \to z; S(y') \to sadd(y',z)\}\\
&(2)\ sadd(x,y) = S(add(x,y))\\
&(1)\ add(x,y) = \textbf{case}\ x\ \textbf{of}\ \{Z \to y; S(x') \to sadd(x',y)\}
\end{aligned}
$$

This example shows that performing supercompilation as process of equality inference is possible in principle.

Equality saturation is guaranteed to eventually perform the same steps as supercompilation since it works in breadth-first manner. This actually makes equality saturation more powerful in theory. Indeed, it already subsumes multi-result supercompilation since it doesn't restrict application of different transformations to either driving or generalization. If we add merging by bisimulation, which is essentially checking equivalence of two expression by residualizing them, then we also get higher-level supercompilation. However, this power comes at a price: without heuristic guidance we risk to be hit by combinatorial explosion. And this actually happens in reality: our experimental prover was unable to pass the KMP-test because of this, and what is worse, generalizing rules had to be disabled which left our prover only with the simplest form of generalization, namely removal of the outermost function call. That's why it would be interesting to make a hybrid between supercompilation and pure equality saturation that would restrict transformation application, but not too much.

Currently our equality saturating prover has an experimental mode that performs driving up to certain depth before performing ordinary equality saturation. It allows our prover to pass the KMP-test as well as a couple of similar examples, but results in regression on some other examples. Although this mode shows that such a combination is possible, it is far from a fully fledged supercompilation/equality saturation hybrid since it lacks most of supercompilation heuristics and works in a sequential way (first driving, and only then breath-first saturation). Therefore, two directions of future research may be named here:

- Developing heuristics to control generalization. Using generalizing transformations without restriction leads to combinatorial explosion. Direct application of mgu from supercompilation doesn't seem to be a good solution because there may be too many pairs of terms due to multiresultness.
- Developing heuristics to control overall rewriting, mainly depth of driving. In supercompilation whistles are used for this purpose, but traditional homeomorphic embedding whistles are also hard to implement in multi-result setting of equality saturation.

Since the main difficulty in using traditional heuristics seems to be with multi-result nature of equality saturation, it is entirely possible that completely new methods should be developed.

References

1. S. Grechanik. Inductive prover based on equality saturation. In A. Klimov and S. Romanenko, editors, *Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation*, Pereslavl-Zalessky, Russia, July 2014. Pereslavl Zalessky: Publishing House "University of Pereslavl".
2. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.
3. V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

On Programming and Biomolecular Computation

Neil D. Jones

DIKU, University of Copenhagen (Denmark)

Abstract. In spite of widespread discussion about connections between biology and computation, one question seems notable by its absence: **Where are the programs?** We propose a model of computation that is at the same time biologically plausible: its functioning is defined by a relatively small set of chemical-like reaction rules; programmable (by programs reminiscent of low-level computer machine code); uniform: new "hardware" is not needed to solve new problems; universal: stored-program: data are the same as programs, and so are executable, compilable and interpretable. The model is strongly Turing complete: a universal algorithm exists, able to execute any program, and not asymptotically inefficient.

The model has been designed and implemented (for now in silico on a conventional computer). We hope to open new perspectives on just how to specify computation at the biological level.; and to provide a top-down approach to biomolecular computation.

(Joint work with Jakob Grue Simonsen, Lars Hartmann, Søren Vrist.)

While this talk is not directly about metacomputation, programming languages will be visible in the model and the way it is developed. It is a modest updating of a META 2010 talk *Programming in Biomolecular Computation.*

The topic may be interesting to META; and it is interesting to me since the feedback received so far has mainly been from people responding to the biological-modeling aspect (e.g., questions such as "are cells deterministic enough", "how would you implement it on a Petri dish", etc.).

This talk has much to do with interpreters and (very) finite-state program execution mechanisms, hopefully leading to some connections between

- Program specialisation and specialisation of biological cells (e.g., zygotes, embryo,...)
- Self-application as in Futamura compared with biological self-repoduction

The topic is a bit wild, and as yet not many people have picked up on the ideas. However I think there is a potential, and it will be will be interesting to hear the metacomputation community's viewpoints .

References

1. L. Hartmann, N.D. Jones, J.G. Simonsen, and S.B. Vrist. Programming in biomolecular computation: Programs, self-interpretation and visualisation. *Scientific Annals of Computer Science*, 21(1):73–106, 2011.
2. Lars Hartmann, Neil D. Jones, and Jakob Grue Simonsen. Programming in biomolecular computation. *Electr. Notes Theor. Comput. Sci.*, 268:97–114, 2010.
3. Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen, and Søren Bjerregaard Vrist. Computational biology: A programming perspective. In *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 403–433, 2011.
4. Neil D. Jones and Jakob Grue Simonsen. Programs = data = first-class citizens in a computational world. *Philosophical Transactions of the Royal Society A*, pages 3305–3318, 2012.

Distilling New Data Types

Venkatesh Kannan and G. W. Hamilton

School of Computing, Dublin City University, Ireland
{vkannan, hamilton}@computing.dcu.ie

Abstract. Program transformation techniques are commonly used to improve the efficiency of programs. While many transformation techniques aim to remove inefficiencies in the algorithms used in a program, another source of inefficiency is the use of inappropriate datatypes whose structures do not match the algorithmic structure of the program. This mismatch will potentially result in inefficient consumption of the input by the program. Previously, Mogensen has shown how techniques similar to those used in supercompilation can be used to transform datatypes, but this was not fully automatic. In this paper, we present a fully automatic datatype transformation technique which can be applied in conjunction with distillation. The objective of the datatype transformation is to transform the original datatypes in a program so that the resulting structure matches the algorithmic structure of the distilled program. Consequently, the resulting transformed program potentially uses less pattern matching and as a result is more efficient than the original program.

1 Introduction

Fold/unfold program transformation has been used to obtain more efficient versions of programs. One of the primary improvements achieved by such transformation techniques is through the elimination of intermediate data structures that are used in a given program, referred to as *fusion* – combining multiple functions in a program into a single function thereby eliminating the intermediate data structure used between them. Transformation techniques such as supercompilation [9, 10] and distillation [2] are based on the unfold/fold transformation framework and achieve such improvements. In particular, the distillation transformation can potentially result in super-linear speedup of the distilled program.

While unfold/fold program transformation redefines functions for optimisation, the data types of the programs produced remain unaltered. For instance, we observe that the programs produced by the distillation transformation are still defined over the original data types. Thus, another source of inefficiency in a program is the potential mismatch of the structures of the data types in comparison to the algorithmic structure of the program [5].

For instance, consider the simple program defined in Example 1 which reduces a given list by computing the sum of neighbouring pairs of elements in the list.

Example 1 (Reduce Neighbouring Pairs).
$reducePairs\ ::\ [Int] \rightarrow [Int]$

$reducePairs\ xs$
where
$reducePairs\ []\ \ \ \ \ \ \ \ \ \ \ \ \ \ = []$
$reducePairs\ (x : [])\ \ \ \ \ \ \ \ \ = x : []$
$reducePairs\ (x_1 : x_2 : xs)\ = (x_1 + x_2) : (reducePairs\ xs)$

Here, we observe that in order to pattern-match a non-empty list, *reducePairs* checks if the tail is non-empty (in which case the second pattern $(x : [])$ is excluded), and then the tail is matched again in the third pattern $(x_1 : x_2 : xs)$. Also, the third pattern is nested to obtain the first two elements x_1 and x_2 in the list. While this pattern is used to obtain the elements that are used in the function body, we observe that the structure of the pattern-matching performed is inefficient and does not match the structure of the *reducePairs* function definition. It desirable to have the input argument structured in such a way that the elements x_1 and x_2 are obtained using a single pattern-match and redundant pattern-matchings are avoided. One such definition of the *reducePairs* function is presented in Example 2 on a new data type $T_{reducePairs}$.

Example 2 (Reduce Neighbouring Pairs – Desired Program).
data $T_{reducePairs}\ ::=\ c_1 \mid c_2\ Int \mid c_3\ Int\ Int\ T_{reducePairs}$

$reducePairs\ xs$
where
$reducePairs\ c_1\ \ \ \ \ \ \ \ \ \ \ \ \ \ = []$
$reducePairs\ (c_2\ x)\ \ \ \ \ \ \ \ \ = x : []$
$reducePairs\ (c_3\ x_1\ x_2\ xs)\ = (x_1 + x_2) : (reducePairs\ xs)$

In [6], Mogensen proposed one of the methods to address these issues by creating data types that suit the structure of programs based on the supercompilation transformation [10, 11]. The resulting transformed programs use fewer constructor applications and pattern-matchings. However, the transformation remains to be automated because functions that allow conversion between the original and new data types were not provided.

In this paper, we present a data type transformation technique to automatically define a new data type by transforming the original data types of a program. The new transformed data type is defined in such a way that its structure matches the algorithmic structure of the program. As a result, the transformed input argument is consumed in a more efficient fashion by the transformed program.

The proposed transformation is performed using the following two steps:

1. Apply the distillation transformation on a given program to obtain the distilled program. (Section 3)
2. Apply the proposed data type transformation on a distilled program to obtain the transformed program. (Section 4)

In Section 5, we demonstrate the proposed transformation with examples and present the results of evaluating the transformed programs. In Section 6, we discuss the merits and applications of the proposed transformation along with related work.

2 Language

The higher-order functional language used in this work is shown in Definition 1.

Definition 1 (Language Grammar).

$$\begin{array}{lll}
\textbf{data}\ T\ \alpha_1 \ldots \alpha_M\ ::=\ c_1\ t_1^1 \ldots t_N^1 \mid \ldots \mid c_K\ t_1^K \ldots t_N^K & & \textit{Type Declaration} \\
t\ ::=\ \alpha_m \mid T\ t_1 \ldots t_M & & \textit{Type Component} \\
\\
e ::= x & & \textit{Variable} \\
\quad \mid\ c\ e_1 \ldots e_N & & \textit{Constructor Application} \\
\quad \mid\ e_0 & & \textit{Function Definition} \\
\qquad \textbf{where} & & \\
\qquad f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = e_1 & & \\
\qquad \vdots & & \\
\qquad f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = e_K & & \\
\quad \mid\ f & & \textit{Function Call} \\
\quad \mid\ e_0\ e_1 & & \textit{Application} \\
\quad \mid\ \textbf{let}\ x_1 = e_1\ \ldots\ x_N = e_N\ \textbf{in}\ e_0 & & \textbf{let}\textit{–Expression} \\
\quad \mid\ \lambda x.e & & \lambda\textit{–Abstraction} \\
\\
p ::= x \mid c\ p_1 \ldots p_N & & \textit{Pattern}
\end{array}$$

A program can contain data type declarations of the form shown in Definition 1. Here, T is the name of the data type, which can be polymorphic, with type parameters $\alpha_1, \ldots, \alpha_M$. A data constructor c_k may have zero or more components, each of which may be a type parameter or a type application. An expression e of type T is denoted by $e :: T$.

A program in this language can also contain an expression which can be a variable, constructor application, function definition, function call, application, **let**-expression or λ-abstraction. Variables introduced in a function definition, **let**-expression or λ-abstraction are *bound*, while all other variables are *free*. The free variables in an expression e are denoted by $fv(e)$. Each constructor has a fixed arity. In an expression $c\ e_1 \ldots e_N$, N must be equal to the arity of the constructor c. For ease of presentation, patterns in function definition headers are grouped into two – $p_1^k \ldots p_M^k$ are inputs that are pattern-matched, and $x_{(M+1)}^k \ldots x_N^k$ are inputs that are not pattern-matched. The series of patterns $p_1^k \ldots p_M^k$ in a function definition must be non-overlapping and exhaustive. We use [] and (:) as shorthand notations for the *Nil* and *Cons* constructors of a *cons*-list.

Definition 2 (Context). A context E is an expression with *hole*s in place of sub-expressions. $E[e_1, \ldots, e_N]$ is the expression obtained by filling holes in context E with the expressions $e_1, \ldots, e_N$.

3 Distillation

Given a program in the language from Definition 1, *distillation* [2] is a technique that transforms the program to remove intermediate data structures and yields a *distilled program*. It is an unfold/fold-based transformation that makes use of well-known transformation steps – unfold, generalise and fold [7] – and can potentially provide super-linear speedups to programs.

The syntax of a distilled program $de^{\{\}}$ is shown in Definition 3. Here, ρ is the set of variables introduced by **let**–expressions that are created during generalisation. These bound variables of **let**-expressions are not decomposed by pattern-matching in a distilled program. Consequently, $de^{\{\}}$ is an expression that has fewer intermediate data structures.

Definition 3 (Distilled Form Grammar).

$$
\begin{array}{rcll}
de^{\rho} & ::= & x\ de_1^{\rho} \ldots de_N^{\rho} & \textit{Variable Application} \\
& | & c\ de_1^{\rho} \ldots de_N^{\rho} & \textit{Constructor Application} \\
& | & de_0^{\rho} & \textit{Function Definition} \\
& & \textbf{where} & \\
& & f\ p_1^1 \ldots p_M^1\ x_{(M+1)}^1 \ldots x_N^1 = de_1^{\rho} & \\
& & \vdots & \\
& & f\ p_1^K \ldots p_M^K\ x_{(M+1)}^K \ldots x_N^K = de_K^{\rho} & \\
& | & f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N & \textit{Function Application} \\
& & s.t.\ \forall x \in \{x_1, \ldots, x_M\} \cdot x \notin \rho & \\
& | & \textbf{let}\ x_1 = de_1^{\rho}\ \ldots\ x_N = de_N^{\rho}\ \textbf{in}\ de_0^{\rho\ \cup\ \{x_1,\ldots,x_N\}} & \textbf{let}\textit{–Expression} \\
& | & \lambda x.de^{\rho} & \lambda\textit{–Abstraction} \\
p & ::= & x \mid c\ p_1 \ldots p_N & \textit{Pattern}
\end{array}
$$

4 Data Type Transformation

A program in distilled form is still defined over the original program data types. In order to transform these data types into a structure that reflects the structure of the distilled program, we apply the data type transformation proposed in this section on the distilled program. In the transformation, we combine the pattern-matched arguments of each function f in the distilled program into a single argument which is of a new data type T_f and whose structure reflects the algorithmic structure of function f.

Consider a function f, with arguments $x_1, \ldots, x_M, x_{(M+1)}, \ldots, x_N$, of the form shown in Definition 4 in a distilled program. Here, a function body e_k corresponding to function header $f\ p_1^k \ldots p_M^k\ x_{(M+1)}^k \ldots x_N^k$ in the definition of f may contain zero or more recursive calls to function f.

Definition 4 (General Form of Function in Distilled Program).

$$
\begin{array}{l}
f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N \\
\textbf{\textit{where}} \\
f\ p_1^1 \ldots p_M^1\ x_{(M+1)} \ldots x_N \ = \ e_1 \\
\vdots \qquad\qquad\qquad\qquad\qquad\quad \vdots \\
f\ p_1^K \ldots p_M^K\ x_{(M+1)} \ldots x_N \ = \ e_K
\end{array}
$$

The three steps to transform the pattern-matched arguments of function f are as follows:

1. **Declare a new data type for the transformed argument:**
   First, we declare a new data type T_f for the new transformed argument. This new data type corresponds to the data types of the original pattern-matched arguments of function f. The definition of the new data type T_f is shown in Definition 5.

   Definition 5 (New Data Type T_f).

   $$
   \begin{array}{l}
   \textbf{data}\ T_f\ \alpha_1 \ldots \alpha_G \ ::= \ c_1\ T_1^1 \ldots T_L^1\ (T_f\ \alpha_1 \ldots \alpha_G)_1^1 \ldots (T_f\ \alpha_1 \ldots \alpha_G)_J^1 \\
   \qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\
   \qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \quad c_K\ T_1^K \ldots T_L^K\ (T_f\ \alpha_1 \ldots \alpha_G)_1^K \ldots (T_f\ \alpha_1 \ldots \alpha_G)_J^K
   \end{array}
   $$

   where

   $\alpha_1, \ldots, \alpha_G$ *are type parameters of the data types of pattern-matched arguments.*

   $\forall k \in \{1, \ldots, K\} \cdot$

   c_k *is a fresh constructor for* T_f *corresponding to* $p_1^k \ldots p_M^k$ *of the pattern-matched arguments.*

   $$
   f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots\ , \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}
   $$

   $$
   \left\{(z_1 :: T_1^k), \ldots, (z_L :: T_L^k)\right\} = fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\}
   $$

   Here, a new constructor c_k of the type T_f is created for each set $p_1^k \ldots p_M^k$ of the pattern-matched inputs $x_1 \ldots x_M$ of function f that are encoded. As stated above, our objective is to transform the arguments of function f into a new type whose structure reflects the recursive structure of f. To achieve this, the components bound by constructor c_k correspond to the variables in $p_1^k \ldots p_M^k$ that occur in the context E_k and the transformed arguments of the recursive calls to function f.
2. **Define a function to build the transformed argument:**
   Given a function f of the form shown in Definition 4, we define a function $encode_f$, as shown in Definition 6, to build the transformed argument for function f.

Definition 6 (Definition of Function *encode*$_f$).

$encode_f\ x_1 \ldots x_M$

where

$$\begin{array}{l} encode_f\ p_1^1 \ldots p_M^1 \ = \ e'_1 \\ \vdots \qquad\qquad\qquad\quad \vdots \\ encode_f\ p_1^K \ldots p_M^K \ = \ e'_K \end{array}$$

where

$\forall k \in \{1, \ldots, K\}\cdot$

$e'_k = c_k\ z_1^k \ldots z_L^k\ (encode_f\ x_1^1 \ldots x_M^1) \ldots (encode_f\ x_1^J \ldots x_M^J)$

$\{z_1^k, \ldots, z_L^k\} = fv(E_k) \setminus \{x_{(M+1)}, \ldots, x_N\}$

$$f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots\ , \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$$

Here, the original arguments $x_1 \ldots x_M$ of function f are pattern-matched and consumed by $encode_f$ in the same way as in the definition of f. For each pattern $p_1^k \ldots p_M^k$ of the arguments $x_1 \ldots x_M$, function $encode_f$ uses the corresponding constructor c_k whose components are the variables $z_1^k, \ldots, z_L^k$ in $p_1^k \ldots p_M^k$ that occur in the context E_k and the transformed arguments of the recursive calls to function f.

3. **Transform the distilled program :**
After creating the transformed data type T_f and the $encode_f$ function for each function f, we transform the distilled program as shown in Definition 7 by defining a function f', which operates over the transformed argument, corresponding to function f.

Definition 7 (Definition of Transformed Function Over Transformed Argument).

$f'\ x\ x_{(M+1)} \ldots x_N$

where

$$\begin{array}{l} f'\ \left(c_1\ z_1^1 \ldots z_L^1\ x_1^1 \ldots x_1^J\right)\ x_{(M+1)} \ldots x_N \quad = e'_1 \\ \vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \vdots \\ f'\ \left(c_K\ z_1^K \ldots z_L^K\ x_K^1 \ldots x_K^J\right)\ x_{(M+1)} \ldots x_N = e'_K \end{array}$$

where

$\forall k \in \{1, \ldots, K\}\cdot$

$$e'_k \ = \ E_k \left[f'\ x_k^1\ x_{(M+1)}^1 \ldots x_N^1,\ \ldots,\ f'\ x_k^J\ x_{(M+1)}^J \ldots x_N^J\right]$$

$$f\ p_1^k \ldots p_M^k\ x_{(M+1)} \ldots x_N = E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1, \\ \ldots\ , \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$$

The two steps to transform function f into function f' that operates over the transformed argument are:

(a) In each function definition header of f, replace the original pattern-matched arguments with the corresponding pattern of their transformed data type T_f.
For instance, a function header $f\ p_1 \ldots p_M\ x_{(M+1)} \ldots x_N$ is transformed to the header $f'\ p\ x_{(M+1)} \ldots x_N$, where p is the pattern created by $encode_f$ corresponding to the original pattern-matched arguments $p_1, \ldots, p_M$.

(b) In each call to function f, replace the original arguments with their corresponding transformed argument.
For instance, a call $f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N$ is transformed to the function call $f'\ x\ x_{(M+1)} \ldots x_N$, where x is the transformed argument corresponding to the original arguments $x_1, \ldots, x_M$.

4.1 Correctness

The correctness of the proposed transformation can be established by proving that the result computed by each function f in the distilled program is the same as the result computed by the corresponding function f' in the transformed program. That is,

$$\big(f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N\big) = \big(f'\ x\ x_{(M+1)} \ldots x_N\big)$$
where $x = encode_f\ x_1 \ldots x_M$

Proof:
The proof is by structural induction over the transformed data type T_f.

Base Case:
For the transformed argument $x_k = c_k\ z_1^k \ldots z_L^k$ computed by $encode_f\ p_1^k \ldots p_M^k$,

1. By Definition 4, L.H.S. evaluates to e_k.
2. By Definition 7, R.H.S. evaluates to e_k.

Inductive Case:
For the transformed argument $x_k = c_k\ z_1^k \ldots z_L^k\ x^{1_k} \ldots x^{J_k}$ which is computed by $encode_f\ p_1^k \ldots p_M^k$,

1. By Definition 4, L.H.S. evaluates to $E_k \begin{bmatrix} f\ x_1^1 \ldots x_M^1\ x_{(M+1)}^1 \ldots x_N^1,\ \ldots\ , \\ f\ x_1^J \ldots x_M^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$.
2. By Definition 7, R.H.S. evaluates to $E_k \begin{bmatrix} f'\ x_k^1\ x_{(M+1)}^1 \ldots x_N^1,\ \ldots\ , \\ f'\ x_k^J\ x_{(M+1)}^J \ldots x_N^J \end{bmatrix}$.
3. By inductive hypothesis, $\big(f\ x_1 \ldots x_M\ x_{(M+1)} \ldots x_N\big) = \big(f'\ x\ x_{(M+1)} \ldots x_N\big)$.

□

5 Examples

We demonstrate and evaluate the data type transformation presented in this paper using two simple examples, including the program introduced in Example 1, which are discussed in this section.

5.1 Reduce Neighbouring Pairs

The *reducePairs* program presented in Example 1 does not use any intermediate data structures. Consequently, the result of applying the distillation transformation yields the same program. Following this, Example 3 presents the transformed data type ($T_{reducePairs}$), the transformation function ($encode_{reducePairs}$) and the transformed program ($reducePairs'$) obtained for the *reducePairs* program in Example 1.

Example 3 (Reduce Neighbouring Pairs – Transformed Program).

$$
\begin{array}{l}
\textbf{data}\ T_{reducePairs}\ a\ ::=\ c_1 \\
\qquad\qquad\qquad\quad \mid\ c_2\ a \\
\qquad\qquad\qquad\quad \mid\ c_3\ a\ a\ (T_{reducePairs}\ a)
\end{array}
$$

$$
\begin{array}{lcl}
encode_{reducePairs}\ [] & = & c_1 \\
encode_{reducePairs}\ (x : []) & = & c_2\ x \\
encode_{reducePairs}\ (x_1 : x_2 : xs) & = & c_3\ x_1\ x_2\ (encode_{reducePairs}\ xs)
\end{array}
$$

$$
\begin{array}{l}
reducePairs'\ xs \\
\textbf{where} \\
\begin{array}{lcl}
reducePairs'\ c_1 & = & [] \\
reducePairs'\ (c_2\ x) & = & x : [] \\
reducePairs'\ (c_3\ x_1\ x_2\ xs) & = & (x_1 + x_2) : (reducePairs'\ xs)
\end{array}
\end{array}
$$

5.2 Reduce Trees

To demonstrate our data type transformation, we present another program in Example 4 that performs a reduction over a list of binary trees. Since this definition does not contain intermediate data structures, the result of applying the distillation transformation is the same program.

Example 4 (Reduce Trees – Original/Distilled Program).

$$
\begin{array}{l}
\textbf{data}\ BTree\ a\ ::=\ L \\
\qquad\qquad\quad \mid\ B\ a\ [BTree\ a]\ [BTree\ a]
\end{array}
$$

$$
\begin{array}{l}
reduceTrees\ ::\ [BTree\ Int] \rightarrow Int \\
reduceTrees\ ts \\
\textbf{where} \\
\begin{array}{lcl}
reduceTrees\ [] & = & 0 \\
reduceTrees\ (L : xs) & = & reduceTrees\ xs \\
reduceTrees\ \big((B\ x\ lts\ rts) : xs\big) & = & x + (reduceTrees\ lts) \\
 & & \quad +(reduceTrees\ rts) + (reduceTrees\ xs)
\end{array}
\end{array}
$$

Example 5 presents the transformed data type ($T_{reduceTrees}$), the transformation function ($encode_{reduceTrees}$) and the transformed program ($reduceTrees'$) obtained for the distilled *reduceTrees* program using the proposed transformation.

Example 5 (Reduce Trees – Transformed Program).

$$
\begin{array}{l}
\textbf{data}\ T_{reduceTrees}\ a\ ::= \ c_1 \\
\qquad\qquad\qquad\qquad\quad \mid \ c_2\ (T_{reduceTrees}\ a) \\
\qquad\qquad\qquad\qquad\quad \mid \ c_3\ a\ (T_{reduceTrees}\ a)\ (T_{reduceTrees}\ a)\ (T_{reduceTrees}\ a)
\end{array}
$$

$$
\begin{array}{lcl}
encode_{reduceTrees}\ [] & = & c_1 \\
encode_{reduceTrees}\ (L : xs) & = & c_2\ (encode_{reduceTrees}\ xs) \\
encode_{reduceTrees}\ \big((B\ x\ lts\ rts) : xs\big) & = & c_3\ x\ (encode_{reduceTrees}\ lts) \\
 & & \qquad (encode_{reduceTrees}\ rts) \\
 & & \qquad (encode_{reduceTrees}\ xs)
\end{array}
$$

$$
\begin{array}{l}
reduceTrees'\ ts \\
\textbf{where} \\
\begin{array}{lcl}
reduceTrees'\ c_1 & = & 0 \\
reduceTrees'\ (c_2\ xs) & = & reduceTrees'\ xs \\
reduceTrees'\ (c_3\ x\ lts\ rts\ xs) & = & x + (reduceTrees'\ lts) \\
 & & \quad + (reduceTrees'\ rts) + (reduceTrees'\ xs)
\end{array}
\end{array}
$$

5.3 Evaluation

For the two example programs presented in this section, we compare the execution times of the transformed functions *reducePairs′* and *reduceTrees′* against those of their original versions *reducePairs* and *reduceTrees*, respectively, for different input sizes. The resulting speedups achieved by these transformed programs are illustrated in Figure 1. Here, the input sizes for the *reduceTrees* program are the number of values that are present in the input tree that is reduced.

We observe that, as a result of the reduced pattern-matchings performed in the transformed programs, the transformed functions consume the transformed arguments more efficiently resulting in a speedup of `1.26x` – `1.67x` for the two examples evaluated in this section.

Additionally, Figure 2 illustrates the cost of transforming the arguments (using the $encode_f$ functions) in comparison with the total execution time of the transformed program.

We observe that the cost of transforming the arguments is non-trivial. However, given the relation between the original data type and the new transformed data type, which is defined by the $encode_f$ function, the user can benefit by producing the inputs in the proposed transformed data type and by using the efficient transformed program.

6 Conclusion

6.1 Summary

The data type transformation presented in this paper allows us to modify the program data types into a structure that reflects the structure of the program in distilled form. This is achieved by combining the original pattern-matched arguments of each function in the distilled program. The transformation combines

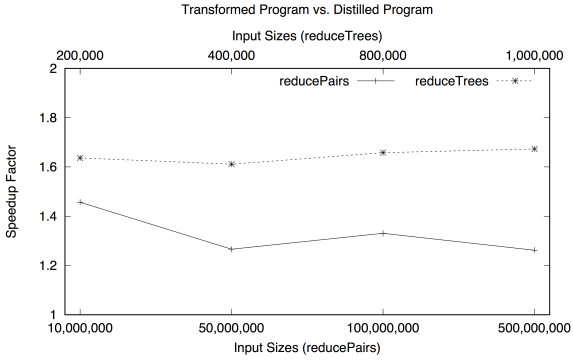


Fig. 1. Speedups of Transformed Programs vs. Distilled Programs

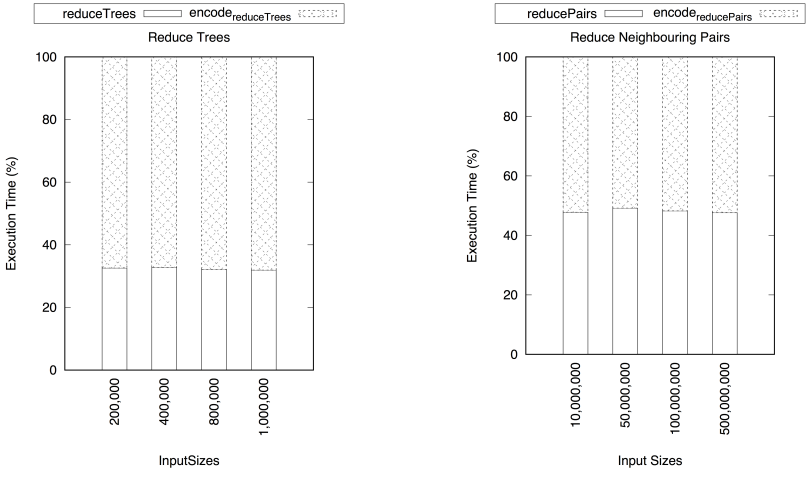


Fig. 2. Cost Centre of Transformed Programs

groups of patterns that are matched with the arguments into a single pattern for the transformed argument. By using the transformation function ($encode_f$) that specifies the correspondence between the original data types and the trans-

formed data type, the user can produce a transformed argument which requires less pattern-matching. Consequently, the transformed program can potentially consume the input data in an optimised fashion.

Furthermore, this data type transformation can also be used to facilitate automatic parallelisation of a given program. By defining algorithmic skeletons that operate over the newly defined transformed data type, we can identify instances of the skeletons – polytypic [4] and list-based [3] – in the transformed program. Following this, we can use efficient parallel implementations of the skeletons to execute the transformed program on parallel hardware.

6.2 Related Work

The importance of such data type transformation methods has been discussed in other works such as [1,5]. Creating specialised data types that suit the structure of a program can provide flexibility to statically typed languages that is similar to dynamically typed languages.

Mogensen presented one of the initial ideas in [5] to address data type transformation using constructor specialisation. This method improves the quality of the transformed programs (such as compiled programs) by inventing new data types based on the pattern-matchings performed on the original data types. It is explained that such data type transformation approaches can impact the performance of a program that uses limited data types to encode a larger family of data structures as required by the program.

To improve on Mogensen's work in [5], Dussart et al. proposed a polyvariant constructor specialisation in [1]. The authors highlight that the earlier work by Mogensen was monovariant since each data type, irrespective of how it is dynamically used for pattern-matching in different parts of a program, is statically analysed and transformed. This monovariant design potentially produces dead code in the transformed programs. Dussart et al. improved this by presenting a polyvariant version where a data type is transformed by specialising it based on the context in which it is used. This is achieved in three steps: (1) compute properties for each pattern-matching expression in the program based on its context, (2) specialise the pattern-matching expression using these properties, and (3) generate new data type definitions using the specialisations performed.

More recently, in [6], Mogensen presents supercompilation for data types. Similar to the unfold, fold and special-casing steps used in the supercompilation transformation, the author presents a technique for supercompiling data types using the three steps designed for data types. This technique combines groups of constructor applications in a given program into a single constructor application of a new data type that is created analogous to how supercompilation combines groups of function calls into a single function call. As a result, the number of constructor applications and pattern-matchings in the transformed program are fewer compared to the regular supercompiled programs. What remains to be done in this technique is the design of functions that allow automatic conversion between the original and supercompiled data types. We address this aspect in

our proposed transformation technique by providing automatic steps to declare the transformed data type and to define the transformation function.

In [8], Simon Jones presents a method to achieve the same objective of matching the data types used by a program and the definition of the program. The main difference to this approach is that their transformation specialises each recursive function according to the structure of its arguments. This is achieved by creating a specialised version of the function for each distinct pattern. Following this, the calls to the function are replaced with calls to the appropriate specialised versions. To illustrate this transformation, consider the following definition of function *last*, where the tail of the input list is redundantly checked by the patterns $(x : [])$ and $(x : xs)$.

$$\begin{array}{lcl} last\ [] & = & error\ \text{“}last\text{”} \\ last\ (x : []) & = & x \\ last\ (x : xs) & = & last\ xs \end{array}$$

Such a definition is transformed by creating a specialised version of the *last* function based on the patterns for the list tail, resulting in the following definition for the *last* function which avoids redundant pattern-matching.

$$\begin{array}{lcl} last\ [] & = & error\ \text{“}last\text{”} \\ last\ (x : xs) & = & last'\ x\ xs \\ & & \textbf{where} \\ & & last'\ x\ [] = x \\ & & last'\ x\ (y : ys) = last'\ y\ ys \end{array}$$

This transformation was implemented as a part of the Glasgow Haskell Compiler for evaluation and results in an average run-time improvement of 10%.

Acknowledgement

This work was supported, in part, by the Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Research Centre (www.lero.ie).

References

[1] Dirk Dussart, Eddy Bevers, and Karel De Vlaminck. Polyvariant constructor specialisation. *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1995.

[2] 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.

[3] Venkatesh Kannan and G. W. Hamilton. Program Transformation to Identify List-Based Parallel Skeletons. *4th International Workshop on Verification and Program Transformation (VPT)*, 2016.

[4] Venkatesh Kannan and G. W. Hamilton. Program Transformation to Identify Parallel Skeletons. *24th International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2016.
[5] Torben Æ. Mogensen. Constructor specialization. *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 22–32, 1993.
[6] Torben Æ. Mogensen. Supercompilation for datatypes. *Perspectives of System Informatics (PSI)*, 8974:232–247, 2014.
[7] A. Pettorossi, M. Proietti, and R. Dicembre. Rules And Strategies For Transforming Functional And Logic Programs. *ACM Computing Surveys*, 1996.
[8] Simon Peyton Jones. Call-pattern specialisation for haskell programs. *SIGPLAN Not.*, 42(9):327–337, 2007.
[9] M. H. Sørensen, R. Glück, and N. D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 1996.
[10] Valentin F. Turchin. A Supercompiler System Based on the Language Refal. *SIGPLAN Notices*, 1979.
[11] Valentin F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems*, 1986.

Enhanced PCB-Based Slicing

Husni Khanfar and Björn Lisper

School of Innovation, Design, and Engineering, Mälardalen University,
SE-721 23 Västerås, Sweden
{husni.khanfar,bjorn.lisper@mdh.se}@mdh.se

Abstract. Program slicing can be used to improve software reliability. It enables identification and checking of critical points by removing program parts that do not influence them. The standard slicing method computes the dependencies of an entire program as a prerequisite to the slicing. Demand-driven techniques such as our work Predicate Control Block (PCB)-based slicing computes only the dependencies affecting critical points. This improves the performance for single slices.
This paper extends PCB-based slicing to efficiently compute several slices from the same code. This is done by storing the computed data dependencies in a form of graph to reuse them between individual slices. We also show how PCB-based slicing can be done interprocedurally in a demand-driven fashion. Moreover, we describe a filtering technique that reduces the exploration of irrelevant paths. These two improvements enhance the algorithm performance, which we show using synthetic benchmarks.

Keywords: Program Slicing, Reliable Software, Predicate Control Block

1 Introduction

Backward program slicing extracts the set of statements (so-called "slice") that may affect a slicing criterion. A slicing criterion refers to the value of a particular variable at a program location. In considering critical points in reliable systems as slicing criteria, backward slicing enables us to study many aspects related to those points. For a given slicing criterion, the slicing computes the statements, inputs and conditions that possibly affect the slicing criterion. The effect can appear by two ways: *Control Dependence*, which occurs where a predicate controls the possible execution of statements and *Data Dependence* which occurs when a variable updated at a statement is used in another.

Software programs are getting more complex and it is important for those programs to be reliable. Software reliability refers to the continuity of correct service [14]. To deliver a correct service, the cornerstone is in being fault-free. To find faults, the static analysis methods such as path simulation or the verification methods such as model checking are used. Larson states in [15] that "A major problem with path-based defect detection systems is path explosion". Model checking performs automatic exhaustive testing and Choi et al states in [3] that it might suffer from state-space explosion. The fact that program slicing reduces

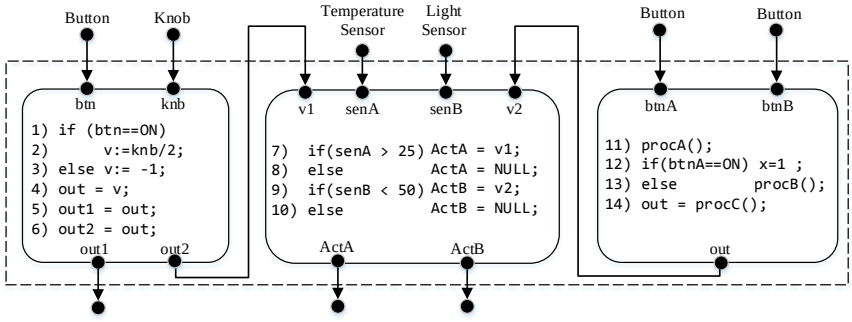


Fig. 1: Simple Control Unit

the program size is important because it can help alleviate the path and state-space explosion.

Program slicing can also be used to check for possible dependencies. consider the simple control unit in Fig. 1. Assume that `ActA` is only to be controlled by `v1` and `senA`, the dependence on any other input being considered a fault. For studying this system with respect to `ActA`, we choose `ActA` as a slicing criterion. The slice taken will show that `senA` may be dependent on v1 and `senA` but is surely not dependent on `senB` or `v2`, ensuring that the system is correct with respect to input dependencies.

The unit in Fig. 1 has six inputs. If we suppose that each button has 2 states, each knob and sensor has 10 states, then we need 32K states to cover all the input combinations. If the aim is to study `ActA`, then it is enough to generate the states relevant to `ActA`. These relevant states are computed from the inputs that `ActA` is dependent on. Notice that the slice of the control unit with respect to `ActA` consists of the program lines: {1,2,3,4,6,7,8} and the program inputs: {`buttonA,Knob1,v1,senA`}. In this case the number of states are: 400. Apparently, the state-space is reduced significantly.

The most common slicing technique is designed on the program representation Program Dependence Graph (PDG) [11]. The PDG consists of nodes and edges which represent statements and direct data and control dependencies respectively. To construct a PDG, a deep comprehensive analysis is performed to compute all dependencies in prior of the slicing. This analysis is reported by many authors [3, 4, 6] as very time and space consuming.

Demand-driven approaches compute data and control dependencies on demand during the slicing operation and not in prior. This gets rid of computing unrelated dependencies which cause unnecessary computations. Our previous work in Predicate Control Block(PCB)-Based slicing [1] is an example of a demand-driven approach. PCB is a basis of program representations that models well-structured, inter-procedural and jump-free programs. This new slicing method is designed for applications that are compliant with MISRA-C and SPARK, which are software development guidelines for writing reliable and safety-critical applications in C and Ada languages respectively.

Demand-driven slicing approaches focus on eliminating unrelated dependencies, but these dependencies are not the only source of unnecessary computations. In computing data dependencies relevant to a particular program point p, all backward or forward paths from p have to be explored. Most of these paths do not include any relevant information. e.g. in Fig. 1 for ActA point, we explore (9) and (10) and this is wasteful.

Sometimes, many critical points in a program need studying. Thus, a slice for each point has to be produced. In demand-driven slicing approaches such as PCB-based approach, when a source code is sliced many times in accordance to many individual slicing criteria, the data dependencies involved in each slice have to be computed from scratch even though some or all of them were already computed for a previous slice. These computations are unnecessary because they already were done before. e.g. in our example, (4) is dependent on (2) and (3) which are dependent on (1). These dependencies affect `out1` and `out2`. If `out1` and `out2` are two individual points of interest, then we must compute these dependencies for each of them.

The contributions of this paper are:

1. Extending the PCB-based algorithm to reuse information from previous slicing, making it more efficient when slicing the same code several times for individual slicing criteria.
2. Extending the PCB-based algorithm to the interprocedural case. This was sketched in [1]: here we explain the full method.
3. Clarifying some parts of the PCB-Based slicing algorithm presented in [1]. This includes the formal definition of how to represent programs by PCBs, and how to handle the communication of data flow information between different program parts efficiently. Thus reducing the exploration of irrelevant paths.

The rest of the paper is organized as follows. Section 2 introduces essential background. In Section 3 we describe PCBs and summarizes PCB-Based slicing approach in [1]. We illustrate how the data dependencies are saved and retrieved between slices in Section 4, while in Section 5 the communications between PCBs are filtered. Section 6 presents the two-modes algorithm. Section 7 onfly interprocedual slicing, In Section 8 we give an account for our experimental evaluation. Section 9 gives an account for related work, Section 10 concludes the paper and Section 11 acknowledgments

2 Preliminaries

2.1 Model Language

WHILE is a simplified model language for languages for safety-critical applications, such as SPARK Ada and MISRA-C. A program is a statement (**s**) consisting of a sequence of statements. WHILE statements are classified into two main types; elementary and conditional statements. Elementary statements are

assignment, *skip*, boolean expressions and the special *in_child* explained later. Conditional statements are *if*, *while* and *ifelse* statements.

We assume that each elementary and conditional statement is uniquely labeled in a program. **Label** is the set of global labels in a program and $\ell \in$ **Label**. Let a denote arithmetic expressions and b boolean expressions. The abstract syntax of the WHILE language is:

$$
\begin{aligned}
cs &::= [\texttt{if } [b]^{\ell} \texttt{ then } s']^{\ell'} \;\mid\; [\texttt{if } [b]^{\ell,\ell'} \texttt{ then } s' \texttt{ else } s'']^{\ell''} \;\mid\; [\texttt{while } [b]^{\ell} \texttt{ do } s']^{\ell'} \\
es &::= [x := a]^{\ell} \;\mid\; [b]^{\ell} \;\mid\; [skip]^{\ell} \;\mid\; [in\_child]^{\ell} \\
s &::= es \;\mid\; s'; s'' \;\mid\; cs
\end{aligned}
$$

2.2 Strongly Live Variable (SLV) Analysis

A data dependence relation is a relation on program labels relating points of variable definitions with their uses. We write $\ell \xrightarrow{v} \ell'$ to signify that ℓ is data dependent on ℓ', i.e., that the statement labeled ℓ' defines a variable v used by the statement labeled ℓ.

The data dependence relation has two sides: *Definition* and *Use*. The *Definition* is a statement where a variable (v) is updated and then reaches without being redefined to a statement using v (*Use*). Accordingly, *Use* is data dependent on *Definition*. In this context, the data dependence is symbolized by $\rightarrow$ as:

$$\ell \xrightarrow{v} \ell' \;\subseteq Label \times Var \times Label \tag{1}$$

where ℓ is data dependent on ℓ' in terms of the variable v. Sometimes, v is not specified in this relation.

A variable v is live at program point p if there is a definition free path from a use of v to p. A live variable analysis is a backward analysis [10] that computes the set of live variables associated with each program point. Strongly live variable analysis is a restriction of live variable analysis to an initial set of strong live variables, which are the variables of interest [10]. Thus, a variable is strongly live at program point p if there is a definition free path to a use of v and this use defines another strongly live variable.

SLV is a data flow analysis [10]. It generates a variable used in a particular statement as a SLV, propagates it backward untill reaching a statement defining it where it is killed. Thus, SLV analysis can be utilized to find the definitions that affect the used variables in a particular statement (use). This mechanism which computes from a use the set of definitions affecting it, is the main requirement in computing dynamically data dependence facts by backward slicing.

As a traditional dataflow method, SLV analysis relies on *functions* and *equations*; *gen* and *kill* functions generate and remove SLVs respectively from individual elementary statements, and the equations compose a mechanism to propagate backward the SLVs. Since these equations are not used in PCB-based

slicing, they are not presented here. The functions are defined as follows:

$$\begin{aligned} &kill(x := a) = \{x\}, \quad kill(b) = \emptyset, \quad kill(skip) = \emptyset, kill(in\_child) = \mathbf{Var} \\ &gen(x := a) = FV(a), gen(b) = FV(b), gen(skip) = \emptyset, gen(in\_child) = \emptyset \end{aligned} \tag{2}$$

$FV(a)$ denotes the set of program variables that appear in the expression a. **Var** is the set of variables in a program.

3 PCB-Based Slicing

This section recaptures the notions of Predicate Control Blocks (PCBs), PCB graphs and PCB based slicing as introduced in [1]. With respect to previous work the section contributes by describing the derivation of PCBs and PCB graphs in a more detailed manner.

3.1 Predicate Control Block Graphs

A PCB [1] refers to the encapsulation of a predicate and the set of elementary statements which are controlled directly by this predicate.

$$p ::= \{[b, es_1, \ldots, es_n], type\} \tag{3}$$

In addition to a predicate and a sequence of statements, PCBs carry types, *type*, signifying whether the PCB is linear, L, or cyclic, C. Intuitively, linear PCBs correspond to conditional statements, such as *if*, and cyclic PCBs correspond to iterative statements, such as *while*.

In the following let $+\!\!+$ denote concatenation of sequences and let : denote the standard cons operator, i.e., $b : [es_1, \cdots] = [b, es_1, \cdots]$. Further, we lift sequence indexing to PBCs where $p[0] = b$, and $p[n] = es_n$ for $p = \{[b, es_1, \ldots, es_n], type\}$.

A PCB graph is a pair (ϕ, ϵ), consisting of a map from labels to PCBs, ϕ, and a set of edges, ϵ, represented as pairs of labels. Following [1] we refer to the edges of the PCB graph as interfaces, and write $\ell_1 \hookrightarrow \ell_2$ instead of $(\ell_1, \ell_2) \in \epsilon$, whenever the PCB graph is given by the context.

The top-level translation of a program s to a PCB graph is defined as follows for any ℓ not in s.

$$\lambda(s) = (\phi[\ell \mapsto \{true^{\ell} : es, L\}, \epsilon), \; where \; es, (\phi, \epsilon) = \lambda_{\ell}(s)$$

The bulk of the translation is done by $\lambda_{\ell}(s)$, defined in Figure 2, where $final$ returns the last label in a sequence of elementary statements. Given a statement s the $\lambda_{\ell}(s)$ returns a linearized translation of s together with the PCB graph resulting from the translation. It might be worth pointing out a few key points of the algorithm. First, each PCB inherits the label of its predicate. Second, since *if..else* statements generate two PCBs, their predicates carry two distinct labels. Third, the place of each conditional statement is replaced by a placeholder (*skip*

for *if* or *while*; *in_child* for *if..else*)[1], whose label is the label of the original statement. The translation of compound statements works by first translating the parts, and then joining and extending the results to a new PCB graph.

$$
\begin{aligned}
&\lambda_{\ell_p}([x := a]^{\ell}) && = [x := a]^{\ell}, (\emptyset, \emptyset)\\
&\lambda_{\ell_p}([skip]^{\ell}) && = [skip]^{\ell}, (\emptyset, \emptyset)\\
&\lambda_{\ell_p}([\texttt{if } b^{\ell} \texttt{ then } s]^{\ell'}) && = [skip]^{\ell'}, (\phi', \epsilon')\\
&\qquad \textit{where } es, (\phi, \epsilon) = \lambda_{\ell}(s) \textit{ and } \phi' = \phi[\ell \mapsto \{b^{\ell} : es, L\}]\\
&\qquad \textit{and } \epsilon' = \epsilon \cup \{\ell_p \hookrightarrow \ell, final(es) \hookrightarrow \ell'\}\\
&\lambda_{\ell_p}([\texttt{while } b^{\ell} \texttt{ do } s]^{\ell'}) && = [skip]^{\ell'}, (\phi', \epsilon')\\
&\qquad \textit{where } es, (\phi, \epsilon) = \lambda_{\ell}(s) \textit{ and } \phi' = \phi[\ell \mapsto \{b^{\ell} : es, C\}]\\
&\qquad \textit{and } \epsilon' = \epsilon \cup \{\ell_p \hookrightarrow \ell, final(es) \hookrightarrow \ell'\}\\
&\lambda_{\ell_p}([\texttt{if } b^{\ell,\ell'} \texttt{ then } s \texttt{ else } s']^{\ell''}) = [in\_child]^{\ell''}, (\phi'', \epsilon'')\\
&\qquad \textit{where } es, (\phi, \epsilon) = \lambda_{\ell}(s) \textit{ and } es', (\phi', \epsilon') = \lambda_{\ell'}(s')\\
&\qquad \textit{and } \phi'' = (\phi \cup \phi')[\ell \mapsto \{b^{\ell} : es, L\}, \ell' \mapsto \{\neg b^{\ell'} : es', L\}]\\
&\qquad \textit{and } \epsilon'' = \epsilon \cup \epsilon' \cup \{\ell_p \hookrightarrow \ell, final(es) \mapsto \ell'', \ell_p \hookrightarrow \ell', final(es') \hookrightarrow \ell''\}\\
&\lambda_{\ell_p}(s; s') && = es +\!\!+ es', (\phi \cup \phi', \epsilon \cup \epsilon')\\
&\qquad \textit{where } es, (\phi, \epsilon) = \lambda_{\ell_p}(s) \textit{ and } es', (\phi', \epsilon') = \lambda_{\ell'_p}(s')\\
&\qquad \textit{and } \ell'_p = final(es)
\end{aligned}
$$

Fig. 2: Computation of PCB graphs

To illustrate the algorithm consider the program in Figure 3. The algorithm works recursively; in order to translate the top level program, the *while* and the *if* must be translated. In the reverse order of the recursive calls, the *if* is translated first, which gives P_7. No interfaces are created, since the body of the *if* does not contain any compound statements. The resulting PCB graph is returned to the translation of the body of the *while*, and extended with P_4 and interfaces $\ell_3 \hookrightarrow \ell_7$ and $\ell_8 \hookrightarrow \ell_4$. This gives the PCB graph rooted in P_4, which is returned to the top-level translation and the graph. The final result is produced by adding P_0 and interfaces $\ell_1 \hookrightarrow \ell_4$ and $\ell_9 \hookrightarrow \ell_3$.

3.2 PCB-Based Slicing Approach

A slicing criterion is a pair of $< \ell, v >$ where ℓ is a global label and v is a variable. if ℓ belongs to the PCB P, then $< \ell, v >$ is considered as a local problem in P. $< \ell, v >$ is solved in P by propagating it backward among the local statements in P. The propagation starts from ℓ and its aim is to find the statement in P that influences v at ℓ. Since the propagation relies on the order of the internal statements in the PCB, it is important to express the local problem by its local

[1] The reason of using placeholders will be explained in Section 5

index (i) in P rather than its global label ℓ. The local slicing problem in the PCB does not have more than one local solution because each PCB has a unique free-branching path.

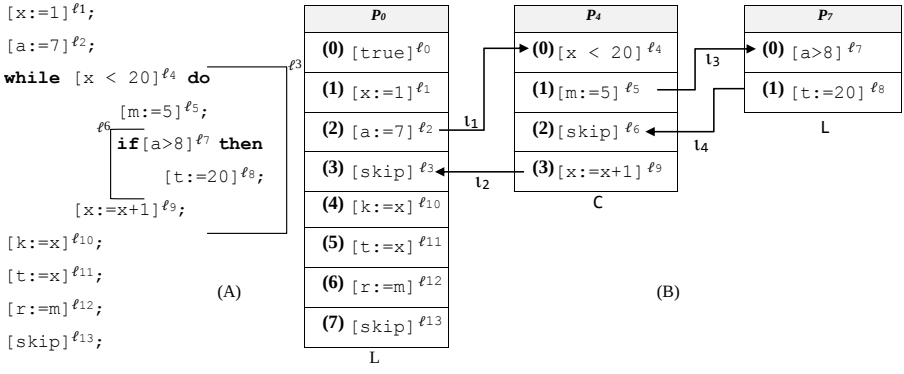


Fig. 3: Example 1

Local slicing problems of the PCB are stored in a single set (S). The PCB works as a standalone process that solves its local problems. Local problems are solved individually. While solving a problem, this problem might be *reproduced* in other PCBs, *killed* and new local problems might be *generated*. This cycle of *reproduction*, *killing*, and *generation* of local problems are iteratively repeated until no more local problem is available in any PCB.

The SLV query is solved by using *kill* and *gen* SLV functions. *kill(s)* gives the variables that s defines. *gen(s)* generates new local slicing problems from s. Therefore and henceforth, local slicing problem are named as SLV query.

Since the PCB represents a branch-free path, there is no use from using conventional fixed point iterations, which is designed to work with tree of branches and requires a set existing at each program point to save the SLVs reaching this point. The PCB has a single set to preserve its SLV queries and each SLV query is solved individually. The PCB encodes the direct control dependency. This encoding enables to capture immediately the control dependencies from the predicate of the PCB and the predicates of parents PCBs.

Suppose $S(P)$ is the single set of P. Each SLV query $< i, v >$ is fetched individually from $S(P)$. The first parameter which should be calculated for this query is its end index e. In linear PCBs, $e = 0$. Otherwise, $e = i + 1$. Then, $< i, v >$ proceeds backward from $P[i]$ toward $P[e]$. In circular PCBs, when $< i, v >$ reaches $P[0]$, it propagates backward by jumping to the last label in P and goes on backward until reaching $P[i + 1]$. In this context, the index of each visited statement in P will be j. Visiting $P[j]$ by $< i, v >$ causes one of these three cases:

case 1: if $v \nsubseteq kill(P[j])$ and $j \neq e$, then $S(P)$ remains as is

case 2: if $v \subseteq kill(P[j])$ then $< i, v >$ is removed from $S(P)$ and does not proceed more. if $P[j]$ was not sliced before, then the variables used in $P[j]$ will be generated as SLV queries and added to $S(P)$. As well, $P[j]$ is sliced.

case 3: if $v \nsubseteq kill(P[j])$ and $j = e$, then $< i, v >$ is removed from $S(P)$ and does not proceed more.

Example 1: In Fig. 3; Suppose $< \ell_{13}, r >$ is a global slicing criterion. It is translated to the SLV query: $< 7, r >$ in $S(P_0)$. Since P_0 is a linear PCB, $e = 0$. So, $< 7, r >$ is solved locally in P_0 by being propagated it from $P_0[7]$ to $P_0[0]$. The first statement visited by $< 7, r >$ is $P_0[7]$. Considering that $kill(P_0[7]) = \emptyset$ $S(P_0)$ remains as is. Next, $< 7, r >$ visits $P_0[6]$. Since $kill(P_0[6]) = r$, $P_0[6]$ is sliced, a new SLV query $< 5, m >$ is generated, $< 7, r >$ is removed from $S(P_0)$ and it no longer proceeds. Similarly, $< 5, m >$ is fetched and it visits the statements from $P_0[5]$ to $P_0[0]$. Since none of them kills m, $< 5, m >$ is removed from $S(P_0)$ after visiting $P_0[0]\square$

Notice that the value of m at $P_0[6]$ is affected also by $P_4[1]$. Therefore, a new slicing problem has to be created in P_4 according to the following rule:

Suppose P and P' are two PCBs connected by $P'[j'] \hookrightarrow P[j]$. When an SLV query $< i, v >$ visits $P[j]$ and not being killed at it, $< i, v >$ is reproduced in P' as $< j', v >$.

Example 2: In Fig. 3, P_0 is connected to P_4 by $P_4[3] \hookrightarrow P_0[3]$. When $< 5, m >$ visits $P_0[3]$, it is reproduced in P_4 as $< 3, m >$, which is processed in P_4 by visiting $P_4[3]$, $P_4[2]$ and $P_4[1]$. At $P_4[2]$, it is reproduced in P_7 as $< 1, m >$ and at $P_4[1]$ it is killed. The query $< 1, m >$ in P_7 does not have a local solution $\square$

Each interface $\ell \hookrightarrow \ell'$ associates to a set $R_m(\ell \hookrightarrow \ell')$, which saves the variable part of each query reproduced through $\ell \hookrightarrow \ell'$. R_m works as a blacklist whose elements are not allowed to be reproduced again through. This prevention is important in avoiding a possible non-termination which could occur when an SLV query $< i, v >$ is generated in a cyclic PCB and neither this PCB nor its child defines v. As a result, v would continuously be propagated between the parent and its child.

For if and while conditional statement which exists in other parent blocks, there is an execution path skipping the main body of the conditional statement. Thus, the local analysis in the parent block can neglect the existence of such conditional statements. For if-then-else conditional statement, there is no such skipping path. Therefore, the local analysis in its parent block could not neglect the existence of if-then-else statement. To handle this situation, if-then-else conditional statement is replaced by an *in_child* placeholder. *in_child*, which is designed especially for working as a placeholder for if-then-else statements, does not generate any SLV but it kills any SLV query visits it. It has this property because if-then-else has two branches, which both might kill the same variable.

For obtaining control dependencies; whenever any statement is sliced, then the predicate s_0 in its PCB has to be sliced if it was not sliced before. This routine has to be recursively applied also to the parent predicate until reaching the most outer PCB.

Suppose P is a child of the PCB P', as soon as $P[k]$ is sliced, then $P[0]$ should be sliced if it was not sliced and the variables used in $P[0]$ are generated as SLV queries. In addition, $P'[0]$ should be sliced if it was not sliced before and the variables used in it have to be generated as SLV queries.

Example 3: In Example 2, $P_4[0]$ is sliced due to slicing $P_4[1]$. Thus, $< 3, x >$ is generated in P_4 and solved internally at $P_4[3]$. In addition, $< 2, x >$ is reproduced in P_0 and solved at $P_0[1]$. Based on that, the slice of the global slicing criterion $< 7, r >$ consists of: $\ell_{12}, \ell_5, \ell_9, \ell_4, \ell_1, \ell_0$ □

4 Partial Data Dependency Graph (PDDG)

Sometimes, the source code might be studied from many different perspectives. e.g. the control unit of Example 1 might be studied first with respect to `ActA` and then to `ActB`. Thus, the same code should be sliced many times for individual slicing criteria. In PCB-based slicing, computing many slices suffers from the fact that the same data dependency should be computed from scratch whenever it becomes a requirement. This section shows a novel method to store and retrieve the data dependencies between slices. To do so, the computed data dependencies are stored in a graph form called Partial Data Dependence Graph (PDDG).

4.1 The Organization of a Partial Data Dependence Graph

In backward slicing, it is required to find the labels of definitions that may affect the variables used in the sliced label ℓ. PDDG is designed to store these definitions in order to be retrieved later. To build a PDDG, a special set $\delta(\ell)$ is added to each label (ℓ). $\delta(\ell)$ is initialized to the empty set. In assuming ℓ is sliced, each statement defining any variable used in ℓ should be computed, sliced and its label has to be added to $\delta(\ell)$. Consequently, the data dependencies are organized in PDDG as *use-definitions* form. In this form, the set of *definitions* affecting a particular label are stored in this label, which is a *use* to those definitions. This design enables to retrieve once all the *definitions* of this *use* when it is sliced again for a different slicing criteria.

In subsection 3.2, we saw how the definitions of ℓ are computed by generating an SLV query for each variable used in ℓ and propagate it backward. When the query $< i, v >$ generated from ℓ reaches a statement ℓ' defining v, then this is the best time to capture the data dependency between ℓ and ℓ'. The problem is that the origin of $< i, v >$ is not known because i is changed whenever it is reproduced in a new PCB. Thus, a new field (src) is added to the SLV query type, which becomes a triple: $< loc, src, var >$. src is assigned to the global label, which the SLV query is generated from. Hence, if $< i, \ell, v >$ is killed at ℓ', then $\ell \xrightarrow{v} \ell'$ which is satisfied by adding (ℓ', v) to $\delta(\ell)$. Based on that, $\delta(\ell)$ is defined as:

$$\delta(\ell) \subseteq \mathcal{P}(LABEL \times VAR) \tag{4}$$

The reason of why $\delta(\ell)$ is not a pool of labels only will be explained in Section 4.2.

4.2 The Hindrance of Interfaces

An interface ι is associated with a set $R_m(\iota)$ in order to prevent any variable from being reproduced more than once. Since the same variable may exist in different SLV queries, the interface black list may cause missing data dependences.

Example 4: In Fig. 3, if $P_0[4]$ and $P_0[5]$ are sliced, then SLV queries $< 3, P_0[4], x >$ and $< 4, P_0[5], x >$ are added to $S(P_0)$. Suppose $< 4, P_0[5], x >$ is fetched first, then it is solved at $P_0[1]$. As well, when $< 4, P_0[5], x >$ visits $P_0[3]$, it is reproduced through ι_2 in P_4 as $< 3, P_0[5], x >$ and $R_m(\iota_2) = \{x\}$. At P_4, it is solved at $P_4[3]$, So, $\delta(P_0[5]) = \{(P_0[1], x), (P_4[3], x)\}$. $< 3, P_0[4], x >$ is solved also at $P_0[1]$ but it could not be reproduced through ι_2 because $x \in R_m(\iota_2)$. Thus, $(P_4[3], x)$ could not be added to $\delta(P_0[4])$. In other words, $P_0[4] \xrightarrow{x} P_4[3]$ is not recognized $\square$

The hindrance of interface is resolved by using a *transition point*, which refers to a label (t) existing in the path from ℓ' and ℓ, where $\ell \xrightarrow{v} \ell'$. The transition point helps in expressing a data dependence relation by two fake data dependencies. Accordingly, $\ell \xrightarrow{v} \ell'$ is represented by $\ell \xrightarrow{v} t$ and $t \xrightarrow{v} \ell'$. To overcome the hindrance of the interface $(\ell_1 \hookrightarrow \ell_2)$, we consider its ingoing side ℓ_2 as a transition point. Hence, if the SLV query $< i, \ell, v >$ visits ℓ_2 and $kill(\ell_2) \neq v$ then the fake data dependence $src \xrightarrow{v} \ell_2$ is created by adding (ℓ_2, v) to $\delta(\ell)$. If $< i, \ell, v >$ is allowed to be reproduced in $\mathcal{M}(\ell_1)$, then it will be reproduced as $< i', \ell_2, v >$, where i' is the index of ℓ_1 in $\mathcal{M}(\ell_1)$.

Example 5: Example 4 is resolved as follows: when $< 4, P_0[5], x >$ reaches $P_0[3]$, then it is reproduced in P_4 as $< 3, P_0[3], x >$ and $(P_0[3], x)$ is added to $\delta(P_0[5])$. When $< 3, P_0[4], x >$ reaches $P_0[3]$, $(P_0[3], x)$ is added to $\delta(P_0[4])$ without being reproduced at P_4. Hence, $P_0[5] \xrightarrow{x} P_4[3]$ is expressed as $P_0[5] \xrightarrow{x} P_0[3]$ and $P_0[3] \xrightarrow{x} P_4[3]$. Similarly, $P_0[4] \xrightarrow{x} P_4[3]$ is expressed as $P_0[4] \xrightarrow{x} P_0[3]$ and $P_0[3] \xrightarrow{x} P_4[3]$ $\square$

Notice that in (4), we defined $\delta(\ell)$ as a pool of pairs rather than a pool of labels. This organization prevents creating non-existing dependencies. See Example 12.

Example 12 Suppose $\iota = \ell_i \hookrightarrow \ell_j$ is in the path of: $\ell_1 \xrightarrow{x} \ell_a$ and $\ell_2 \xrightarrow{y} \ell_b$. If δ was a set of labels only, the following data dependencies would be represented: $\ell_1 \rightarrow \ell_j$, $\ell_2 \rightarrow \ell_j$, $\ell_j \rightarrow \ell_a$ and $\ell_j \rightarrow \ell_b$, which means ℓ_1 is data dependent on ℓ_b and this is not correct.

5 SLV Filtering

In PCB-based program representation, interfaces correspond to edges in CFG program representations. Both of them are constructed to model program flows. In this section we explain in depth how the interfaces are implemented in an efficient manner to prevent SLV queries from exploring irrelevant PCBs or paths. This presentation eliminates unnecessary computations.

The cornerstone in making SLV filtration is in associating the interfaces with whitelist sets rather than blacklists which we introduced in Section 3.2. The *White List* is a smart way to implement the black list in a "negative" way,

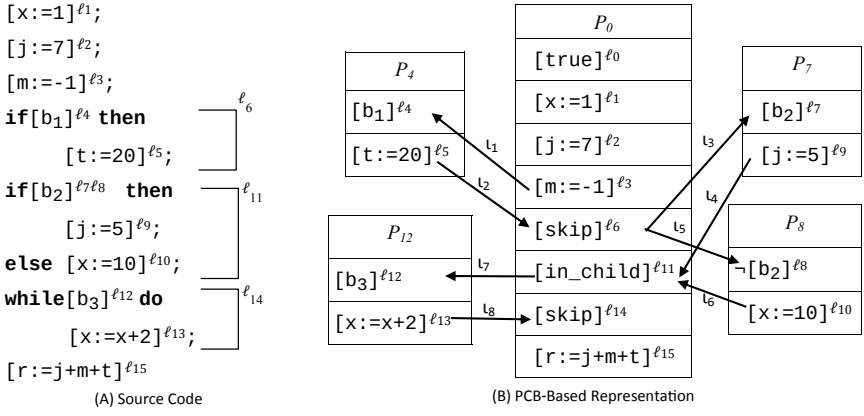


Fig. 4: PCB-Based Representation (Whitelists + Placeholders)

representing what is allowed to go through rather than what is not allowed to go through. The whitelist set associated with an interface $\iota = P.\ell \hookrightarrow P'.\ell'$, where P is a child to P', is initialized to the variables defined in P and its child PCBs. The whitelist associated with ι is denoted $W_m(\iota)$. As soon as any of the variables stored in $W_m(\iota)$ is reproduced through ι, this variable is removed from $W_m(\iota)$ and it becomes no longer allowed to be reproduced again. Based on that, the number of the variables in the whitelist is in the worst case proportional to the number of the variables in P and it is decremented after every reproduction. When the whitelist set becomes empty, then its interface could be deleted to relieve the analysis from the overhead of its existence. This is contrary to the blacklists whose sizes in the worst case are proportional to the number of the variables in the program and they increments after every reproduction. Apparently, changing from blacklists to whitelists makes a significant improvement. (See Table 2 in Section 8.2)

Example 5: Fig.4 is an example of preventing SLV queries from exploring irrelevant paths. Suppose ℓ_{15} is sliced and three SLV queries are generated whose variable components are m, x and t. Notice the t has a definition in P_4. In backward dataflow analysis techniques, t propagates in all statements in P_{12}, P_8, P_7 and P_4 although its unique external solution is in P_4. Similarly, j is reproduced in P_{13}, P_8, P_7 and P_4 although it is defined only in P_7. Figure 4-B shows four ingoing interfaces to P_0: ι_2, ι_4, ι_6 and ι_8. Their whitelist sets are initialized to: $W_m(\iota_2) = \{t\}$, $R_m(\iota_4) = \{j\}$, $W_m(\iota_6) = \{x\}$ and $W_m(\iota_8) = \{x\}$. When t is generated as an SLV from $P_0.\ell_{15}$, it visits ℓ_{14} and ℓ_{11} and neglects P_{12}, P_8 and P_7 because their whitelists do not include t. When it visits ℓ_6, t is reproduced in P_4 because it defines t. To sum up, we can say that t is forwarded directly to P_4.

6 Two-Mode PCB Slicing Algorithm

The PDDG stores and retrieves data dependencies between slices. This mechanism is yielded by designing a slicing method that behaves in two modes. In the first mode the definitions influencing ℓ are retrieved from $\delta(\ell)$ and the second mode generates the variables used in ℓ as SLV queries to reach these definitions. The contents of $\delta(\ell)$ assist the analysis to determine in any mode it should run when ℓ is sliced.

Alg.5a slices the internal labels of the PCB P with respect to SLV queries stored in S_P. These SLVs are fetched individually (line 4) until no more query exists (3). The SLV query $< i, s, v >$ visits local statements from i to e, which is calculated at (7-8). j refers to the index of the current visited statement. j is calculated from the type of P and the current value of j (10).

When $< i, s, v >$ visits $P[j]$, we check whether $P[j]$ kills v. If it does not (14), then v is reproduced if $P[j]$ is an ingoing side of an interface (15). Otherwise (16), $P[j]$ should be sliced (18,19) and added to $\delta(s)$ (17). At this point, there are two modes; if $P[j]$ was already sliced in a previous slice (20) then the definitions stored in $\delta(P[j])$ should be sliced by the procedure TRCK(21). Otherwise, the second mode generates the variables used in $P[j]$ as SLV queries in S_P (23) and in $S_{P'}$ if there is $P'.\ell' \hookrightarrow P[j]$(25) . After every visit, S_P is updated according to the transfer function $f^{j,e}_{(i,s,v)}(S_P)$ shown in Fig.5e. This function has three cases: $< i, s, v >$ is removed from S_P, S_P is not updated and finally, $< i, s, v >$ is removed from S_P and the variables used in the current visited statement $P[j]$ are generated as SLV queries.

TRCK(ℓ, var, N_{slc}) (Fig.5d) traces *use-definition* chains from ℓ. This is performed by first slicing the definitions stored in $\delta(\ell)$. Then TRCK is called recursively to slice the definitions which are stored in each of these labels and so on. In the pool $\delta(\ell)$, we slice every label stored in $\delta(\ell)$ unless ℓ is a placeholder. In this case, we slice only the labels that influence *var*.

Suppose ℓ that exists in P is sliced. Since ℓ is control dependent on $P[0]$, $P[0]$ should be sliced, which in its turn is control dependent on the predicate of P parent and so on. In Fig. 5d, this hierarchical structure of control dependencies goes on until reaching the PCB representing the most outer PCB.

Fig.5d shows INTFC function. This function reproduces *var* from ℓ to $P'[k]$ if ℓ is an ingoing side of an interface connecting ℓ with $P'[k]$.

Finally, the role of SELECT function is in fetching individually the SLV queries from S_P. The role of PARENT(P) is in getting the parent PCB of P.

7 On-the-fly Interprocedural Slicing

In our previous work [1], we sketched an inter-procedural slicing algorithm for procedures having a single out formal argument and a single return statement. This section extends this method to be applied to real procedures, which have many out formal arguments and multiple return statements or points. Furthermore, the construction of PCBs in inter-procedural programs is formalized math-

ematically, the syntax of the model language is extended to accommodate inter-procedural programs, the special transfer function for call sites is improved, and the *super interface* concept is introduced.

a: SlicePCB($P, \mathcal{I}, N_{slc}$)

```
1 // P: current PCB. I:Interfaces
2 // N_slc:sliced labels
3 while S_P ≠ ∅ do
4   | < i, s, v >:= Select(S_P) ;
5   | j := −1;
6   | if (P is C and i ≠ final(P)) then
7   |   | e := i + 1
8   | else e = 0;
9   | repeat
10  |   | switch j do
11  |   |   | case j = −1 : j := i; break;
12  |   |   | case j > 0 : j := j − 1; break;
13  |   |   | case j = 0 : j := final(P) ;
14  |   |   | if (v ∉ kill(P[j])) then
15  |   |   |   | Intrfc(P[j], s, v, I);
16  |   |   | else
17  |   |   |   | δ(s) := {δ(s) ∪ (P[j], v)};
18  |   |   |   | if (P[j] ∉ N_slc) then
19  |   |   |   |   | N_slc := N_slc ∪ {P[j]};
20  |   |   |   |   | if (δ(P[j]) ≠ ∅) then
21  |   |   |   |   |   | Trck(P[j], v, N_slc);
22  |   |   |   |   | else
23  |   |   |   |   |   | S_P := f^{j,e}_{(i,s,v)}(S_P);
24  |   |   |   |   |   | foreach x ∈ gen(P[j]) do
25  |   |   |   |   |   |   | Intfc(P[j], P[j], x, I);
26  |   |   |   |   | Cntrl(P, N_slc, I);
27  |   |   |   |   | break;// Fetch new SLV
28  |   | until j = e;
29 return N_slc;
```

b: Intfc($\ell, src, var, \mathcal{I}$)

```
1 foreach (P'[k] ↪ ℓ ∈ I) do
2   | i = P'[k] ↪ ℓ;
3   | if (var ∈ W_m(i)) then
4   |   | W_m(i) := W_m(i)\{var} ;
5   |   | S_P' := S_P' ∪ {(k, src, var)};
6 return;
```

c: Trck(ℓ, var, N_{slc})

```
1 foreach (ℓ', var') ∈ δ(ℓ) do
2   | if s_ℓ ≠ skip ∧s_ℓ ≠ in_child then
3   |   | N_slc := N_slc ∪ ℓ' ;
4   |   | Trck(ℓ', var', N_slc );
5   |   | Cntrl(M(ℓ), N_slc, I)
6   | else
7   |   | if var' = var then
8   |   |   | N_slc := N_slc ∪ ℓ' ;
9   |   |   | Trck(ℓ', var', N_slc );
10 return;
```

d: Cntrl($P, N_{slc}, \mathcal{I}$)

```
1 repeat
2   | if (P[0] ∈ N_slc ) then return;
3   | N_slc := N_slc ∪ P[0];
4   | if (δ(P[0]) ≠ ∅) then
5   |   | foreach v ∈ gen(P[0]) do
6   |   |   | Trck(P[0], v, N_slc );
7   | else
8   |   | S_P := S_P ∪ {(0, P[0], v)|v ∈
    |   | gen(P[0])}
9   |   | foreach v ∈ gen(P[0]) do
10  |   |   | Intfc(P[0], P[0], v, I);
11  |   | P := parent(P);
12 until P ≠ ∅;
13 return;
```

e: Transfer Function

$$f^{j,e}_{(i,s,v)}(S_P) =$$

$$\begin{cases} S_P \backslash \{(i,s,v)\} & \text{if } (j = e \wedge v \notin kill(P[j])) \\ & \quad \vee\ (v \in kill(P[j]) \wedge P[j] \in N_{slc}) \\ & \quad \vee\ (v \in kill(P[j]) \wedge \delta(P[j]) \neq \emptyset) \\ S_P & \text{if } j \neq e \wedge v \notin kill(P[j]) \\ S_P \backslash \{(i,s,v)\} \cup \{(j-1, P[j], u) | u \in gen(P[j])\} & \\ \quad \text{if } v \in kill(P[j]) \wedge P[j] \notin N_{slc} \wedge \delta(P[j]) = \emptyset & \end{cases}$$

Fig. 5: Two-Mode Algorithms

The algorithm is restricted to non-recursive procedures. This is a common restriction in safety-critical applications.

To be able to study inter-procedural slicing we must extend the language with procedure declarations and procedure call. Procedure declarations consist of a name, zero or more formal arguments and a body. The formal arguments are declared to be either in arguments, that pass information into the procedure, or out arguments that in addition pass information from the procedure to the caller. The global variables are considered out arguments. Procedure calls are, without loss of generality, restrained to variables. Let F range over procedure names and let $\psi_{P[j]}$ denote the bijective function that maps the actual arguments at call site $P[j]$ to the formal arguments of the procedure. For a procedure F and its output formal parameter v, $\mu_F(v)$ refers to the set of formal arguments that might influence v. The extended syntax is defined as follows.

$$
\begin{aligned}
p &::= d \mid d\ p \\
arg &::= \texttt{in}\ x \mid \texttt{out}\ x \\
d &::= [\texttt{proc}\ F\ \overline{\texttt{arg}}\ s]^{\ell_F} \\
es &::= \ldots \mid [\texttt{call}\ F\ \overline{x}]^{\ell} \mid [\texttt{return}]^{\ell}
\end{aligned}
$$

We extend the notion of interfaces from being between two labels to being a relation on sets of labels. For clarity we write ℓ for the singleton set $\{\ell\}$. Let $calls(F)$ be the set of labels of calls to F.

To compute the PCB-graph in the presence of procedures, the algorithm found in Section 3.1 is extended as shown in Figure 6.

$$
\begin{aligned}
\lambda_{\ell_p}([\texttt{call}\ F\ \overline{x}]^{\ell}) &= [\texttt{call}\ F\ \overline{x}]^{\ell}, (\emptyset, \emptyset) \\
\lambda_{\ell_p}([\texttt{return}]^{\ell}) &= [\texttt{return}]^{\ell}, (\emptyset, \{\ell \hookrightarrow calls(F)\})
\end{aligned}
$$

On the top-level

$$
\begin{aligned}
\lambda([\texttt{proc}\ F\ \overline{\texttt{arg}}\ s]^{\ell}) &= (\phi[\ell \mapsto \{true^{\ell} : es, L\}], \epsilon \cup \{calls(F) \hookrightarrow \ell\}) \\
&\quad \textit{where } es, (\phi, \epsilon) = \lambda_{\ell}(s) \\
\\
\lambda(d\ p) &= (\phi_1 \cup \phi_2, \epsilon_1 \cup \epsilon_2) \\
&\quad \textit{where } es, (\phi_1, \epsilon_2) = \lambda(d) \textit{ and } (\phi_2, \epsilon_2) = \lambda(p)
\end{aligned}
$$

Fig. 6: Extension of PCB graph computation

The resulting algorithm introduces two inter-procedural interfaces: one going from the call sites to the entry label of the procedure (*Many-to-One* Interface) and one going from a return label to the call sites (*One-to-Many* Interface). Each of these two interfaces could be expressed by a set of a normal interfaces (single-

ton label to singleton label). Thus, *Many-to-One* and *One-to-Many* interfaces are referred to *Super Interfaces*.

The Super Interface comprises of many thin parts linked through a joint to a single thick part. Together with the single thick part, each thin part constitutes a normal interface. Hence, the thick part is shared between all normal interfaces contained in a super interface. This design allows some inter-procedural information to be shared between the different call sites of a procedure, whereas other information is exclusively linked to individual call sites

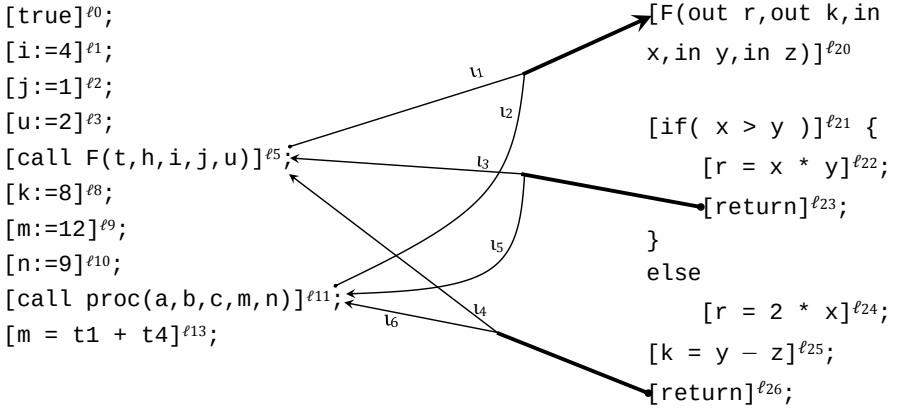


Fig. 7: Inter-procedural Example

In Fig. 7, a Many-to-One super interface is: $\ell_5, \ell_{11} \hookrightarrow \ell_{20}$. It contains two normal interfaces: $\ell_5 \hookrightarrow \ell_{20}$ and $\ell_{11} \hookrightarrow \ell_{20}$. Further, we have two One-to-Many interfaces: $\ell_{23} \hookrightarrow \ell_3, \ell_5$ and $\ell_{26} \hookrightarrow \ell_4, \ell_6$. Similarly, each could be expressed by two normal interfaces.

The shared thick part in a *Many-to-One* interface is linked with a μ_F definition for each out formal argument. The thin part holds a ψ_j^{-1} definition for each formal argument. In Fig. 7, the thin part of ι_1 associates with $\psi_{\ell_5}^{-1}(r) = t$, $\psi_{\ell_5}^{-1}(k) = h$, $\psi_{\ell_5}^{-1}(x) = i$, $\psi_{\ell_5}^{-1}(y) = j$, $\psi_{\ell_5}^{-1}(j) = u$. The thin part of ι_2 associates with $\psi_{\ell_{11}}^{-1}(r) = a$, $\psi_{\ell_{11}}^{-1}(k) = b$, $\psi_{\ell_{11}}^{-1}(x) = c$, $\psi_{\ell_{11}}^{-1}(y) = m$, $\psi_{\ell_{11}}^{-1}(j) = n$. The thick shared part of ι_1 and ι_2 contains $\mu_{\ell_{20}}(r)$ and $\mu_{\ell_{20}}(k)$.

The thick shared part in a *One-to-Many* super interface associates with a white list that contains initially all the out formal arguments. A thin part connecting a return statement to a j call site holds a ψ_j definition for each out actual argument at j. Based on that, the shared thick part in $\ell_5, \ell_{11} \hookrightarrow \ell_{20}$ associates with $R_m = \{r, k\}$. The thin part in ι_3 holds $\psi_{\ell_5}(t) = r$ and $\psi_{\ell_5}(h) = k$.

The SLV queries reproduced through super interfaces are processed in two stages, one over the thick part and the another in the thin part regardless of the order, before they reach the opposite sides. We indicate to this fact by saying

the variable is *thrown* on a thin or thick part. The *throwing* of a variable on a super interface part means that the variable is going to be processed according to the stage of this part.

There are two types of inter-procedural slicing. The first type is the *Up* slicing. Up slicing occurs when the procedure body is a source of SLV queries due to the existence of slicing criteria in it. Since any of call sites might run the procedure body, the procedure header is control dependent on all its call sites. Thus, the direction of SLV queries is from the procedure body to all its call sites.

The second type is the *Down* slicing. This type refers to the situation where a particular call site is the source of SLV queries in a procedure body. Therefore, for the statements which are sliced in the procedure body with respect of these SLV queries, the procedure header is control dependent on this call site. In other words, in down slicing, the SLV queries reach a procedure's header are not reproduced in all call sites. Instead, the transfer function shown in Sec. 7.1 is applied.

To maintain the context-sensitivity, we define two stacks, ST_{call} and ST_{var}. ST_{call} stores the last call site and ST_{var} stores the formal arguments used to compute μ_F.

7.1 The Transfer Function

In the down slicing, when the call site whose label is $P[j]$ is visited by an SLV query $< i, s, x >$, S_P has to be updated according to the following transfer function:

$$f^{j,F}_{i,e,x}(S_P) = \begin{cases} S_P & \psi_j(x) \text{ is undefined} \\ S_P \backslash \{(i,s,x)\} \cup \{(j-1, P[j], \psi_j^{-1}(u) \mid u \in \mu_F(\psi_j(x))\} & \mu_F(\psi_j(x)), \psi_j(x) \text{ defined} \end{cases} \quad (5)$$

The first case occurs when x is not an *out* actual argument at $P[j]$, so S_P is not updated. In the second case, x is an *out* actual argument at $P[j]$, $\mu_F(\psi_j(x))$ is computed. Thus, $< i, s, x >$ is killed and the actual arguments at $P[j]$ whose correspondent formal arguments belong to $\mu_F(\psi_j(x))$ are generated as SLV queries.

7.2 The Algorithm of Down Slicing and the Computation of μ_F

Suppose $P[j]$ is a call site of F and ι_r is an interface from $P[j]$ to a return statement in F and ι_h is an interface from F header to $P[j]$. The down slicing algorithm is:

1. When an SLV query $< i, s, x >$ visits a call site $P[j]$ and x is an out argument at $P[j]$, then:
   - if ι_h contains an already computed $\mu_F(\psi_{P[j]}(x))$, then we go to 9
     Note: ι_h and ι_r are known from the call site side.
   - Otherwise:
     - x is thrown on the thin part of ι_r.

- The analysis is completely frozen on $P[j]$ side.

2. if the thin part of ι_r receives a variable x, then we get $\psi_{P[j]}(x)$, say v. Then, v is pushed in ST_{vars} and $P[j]$ is pushed in ST_{calls}. Finally, v is thrown on the thick part of ι_r.
3. At the thick part of ι_r, v is removed from $W_m(\iota_r)$ and it is reproduced in the opposite side of ι_r (a return statement in F).
4. F's body is sliced with respect to the SLV queries of its PCBs. When no more SLV query is alive in F's PCBs, we move to the thick part of the many-to-one interface.
5. In the thick part of the many-to-one super interface:
   - We read from the top of ST_{vars} the formal argument which F is sliced with respect to. In other words, we retrieve v. Then we redefine $\mu_F(v)$ held by the thick part from the dependencies from v at return statements to F's formal parameters located in F's header. The edges of PDDG can be tracked to find these dependencies.
   - We retrieve from the top of ST_{calls} the call site, which is $P[j]$. From $P[j]$ we find ι_h. Then we move to the thin part of ι_h.
6. By using the $\psi^{-1}_{P[j]}$ definitions held by the thin part of ι_h, we reverse each formal argument in $\mu_F(v)$ to its actual argument at $P[j]$.
7. ST_{var} and ST_{calls} are popped.
8. P is released from being frozen.
9. The transfer function in Eq. 5 is applied and computed from $\mu_F(v)$ and $\psi^{-1}_{P[j]}$ definitions held in ι_h.
10. The analysis goes on.

8 Results and Discussions

To measure the efficiency of the proposed approach, we have implemented four algorithms:

- **A**: implements the proposed methods in this paper; two-modes PCB-based algorithm, PDDG and filtering SLVs through shrinking sets.
- **B**: is an implementation of the original PCB-based slicing [1]. It produces single slices, filters SLVs by whitelists, but it does not implement PDDG.
- **C**: PDG-based slicing [11]
- **D**: is an implementation of the original PCB-based slicing [1]. It produces single slices, filters SLVs by blacklists and it does not implement PDDG.

These algorithms are implemented by using Microsoft Visual C++ 2013. The experiments have been run on an Intel Core i5 with a 2.66GHz processor, 8 GB RAM, and 64-bit operating system.

8.1 The Efficiency of Using PDDG and Two-Mode Algorithm

To setup the comparisons, a synthetic program is produced automatically. It is an intra-procedural program, $125K$ statements, the predicates constitute 26%

No.Slice	Slice Size	(A) ms	(B) ms	Speedup	(C) ms
1	69%	234	180	0,8	12204
2	33%	468	108	0,2	1
3	14%	1	54	54	1
4	15%	1	54	54	1
5	51%	234	144	0,6	1
6	21%	1	72	72	1
7	79%	18	216	12	1
8	58%	1	180	180	1
9	40%	1	108	108	1
10	37%	1	108	108	1
11	45%	1	126	126	1
12	10%	1	18	18	1
13	20%	1	72	72	1
14	39%	1	108	108	1
15	50%	1	144	144	1
sum		965	1692		12204

Table 1

of the program, and it has 50 variables. This program has to be sliced by (A), (B) and (C) according to 15 distinct and individual slicing criteria.

In Table 1, the first entry shows the times for computing the first slice. Then, for each subsequent slice, the additional time for computing this slice is shown. The last entry shows the total times for computing all 15 slices. The second column gives the size of each slice relative the size of the whole program.

(B) computes every slice individually from scratch. (B) shows that as slices gets bigger, more computations are performed and intuitively more execution time is consumed. (C) shows that for PDG-based slicing, the first slice is the heaviest than others with respect to the execution time. Afterwards, very little time is consumed to compute each slice.

most the PDDG is constructed while (A) computes the slices: (1,2,5). Afterwards, the speedups (B / A) shown in the fifth column vary from 4.8 to 156 for the slices from 5 to 15. Hence, the data dependencies that are accumulated in (1,2,5) are used in computing the slices from 6 to 15. The main advantage of the two-modes slicing algorithm is that it does not need a full comprehensive analysis of the program at the beginning. As well, it does not lose previous computations.

In comparing (A) and (C), we find that both of them depend on a graph form to retrieve their dependencies. By comparing their results, we find that for the slices from 6 to 15, the execution times are very close together, which is because for both algorithms the slicing mainly turns into a backwards search in a dependence graph. Finally, the last row accumulates the execution times

obtained by each implementation. The results indicate that the two-modes slicing perform significantly better than our previous PCB-based algorithm when computing many slices for the same code, and that also PDG-based slicing is outperformed for a moderate number of slices.

8.2 Whitelist vs Blacklist

The two local implementations (B and D) are used to measure the results of the change from blacklists to whitelists. To do so, six synthetics source code programs were produced to be analyzed by (B) and (D). These six programs differ in their number of variables, which varies from 25 to 800. To ensure fair comparisons, other factors which could influence execution times are fixed. Thus, the size of each synthetic program is $50K$, the number of predicates in every program is around 6500 and the slice size is 70% from the total size.

No. Var.	25	50	100	200	400	800
Blacklists - D (s)	0.10	0.35	1.85	10.85	64	424
Whitelists - B (s)	0.014	0.052	0.083	0.146	0.25	0.48
Speedup (D/B)	7.1	6.7	22.2	74.3	256	883

Table 2: Whitelists *vs* Blacklists

There are two facts can be easily read from Table. 2, the first is that using whitelists enhances significantly the performance of the analysis. The second, which is more important than the first, shows that the superiority of the whitelists increases as more variables are added to the source code. While more variables are added to program, thus generating more SLV queries, moving more SLV queries between the PCBs and performing more operations through black and white lists. Since the blacklists sizes are proportional to the number of the programs' variables and on the other side, the whitelists sizes are proportional to the PCBs' variables, the execution times of (D) shows much more higher sensitivity than (B) to the change of the number of programs' variables.

9 Related Work

Program slicing was first introduced by Weiser [13] in the context of debugging. Ottenstein et al. [9, 11] introduced the PDG, and proposed its use in program slicing. PDG-based slicing has then been the classical program slicing method. Horwitz et al. [17] extended the PDG to as System Dependence Graph to capture calling context of procedures.

Hanjal and Forgàcs [6] propose a complete demand-driven slicing algorithm for inter-procedural well-structured programs. Their method is based on the propagation of tokens over Control Flow Graph (CFG). Harrold and Ci [16] propose a demand-driven method that computes only the information which

is needed in computing a slice. Harrold's method is based in Augmented CFG (ACFG). Furthermore, there are more special works for computing interprocedural information in a demand-driven way [18, 19].

10 Conclusions and Future Work

We have shown how to extend our previous algorithm for demand-driven slicing of well-structured programs [1] into an algorithm that can efficiently compute several slices for the same program. The main mechanism for achiveing good performance is to store and reuse previously computed data dependencies across several slices. We also make some clarifications regarding the original algorithm [1], including a formal description how the underlying PCB program representation is computed from the program code, and a description of how the "filtering" of Strongly Live Variables at interfaces can be implemented efficiently.

An experimental evaluation indicates that the new two-mode algorithm, with stored and reused data dependences, performs considerably better than the previous version when taking several slices of the same code. It also performs significantly better than the standard, PDG-based algorithm in the experiment

There are a number of possible future directions. One direction is to directly apply the slicing algorithm to speed up the verification of safety-critical software. For instance, SPARK Ada programs are often filled with assertions to be checked during the verification process. Formal methods for checking assertions, like symbolic execution [20], can be very prone to path explosion: slicing with respect to different slicing criteria derived from the assertions can then help to keep the complexity under control. A second direction is to generalise the slicing approach to richer languages including procedures, pointers, and object-oriented features, and to gradually relax the requirements on well-structuredness. A final observation is that the SLV analysis performed by our algorithm provides a pattern to perform other dataflow analyses, like Reaching Definitions and Very Busy Expressions [10], efficiently on well-structured code.

11 Acknowledgments

The authors thank Daniel Hedin, Daniel Kade, Iain Bate and Irfan Sljivo for their helpful comments and suggestions. This work was funded by The Knowledge Foundation (KKS) through the project 20130085 Testing of Critical System Characteristics (TOCSYC), and the Swedish Foundation for Strategic Research under the SYNOPSIS project.

References

1. Khanfar, H., Lisper, B., Abu Naser, M.: Static Backward Slicing for Safety Critical Systems. ADA-Europe 2015, The 20th International Conference on Reliable Software Technologies, , 9111 (50-65), June 2015

2. Lisper, B., Masud, A.N., Khanfar, H.: Static backward demand-driven slicing. In: Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation. pp. 115–126. PEPM '15, ACM, New York, NY, USA (2015)
3. Yunja Choi, Mingyu Park, Taejoon Byun, Dongwoo Kim: Efficient safety checking for automotive operating systems using property-based slicing and constraint-based environment generation. Sci. Comput. Program. 103: 51-70 (2015)
4. Kraft, J.: Enabling Timing Analysis of Complex Embedded Software Systems. Ph.D. thesis, Mälardalen University Press (August 2010)
5. Ákos Hajnal and István Forgács: A precise demand-driven definition-use chaining algorithm, Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on, Budapest, 2002, pp. 77-86. doi: 10.1109/CSMR.2002.995792
6. Ákos Hajnal and István Forgács: A demand-driven approach to slicing legacy COBOL systems",Journal of Software Maintenance,volume 24, no. 1, pages67–82, 2012
7. Agrawal, G.: Simultaneous demand-driven data-flow and call graph analysis, in Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on Software Maintenance, ICSM 1999,pages 453–462.
8. Agrawal, H.: On slicing programs with jump statements. SIGPLAN Not. 29(6), 302–312 (Jun 1994)
9. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (Jul 1987)
10. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2^{nd} edition. Springer (2005), iSBN 3-540-65410-0
11. Ottenstein, K.J., Ottenstein, L.M., The program dependence graph in a software development environment. SIGSOFT Softw. Eng. Notes 9(3), 177–184 (Apr 1984)
12. Tip, F., A survey of program slicing techniques. Journal of Programming Languages 3, 121–189 (1995)
13. Weiser, M.: Program Slicing. IEEE Transactions on Software Engineering, 352–357 (Jul 1984)
14. Dimov Aleksandar and Chandran, Senthil Kumar and Punnekkat, Sasikumar: How Do We Collect Data for Software Reliability Estimation, 11th International Conference on Computer Systems and Technologies,CompSysTech '10,2010, ACM.
15. Eric Larson: Assessing Work for Static Software Bug Detection, 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies, WEASELTech '07, 2007, ACM NewYork
16. M. J. Harrold and N. Ci, Reuse-driven interprocedural slicing. In The 20th International Conference on Software Engineering, pages 7483, Apr. 1998.
17. S. Horwitz, T. Reps, and D. Binkley: Interprocedural slicing using dependence graphs". ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 12 Issue 1, Jan. 1990, 26-60. ACM New York, NY, USA
18. E. Duesterwald and M. L. Sofifa: Demand-driven computation of interprocedural data flow. In Proceedings of 22nd ACM Symposium on Principles of Programmzng Languages, pages 37-48, January 1995.
19. S. Horwitz, T. Reps, and M. Sagiv: Demand interprocedural dataflow analysis. SIGSOFT '95 Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, pages 104-115, 1995. ACM New York.
20. J.C. King: Symbolic Execution and Program Testing, Communications of the ACM, Vol: 19, No. 7, July 1976, pp. 385-394. ACM New York

Derivation of Parallel Programs of Recursive Doubling Type by Supercompilation with Neighborhood Embedding*

Arkady V. Klimov

Institute of Design Problems in Microelectronics, Russian Academy of Sciences, Moscow, Russia
arkady.klimov@gmail.com

Abstract. Recursive doubling (RD) is a well-known family of parallel algorithms that have $O(Log(N))$ parallel time when applied to vectors of length N. They are used to solve problems like Reduction, Scan (partial sums), computing some recurrences, solving tridiagonal linear system etc. Typically, in textbooks, these algorithms are explained and derived each in their own way, ad-hoc, based on the author's ingenuity. The paper suggests a general way to derive each such RD algorithm from a raw specification by an equation of the form $Y = F(Y)$, in which no idea of the RD principle is pre-contained. First, the base process P is defined, in which the general RD method is applied to the computation of the fixpoint of the specifier function F. Then, the supercompilation is applied to the base process P, eliminating all redundancies and producing a particular efficient RD code.
All necessary definitions in the area of supercompilation are presented. The supercompiled graph in all our examples has just a single loop that is built by looping strategy based on neighborhood embedding. The supercompilation process is carried out manually, all the steps being presented in the paper. The application of the method to three well-known problem examples is demonstrated.

Keywords: recursive doubling, parallel algorithms, SIMD, specification, fixpoint, supercompilation, neighborhood analysis, neighborhood embedding, program synthesis.

1 Introduction

Recursive doubling (RD) is a well-known family of parallel algorithms that have $O(Log(N))$ parallel time when applied to vectors of length N. They are used for problems like Reduction, Scan (partial sums), computing some recurrences, solving tridiagonal linear system etc. This method is also known as *cascade*, or *pairwise*, *summation* (usually for partial sums) or *cyclic reduction* (for linear

* Supported in part by the budget funding and the RAS Presidium Program № I.33P on strategic directions of science development in 2016.

recurrences and tridiagonal systems) [2, 11]. With our approach they all appear to be the same, though in [2] they are presented as different methods.

Usually the RD method requires to reveal an associative binary operation, using which the problem could be reformulated and reduced to calculation of a convolution (or partial sums) of some vector with this operation. However, this operation and the respective reformulation of the problem are often difficult to guess.

We claim that these RD algorithms can be systematically derived from a problem specifications given in the form of equations like $Y = F_X(Y)$, where X is an input and Y the result. First, a general but inefficient method of computing the fixpoint solution is considered. It is the process P of symbolic evaluation of $(F_X)^n$, which leads to solution of the form $Y = (F_X)^n(\perp)$, where $\perp$ means "unknown", and n is large enough for the right hand side to compute to a result vector. To obtain $(F_X)^n$ for sufficiently large n we apply recursive doubling scheme which requires $Log(n)$ doubling steps.

However, computing this way immediately is rather expensive. To improve the efficiency, it is a good idea to apply supercompilation to the process P which would result in an efficient parallel SIMD-like program solving the original problem. We show the work of this idea for three classic examples of problems for which algorithms operating on RD principles are known. We perform the supercompilation process manually presenting all its steps in the paper. The technique has not been implemented in the computer yet.

The supercompiled program graph in all our examples has just a single loop that is built by looping strategy based on the so-called *neighborhood* embedding. The uniqueness of this strategy is that it relies on the computing process as a whole rather than just on the states the process produces.

Though all our result algorithms are textbook ones, the contribution of the paper is a general method to derive them systematically from very simple and evident problem specifications. In addition, the paper fills the gap in publications on the usage of neighborhood analysis in supercompilation, as it was proposed by V.Turchin [9, 10].

The paper is organized as follows. In Section 2 a general idea of supercompilation is briefly presented. The concepts of *driving*, *computation tree*, *configuration graph*, *generalization*, *looping*, *whistle*, and *neighborhood strategy* are introduced, which will be needed below. In Section 3, the common pattern of RD code derivation from a tasks specification is presented. In Section 4, this pattern is applied to 3 distinct problem examples. Section 5 presents evaluation of the topic, identifying our contribution, and the last section concludes the paper.

2 Basics of the Supercompilation

For the purpose of this paper below we describe supercompilation briefly. For more systematic presentation the reader may refer to [3, 5, 6, 8–10].

The word supercompilation is made of two parts: *super* and *compilation*. The name reflects the basic idea that a program is compiled by a supervision over

the process of its generalized computation. Such computation is performed over not a concrete data but rather over generalized data or states (both input and intermediate) called *configurations* and represented with symbolic configuration variables (*c-variables*) indicating unknown parts of state information. The dual nature of a configuration is that, on the one hand, it is a syntactic objects composed of c-variables, constants and constructors and, on the other hand, it can be regarded as a set of concrete states. Therefore, the configurations can be compared for set-theoretic inclusion, which usually corresponds to a syntactic embedding by means of a substitution.

A generalized computation is called *driving* and involves the following three types of actions:

1. If normal computational process assumes a choice based on current values of some c-variables, the driving process splits into two *branches* with the indication of the predicate, generating the splitting. Accordingly, for each of the branches the set of possible values for these variables is narrowed. A variable narrowed to a constant can be replaced with the constant. A variable narrowed to a structure can be split into several variables corresponding to structure fields.
2. If the process executes an operation forming a new data item from old ones, then an updated configuration with a new c-variable is created, and the arc leading to it from the old one is labeled with the substitution for this variable of an expression specifying the method of calculating its value from existing c-variables.
3. The process can perform a state reconstruction not linked with the emergence of a new data (for example, moving a structure field). Then a new node appears in the tree with a new configuration comprised of only old c-variables. None labels are placed on the arc to it. Such configuration (the source one) is called *transit*, and further it can be removed.

These three types of actions being applied iteratively to unfinished configurations produce a tree of generalized states whose branches represent generalized computation paths. These branches evolve independently of each other. Some of them may be (potentially) infinite. Now we are to fold the infinite tree into finite graph, representing the same set of paths.

To achieve this, each time a new configuration appears in the process of driving a *looping strategy* is applied which compares the new configuraton with all old ones (or their parts) seeking for their admissible generalizations into which the new one could be embedded. In case of successful embedding without extra generalization the branch evolution stops and an arc from the new configuration to the old one labeled with unifying substitution is drawn.

If a generalization is needed, the strategy decides whether to generalize and loop now or to continue driving. In the former case (generalization) the arc is drawn from the old configuration to the generalized one with the needed substitutions and all the subtree growing from it is rebuilt anew. If you are lucky, another new configuration would embed into the generalized old one, otherwise

we repeat the search. (The case of embedding into a part of an old configuration is not considered, as it is useless here. Just mention that a pruning is done and the sub-graph is built that forms a subroutine).

Choosing a strategy implies a compromise between the desire to have residual program smaller, and the desire to perform more work statically. For example, if you allow to make any possible generalization, the process will complete quickly with a small residual graph, but most of actions will remain in it. This is a *conservative* strategy. Yet if you hold from looping, more actions will be supercompiled out, but the process of supercompilation can go to infinity. Such strategy is called *aggressive*.

Often some rather aggressive strategy is used, and above it a so-called *whistle* is introduced, which occasionally generates a signal forcing a loop.

When and if the process is completed, the final graph is converted to a residual program. Wherein:

1. All elements except variables are erased in graph node configurations. The variables represent local data available at the current program point.
2. Substitution on the arc converts to assignment statements.
3. Branching converts to conditional statement.

The described process is usually carried out automatically by the supercompiler, as a program that directs the computation process and an initial configuration are given. In our case, the "program" is the general scheme of the recursive doubling method for computing a fixed point, the "initial configuration" is the task specification in the form of equation, and the process of "supercompilation" is carried out manually, imitating the work of the supercompiler.

For generalizations and looping the neighborhood strategy will be used. If a few driving steps were made from a configuration, then a *neighborhood* of this configuration is a set of configurations that includes this one together with all those from which the same steps would be made. The neighborhood (more precisely, its approximation from below) may be obtained by replacing unused parts of initial configuration with metavariables. Then, reducing metavariables to normal configuration variables, we obtain a generalization of the configuration, using which we keep all action performed earlier. A loop is made only when a new configuration (or rather its corresponding generalization) is embedded into a neighborhood of one of the previous configurations. This strategy does not guarantee the termination, but in our cases it is sufficient.

In essence, there are two major components in the supercompilation. The first is a formal inference rule, equivalent to the mathematical induction, which is provided by the mechanism of looping on condition of configurations embedding. The other component is a strategy, in our case the neighborhood one, which is nothing more than a heuristic for detecting inductive hypotheses. Together, these two components provide a fully algorithmic "intelligence" for algorithm transformations.

The idea of neighborhood strategy for supercompilation was proposed by V. Turchin [9,10]. The formalization of the concept of neighborhood and neighborhood analysis in the broader context was carried out by S. Abramov [1].

3 General Scheme of Algorithm Derivation

As an initial "language specification" we shall use the language of recursive equations. If the input is denoted by X and the output (including, possibly, interim results) by Y the specification takes the form

$$Y = \mathcal{R}(X, Y) \tag{1}$$

where $\mathcal{R}$ is an expression with variables X and Y. For a given X, equation (1) can be rewritten as:

$$Y = \mathcal{R}_X(Y) \tag{2}$$

Let us assume that the domains of X and Y have complete partial order structure and that all functions used in $\mathcal{R}$ are monotonic. Then the solution of equations of the form (2) can be written as the limit

$$Y = \lim_{n\to\infty} (\mathcal{R}_X)^n(\bot) \tag{3}$$

where the degree is the n-fold superposition of the function. In practice, one may take the value of $(\mathcal{R}_X)^n$ for some finite n, starting from which the sequence is constant. It will be the case if, e.g., expression $(\mathcal{R}_X)^n$ does not contain the argument variable Y.

For our purposes it will be enough to take the domain of vectors of length N with elements of a type T extended with value $\bot$, such that $a \succeq \bot$ for all $a \in T$. The element $\bot$ means *undefined* and relation $\succeq$ means *more (or equally) exact*. All operations ϕ over type T are extended by the rule $\phi(\ldots, \bot, \ldots) = \bot$ and on the vectors are defined element-wise. The vector $\bot = [\bot, \ldots, \bot]$. In addition, we'll need the shift operator for vectors. Consider vector $X = [x_1, x_2, \ldots, x_N]$. Then for $k \geq 0$ by definition we have

$$\begin{aligned} \mathsf{shift}(\ \ k, X) &= [0, \ldots, 0, x_1, \ldots, x_{N-k}], && (k \text{ zeros from left}) \\ \mathsf{shift}(-k, X) &= [x_{k+1}, \ldots, x_N, 0, \ldots, 0], && (k \text{ zeros from right}) \end{aligned}$$

As an example, consider a simple problem: calculation of the vector B of the partial sums of vector A. It can be defined by the following equation:

$$B = \mathcal{R}_A(B) = \mathsf{shift}(1, B) + A \tag{4}$$

Indeed, it is easy to see that

$$(\mathcal{R}_A)^k(\bot) = [a_1, a_1 + a_2, \ldots, a_1 + a_2 + \cdots + a_k, \bot, \ldots, \bot] \tag{5}$$

and therefore for $k \geq N$ the formula (5) yields the desired result.

It is clear that to calculate $(\mathcal{R}_A)^N(\bot)$ immediately by N-fold applicaton of $\mathcal{R}_A$ is very expensive. Therefore, we first calculate the N-th degree of $\mathcal{R}_A$ using the doubling method:

$$\begin{aligned} &\mathcal{F}_1 = \mathcal{R}_A \\ &\mathcal{F}_2 = \mathcal{F}_1 \circ \mathcal{F}_1 \\ &\mathcal{F}_4 = \mathcal{F}_2 \circ \mathcal{F}_2 \\ &\ldots \end{aligned} \tag{6}$$

The process (6) may stop if an expression $\mathcal{F}_{2^n}$ does not contain symbol Y (or it becomes clear that the limit is achieved). Then the desired result is $\mathcal{F}_{2^n}(\perp)$. This explains the type name of produced algorithms: *recursive doubling*.

If we perform the superposition operation $\circ$ formally, then no effect will take place, because the calculation of $F_{2^n}(\perp)$ will still have to make 2^n calls to initial function $\mathcal{R}_A$. However, making formula manipulation each step is also expensive. Thus, it is a good idea to supercompile the process (6) with unknown input A, such that only calculations dependent of A remain for dynamic execution.

As the supercompilation would only make sense for a particular problem $\mathcal{R}_A$, we give here only general comments on the scheme, and the full description of the supercompilation process will be set forth by specific examples in Section 4.

The initial configuration is the equation of the form $Y = \mathcal{R}_X(Y)$, in which X is a configuration variable indicating the input and Y is just a formal symbol. Performing the step is to move to the configuration $Y = \mathcal{R}_X(\mathcal{R}_X(Y))$, in which various simplifications are produced. Herein are used the usual properties of arithmetic operations and the following properties of the shift operator:

$$\mathsf{shift}(k, A \odot B) = \mathsf{shift}(k, A) \odot \mathsf{shift}(k, B) \tag{7}$$

$$\mathsf{shift}(k, \mathsf{shift}(k, A)) = \mathsf{shift}(k + k, A) \tag{8}$$

$$\mathsf{shift}(k, (\mathsf{shift}(k, A)) = \mathsf{shift}(k, \mathbf{1}) \cdot A, \textit{where } \mathbf{1} = [1, 1, \ldots, 1] \tag{9}$$

$$\mathsf{shift}(k, X) = \mathbf{0}, \textit{if } k \geq N \tag{10}$$

where $\odot$ is any operation on elements (such that $0 \odot 0 = 0$).

To provide the termination property of residual program the supercompiler must recognize formula that does not depend on Y. To this end, based on the property (10), we will act according to the following rule: when the formula has a sub-expression of the form $\mathsf{shift}(k, X)$, we introduce the splitting predicate $k \geq N$ and on the positive branch replace this sub-expression with $\mathbf{0}$. This will lead to possible exclusion of the argument Y from configurations on some branches, which then can become terminal.

The supercompilation will be effective if we find a common form for the steps. For doubling, it will be possible if we succeed to represent the configuration after the step in the same form as before the step, in which case the loop is possible. In case of success, the final residual graph is converted to the code in a programming language. The code will hold element-wise vector operations, which are easily parallelizable.

4 Examples

4.1 Linear Recurrence

This is a generalization of example (4). Given are two vectors A and B of length N. We are to solve the following equation

$$Y = A * \mathsf{shift}(1, Y) + B \tag{11}$$

In a more familiar mathematical language, this is equivalent to the calculation of the sequence $\{y_i\}$ defined by the following recurrence:

$$\begin{aligned} y_0 &= 0 \\ y_i &= a_i * y_{i-1} + b_i \ , \quad i > 0 \end{aligned} \tag{12}$$

Let us supercompile the computation process for solving equation (11) by doubling method. The initial configuration C0 is the original equation:

C0: $$Y = A * \mathsf{shift}(1, Y) + B$$

The doubling step is to substitute for the variable Y in the right-hand side with the right hand side itself and simplify:

C1: $$\begin{aligned} Y &= A * \mathsf{shift}(1, A * \mathsf{shift}(1, Y) + B) + B \\ &= [A * \mathsf{shift}(1, A)] * \mathsf{shift}(1 + 1, Y) + [A * \mathsf{shift}(1, B) + B] \end{aligned}$$

It is easy to notice the similarity of this configuration with the original one. But how can the machine "notice" this? Formally, it is necessary to represent both configurations as special cases (by substitutions) of a common configuration. But which generalizations should be considered valid? An answer may be given by neighborhood analysis. We carry out a step from configuration C0 to configuration C1, watching for what information from C0 representation we use. To this end, we cover the description of the configuration C0 with a translucent strip, and whenever a symbol is explicitly used in the process, we make it out of the strip. On step completion only the symbols A, B and 1 remain intact:

U0: $$Y = \boxed{A} * \mathsf{shift}(\boxed{1}, Y) + \boxed{B}$$

This means that no matter what could be in their place, the step would be carried out in the same way. Note that these parts can go without changes into new configurations after the step:

U1: $$Y = [\boxed{A} * \mathsf{shift}(\boxed{1}, \boxed{A})] * \mathsf{shift}(\boxed{1} + \boxed{1}, Y) + [\boxed{A} * \mathsf{shift}(\boxed{1}, \boxed{B}) + \boxed{B}]$$

The result of the driving step with neighborhood is shown in Fig. 1 (left). There appeared a split due to property (10) and new configurations on the arc ends. Intact parts of the original configuration are shaded.

Now we can replace all the unused parts with metavariables. Let it be $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{K}$. Considering all sorts of substitutions for them, we get a variety of configurations for which the driving step is identical to step from this configuration. In terms of metavariables it is a step from metaconfiguration

M0: $$Y = \mathcal{A} * \mathsf{shift}(\mathcal{K}, Y) + \mathcal{B}$$

to metaconfiguration

M1: $$Y = [\mathcal{A} * \mathsf{shift}(\mathcal{K}, \mathcal{A})] * \mathsf{shift}(\mathcal{K} + \mathcal{K}, Y) + [\mathcal{A} * \mathsf{shift}(\mathcal{K}, \mathcal{B}) + \mathcal{B}]$$

Thus, as a result of the neighborhood analysis a generalized configuration M0 has been built, which will be considered as a maximum permissible generalization of the initial configuration C0, to which the looping is feasible.

Recall that the rule (10) for M0 yields the predicate $k \geq N$ with the branch to the final metaconfiguration $Y = B$. Metaconfiguration M1 on the other branch should be driven further as it contains Y. But first we must check whether it falls into the processed one (M0). We see that it does, and therefore we may not proceed doing the next step but make a loop into the generalized old configuration M0. To this end, in the neighborhood M0 we replace metavariables with usual configuration variables such that both old and new configurations C0 and C1 can be obtained from it by substitutions. Those substitutions become labels on the arcs into the new generalized configuration. The result is shown in the right part of Fig. 1.

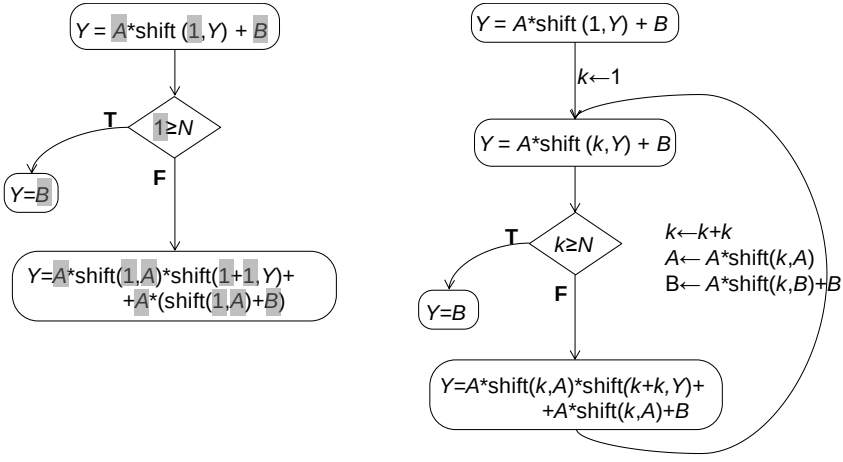


Fig. 1. Construction of the graph configurations in the Linear Recurrence. To the left is the first doubling step with its neighborhood. Untouched parts of the initial configuration are shaded. In their place, new variables are introduced (if they did not exist yet). To the right is the final graph after generalization and looping.

Now we convert the resulting graph into the program in a programming language: erase configurations (except the final) and replace the substitutions with assignment operators (changing their order and introducing interim variables if necessary). The result is shown in Fig. 2. It is a simple cyclic program built of parallel vector operations. It is easy to see that the loop performs Log (N) iterations, each taking an O(1) parallel time.

Interestingly, if the size of the vector N were a given constant, say 1000, the first driving step would lead to the lack of a lateral branch, as predicate $1 \geq 1000$ yields known **false**. So, the value of 1 would have been used, and therefore there

```
k:=1;
while k<N do
   B:=A*shift(k,B)+B;
   A:=A*shift(k,A);
   k:=k+k;
end;
Y:=B;
```

Fig. 2. The generated code for the Linear Recurrence

would be no embedding. The neighborhood strategy in this case would lead to a complete loop unfolding (here 10 rounds).

4.2 Binary Addition

Consider the task of adding two binary integers A and B. The desired result is a scheme with fast carry, which works for the $Log(n)$ clock cycles, where n is the bit size of summands.

We begin with the following system of recurrent equations defining the vectors carry C and sum S:

$$\begin{aligned} C &= A \wedge B \vee (A \vee B) \wedge \ shift(1, C) \\ S &= A \oplus B \oplus \mathsf{shift}(1, C) \end{aligned} \tag{13}$$

where $\wedge$, $\vee$, $\oplus$ are bitwise conjunction, disjunction and addition modulo 2 respectively ($\wedge$ has the highest priority).

Let us solve the equation for C by doubling. The initial configuration is

C0: $$C = A \wedge B \vee (A \vee B) \wedge \mathsf{shift}(1, C)$$

We do not show the step result (configuration C1) as it is bulky and not interesting. For configuration C0 we get the following neighborhood:

U0: $$C = A \wedge B \vee (A \vee B) \wedge \mathsf{shift}(1, C)$$

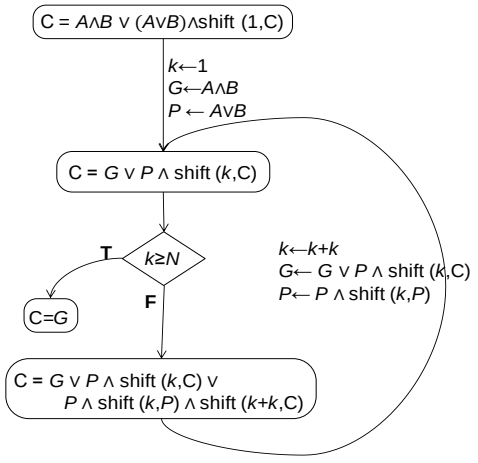
Replace unused parts (selected by gray background) with metavariables, let it be $\mathcal{G}$, $\mathcal{P}$ and $\mathcal{K}$. Metaconfiguration

M0: $$C = \mathcal{G} \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, C)$$

goes by the driving step into metaconfiguration

M1: $$\begin{aligned} C &= \mathcal{G} \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, \mathcal{G} \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, C)) = \\ &= \mathcal{G} \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, \mathcal{G}) \vee \mathcal{P} \wedge \mathsf{shift}(\mathcal{K}, \mathcal{P}) \wedge \mathsf{shift}(\mathcal{K} + \mathcal{K}, C) \end{aligned}$$

One can see that metaconfiguration M1 embeds into the initial one M0. Making the needed actions for generalization and looping (structurally they are similar to the previous example, just operations are different), we get the graph shown in Fig. 3.




```
k:=1;
G:= A ∧ B;
P:= A ∨ B;
while k<N do
    G:=G ∨ P ∧ shift(k,G);
    P:=P ∧ shift(k,P);
    k:=k+k;
end;
C:=G;
S:= A ⊕ B ⊕ shift(1,C);
```

Fig. 3. To the left is the finished configuration graph for Binary Addition problem. To the right is the corresponding program "with fast carry".

4.3 Solving tridiagonal linear equations

We write the tridiagonal system of equations as

$$X = A \cdot \mathsf{shift}(1, X) + B \cdot \mathsf{shift}(-1, X) + C \tag{14}$$

where A, B, C are input vectors and X is the vector of unknowns.

A distinctive feature of this problem is that the doubling step transforms not only the right hand side, but the entire equation. The initial configuration is:

C0: $$X = A \cdot \mathsf{shift}(1, X) + B \cdot \mathsf{shift}(-1, X) + C$$

We perform the step and simplify the result with the use of the property (9):

C1:
$$\begin{aligned} X = {} & A \cdot \mathsf{shift}(1, A \cdot \mathsf{shift}(1, X) + B \cdot \mathsf{shift}(-1, X) + C) + \\ & B \cdot \mathsf{shift}(-1, A \cdot \mathsf{shift}(1, X) + B \cdot \mathsf{shift}(-1, X) + C) + C \\ = {} & A \cdot \mathsf{shift}(1, A) \cdot \mathsf{shift}(1 + 1, X) + \\ & B \cdot \mathsf{shift}(-1, B) \cdot \mathsf{shift}(-(1 + 1), X) + \\ & A \cdot \mathsf{shift}(1, C) + B \cdot \mathsf{shift}(-1, C) + C + \\ & A \cdot \mathsf{shift}(1, B) \cdot X + B \cdot \mathsf{shift}(-1, A) \cdot X \end{aligned}$$

By grouping linear on X terms to the left and dividing both parts by the coefficient of X, we reduce the configuration C1 to the following form:

C1′:
$$\begin{aligned} X = {} & (A \cdot \mathsf{shift}(1, A))/D \cdot \mathsf{shift}(1 + 1, X) + \\ & (B \cdot \mathsf{shift}(-1, B))/D \cdot \mathsf{shift}(-(1 + 1), X) + \\ & (A \cdot \mathsf{shift}(1, C) + B \cdot \mathsf{shift}(-1, C) + C)/D, \end{aligned}$$

where $D = 1 - A \cdot \mathsf{shift}(1, B) - B \cdot \mathsf{shift}(-1, A)$. Consider the neighborhood U0 of initial configuration, where all intact parts are shaded:

U0: $$X = \boxed{A} \cdot \mathsf{shift}(\boxed{1}, X) + \boxed{B} \cdot \mathsf{shift}(-\boxed{1}, X) + \boxed{C}$$

Note that the two occurrences of the number 1 were touched only by comparing them with each other (through the use of property (9)), so, in the neighborhood expression M0, they should be represented with the same metavariable $\mathcal{K}$.

M0: $$X = \mathcal{A} \cdot \mathsf{shift}(\mathcal{K}, X) + \mathcal{B} \cdot \mathsf{shift}(-\mathcal{K}, X) + C$$

After the step, at both corresponding locations in metaconfiguration M1 occurs the same sub-expression $(\mathcal{K} + \mathcal{K})$, which allows to embed.

M1: $$\begin{aligned} X =& (\mathcal{A} \cdot \mathsf{shift}(\mathcal{K}, \mathcal{A}))/D \cdot \mathsf{shift}(+\mathcal{K}, X) + \\ & (\mathcal{B} \cdot \mathsf{shift}(-\mathcal{K}, \mathcal{B}))/D \cdot \mathsf{shift}(-(\mathcal{K} + \mathcal{K}), X) + \\ & (\mathcal{A} \cdot \mathsf{shift}(\mathcal{K}, \mathcal{C}) + \mathcal{B} \cdot \mathsf{shift}(-\mathcal{K}, \mathcal{C}) + \mathcal{C})/D, \end{aligned}$$

where $D = 1 - \mathcal{A} \cdot \mathsf{shift}(\mathcal{K}, \mathcal{B}) - \mathcal{B} \cdot \mathsf{shift}(-\mathcal{K}, A)$ Thus, the metaconfiguration M1 completely embeds into the neighborhood metaconfiguration M0, which results in a final graph on Fig. 4. After erasing all the excess, we obtain the residual program shown in Fig. 5.

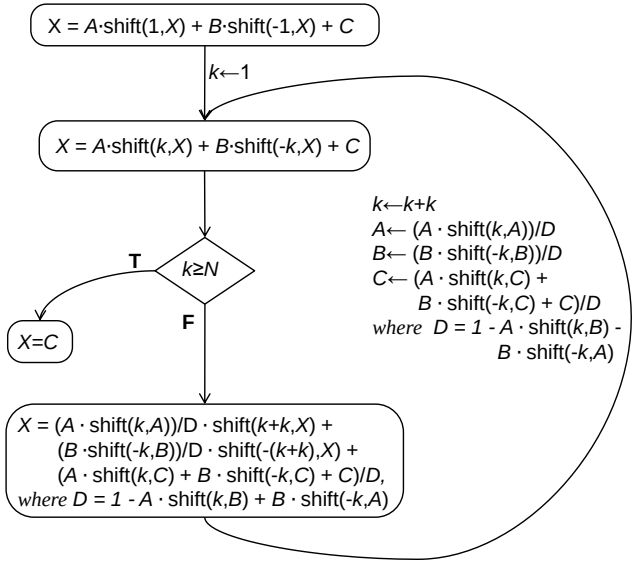


Fig. 4. Configuration graph for solving tridiagonal system of equations

```
k:=1;
while k<N do
   D:=1A*shift(k,B)B*shift(k,A);
   C:=(A*shift(k,C)+B*shift(k,B)+C)/D;
   A:=(A*shift(k,A))/D;
   B:=(B*shift(k,B))/D;
   k:=k+k;
end;
X:=C;
```

Fig. 5. The generated code for solving tridiagonal system of equations

5 Related Work

As was mentioned in the Introduction, all algorithms derived here are well known and described in textbooks [2, 11]. And though the rationale for the algorithm for the Linear Recurrence problem in [2] is meaningfully the same as ours, it is described there in the style of common reasoning, and the Fortran code (which is exactly the same as in Fig. 2) is given just as a hand-written implementation. On the contrary, in our paper we demonstrate the possibility of mechanical derivation using a supercompilation mechanism. To the best of our knowledge, no existing compiler is able to generate parallel code based on recursive doubling, unless an associative operation for functions Scan or Reduce is given explicitly.

An interesting and rather complicated case of algorithm based on doubling is considered by B. Steinberg [7]. He explores the possibility of calculating by doubling method the recursion of the form:

$$Y_i = if\ P_i(Y_{i-1})\ then\ A_i\ else\ F_i(Y_{i-1}),$$

where A is a given numerical vector, and P, F are functions, that depend on index i as well as on the main argument. The author presents conditions on functions P, F and vector A, in which it is possible, and the respective theorem is proved. All is presented in the style of traditional mathematics, relying on the author's ingenuity. We hope that, by means of a supercompiler with a well-developed system of formula transformations, a similar result including the necessary restrictions on input functions can be obtained automatically.

6 Conclusion

The algorithms derived here are well known and described in the textbooks [2,11], but their explanation and/or derivation is typically based on conjectures, intuition and special considerations. Here we suggest a general method to systematically derive them from simple definitions presented in the form of recursive

equations like $Y = F(Y)$. Then, a general recursive doubling method for calculating the fixed point is applied. This process is subject to execution under the control of a supercompiler. As a result, an efficient parallel code is generated in a language with parallel vector operations. Its execution takes $O(log(N))$ parallel time, where N is the size of the problem. We do not give a detailed analysis of their performance, as there are a lot publications about that. The purpose of the paper is to present the very method of obtaining algorithms. The method was announced by the author in 1988 [4]. Now, this paper reveals the details.

In the paper the supercompilation process is carried out manually, but following certain rules. The paper shows the techniques needed in a compiler for automated parallelization of a kind of loops. Of particular importance is the tool for formula transformations, which should allow to convert a formula to a certain pattern that is itself created in the same process.

References

1. Sergei M. Abramov. *Metavychislenija i ikh prilozhenija (Metacomputation and its applications).* Nauka, Moscow, 1995. (In Russian).
2. R.V. Hockney and C.R. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, Ltd., Bristol, UK, 1981.
3. 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.
4. Arkady V. Klimov. Ob odnom metode polucheniya bystrodejstvujuschih parallelnyh algoritmov (on a method to obtain high performance parallel algorithms). In *Semioticheskie aspekty formalizacii intellektualnoj deyatelnosti. All-union school-seminar Borzhomi-88. Tezisy dokladov i soobschenij*, pages 53–56, Moscow, 1988. URL: http://agora.guru.ru/abrau2009/pdf/238_NSSI_2009_Abrau-2009.pdf.
5. 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.
6. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
7. B.J. Steinberg. Rasparallelivanie rekurrentnyh tsiklov s uslovnymi operatorami (parallelizing recurrency loops with conditionals). *Avtomatika i telemekhanika*, 56(9):176–184, 1995.
8. 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.
9. Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
10. Valentin F. Turchin. The algorithm of generalization in the supercompiler. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531–549. North-Holland, 1988.
11. V.V. Voevodin. *"Matematicheskie modeli i metody v parallelnyh protsessah" (Mathematical models and methods in parallel processes).* Nauka, Moscow, 1986.

A Supercompiler Assisting Its Own Formal Verification

Dimitur Krustev

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

Abstract. Program verification is a well known potential application of supercompilation. There are, however, few examples of using supercompilation for practical verification problems. We consider the correctness proof of the supercompiler itself as an interesting and practical task, on which to test the potential of supercompilation in this area. We show that even a simple supercompiler – treating a small first-order subset of Lisp, and working in cooperation with a traditional proof assistant (J-Bob) – can provide a lot of help for checking its own correctness.

1 Introduction

Supercompilation is a program transformation technique – a particularly strong form of partial evaluation [4,8] – originally proposed by Turchin [24], with a long history [12] and many potential applications: program optimization, software testing, program analysis, formal verification. While many of these applications were already described by Turchin in his early works, most of the research on supercompilation has concentrated for many years on program optimization. In recent years researchers show renewed interest in other applications, including analysis and verification of programs and other systems [11, 13, 18, 19]. A good way to increase the adoption of supercompilation for verification is to demonstrate it is applicable and helpful for a wider range of practical problems. By analogy with self-application of supercompilers (and partial evaluators in general) – which has been a hot research topic for many years [4, 20] – we consider verification of the supercompiler itself to be an interesting task. A challenge in the context of verification is how to trust the results of a supercompiler, especially if it is being used for its own verification. A good solution to this problem has recently been proposed by Klyuchnikov et al. [14]: a supercompiler producing not only a transformed program, but also a certificate proving the result is semantically equivalent to the input.

We take this idea a step further by showing that a supercompiler – which is both self-applicable and certifying – can be used successfully as a proof automation tool during the creation of its own correctness proof. Note that we do not speak of fully automatic self-verification: the supercompiler automatically generates only parts of its correctness proof. The overall proof is created manually in a traditional proof assistant. We argue that this organization is actually

an advantage: while typically supercompilers are used as automatic black-boxes, and if they fail to produce the desired result, their user is left with nothing, in our approach the supercompiler can be called many times during an interactive proof session, each time incrementally helping to get closer to the final goal. We outline as particular achievements of the proposed approach the following:

- a way to construct a supercompiler, which is both self-applicable and certifying (Sect. 3);
- an alternative two-phase approach to the construction of a certifying supercompiler: the supercompiler returns only a certificate, and the resulting program can be reconstructed from this certificate as a second step (Sect. 3.1);
- successful application of the described supercompiler as a generic proof automation tool in the context of a traditional proof assistant (Sect. 3.1);
- successful application of the supercompiler to fill in parts of its own correctness proofs (Sect. 4.2).

In Sect. 3 we discuss the overall architecture of our supercompiler and the adaptations made – compared to a classical supercompiler organization – in order to fit better as a proof automation tool inside the J-Bob proof assistant. One important omission is the lack of folding (that is, the supercompiler cannot produce new function definitions). It turns out folding is of no critical importance for our particular application domain, while its lack simplifies a lot both the supercompiler and its correctness proof. Section 4 discusses the more interesting details of the implementation itself, as well as some of the tricks used to simplify further the correctness proof. In Sect. 5 we analyze the performance of the proposed system in two aspects. First, we use the ratio of high-level proof steps (as entered manually by the user) versus low-level proof steps (obtained after the application of supercompilation) as a measure of the level of proof automation the system provides. Second, we analyze the processing time requirements of the overall system (supercompiler + underlying proof assistant), and how they can be improved.

We assume readers are familiar with supercompilation [13, 22, 24], but not necessarily with J-Bob. As J-Bob's principles of operation are important for understanding the rest of the article, we briefly review them in the following section.

2 J-Bob Crash Course

For this supercompiler verification experiment we use a proof assistant called J-Bob. It has been recently published [2] to accompany the new book "The Little Prover" [3] – a gentle introduction to program verification and proof assistants in general. J-Bob can be used to check proofs of properties of programs written in a first-order purely functional subset of Lisp. While this Lisp dialect contains only a few built-in data types and a handful of primitives, it can still be used to

write interesting and complicated programs. For example, J-Bob itself is written in the same Lisp subset it can reason about. Programs are a sequence of function definitions (ordered by the definition-use relation), with their bodies being expressions of 4 syntactic categories (Fig. 1): variables, constants, conditional expressions, and function calls (to both built-in primitives and defined functions). Only direct recursion is allowed and recursive function definitions must be accompanied by a proof of termination. The built-in data types consist only of atoms (natural numbers and symbols) and `cons` pairs. The built-in operations are: `equal`, `atom`, `car`, `cdr`, `cons`, `natp`, `size`, `+`, `<`. As a small example the function for concatenating 2 lists can be written as in Fig. 2.

$$
\begin{aligned}
\mathit{Exp} \ni e &::= x \mid \texttt{'}c \mid \texttt{(if } e_q\ e_a\ e_e\texttt{)} \mid \texttt{(}f\ e_1 \ldots e_n\texttt{)} \\
\mathit{Def} \ni \mathit{def} &::= \texttt{(defun } f\ \texttt{(}x_1 \ldots x_n\texttt{)}\ e_{body}\texttt{)} \\
\mathit{Prg} \ni p &::= \mathit{def}_1 \ldots \mathit{def}_n
\end{aligned}
$$

Fig. 1. Lisp syntax

```
(defun append (xs ys)
  (if (atom xs) ys
    (cons (car xs) (append (cdr xs) ys))))
```

Fig. 2. List append in Lisp

Unlike many other proof assistants J-Bob does not make a distinction between logical statements and Boolean expressions: logical statements *are* represented as Boolean expressions[1]. This feature is particularly useful for our purposes, as supercompilation deals easily with program expressions (including Boolean ones), but not with logical statements. The if-expression serves as the only built-in Boolean operation, but other operations are easily expressible, for example:

$$
\begin{aligned}
a \wedge b &\equiv \texttt{(if } a\ b\ \texttt{'nil)} \\
a \vee b &\equiv \texttt{(if } a\ \texttt{'t}\ b\texttt{)} \\
a \rightarrow b &\equiv \texttt{(if } a\ b\ \texttt{'t)}
\end{aligned}
$$

[1] As Lisp is dynamically typed, there is no static distinction between Boolean-valued and other expressions. By "Boolean expression" we mean an expression, which always results in a Boolean value in all dynamic contexts

As J-Bob's logical statements are just Boolean expressions, its proofs are actually program transformations whose goal is to transform the expression representing a given theorem statement into the constantly true expression (`'t`). Namely, a J-Bob proof is a sequence of steps, each step being one of three kinds:

- rewriting – possibly conditional – based on some equality known to be true (either an axiom or a previously proved theorem);
- unfolding or folding of a defined function;
- evaluation of a built-in function called with constant arguments.

Each step is a pair consisting of:

- a path defining a subexpression;
- a transformation to perform on this subexpression.

Paths are just sequences of navigation steps inside compound expressions: the single-letter atoms `Q`(uestion), `A`(nswer), and `E`(lse) are used to select one of the 3 subexpressions of an if-expression and natural numbers (starting from 1) index the arguments of a function call.

As an example we can consider the proof that the list append function is associative:

```
((dethm append-assoc (xs ys zs)
    (equal (append (append xs ys) zs) (append xs (append ys zs))))
    (list-induction xs)
    ((A 1 1) (append xs ys))
    ((A 1 1) (if-nest-A (atom xs) ys (cons (car xs) (append (cdr xs) ys)
        )))
    ((A 2) (append xs (append ys zs)))
    ((A 2) (if-nest-A (atom xs) (append ys zs) (cons (car xs) (append (
        cdr xs) (append ys zs)))))
    ((A) (equal-same (append ys zs)))
    ...
    (() (if-same (atom xs) 't))
)
```

The proof starts with an indication that we are to proceed by induction on argument `xs`. At that point (if we have not added other steps to the proof yet), J-Bob presents us with a proof obligation based on the body of the theorem and the selected induction scheme:

```
(if (atom xs)
  (equal (append (append xs ys) zs) (append xs (append ys zs)))
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
      ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

In the base case when `xs` is an atom, we must prove the statement directly, otherwise we can use the induction hypothesis `(equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append ys zs)))` inside the proof. The first step of the actual proof – `((A 1 1) (append xs ys))` – unfolds the corresponding occurrence of `append` and the current expression becomes:

```
(if (atom xs)
  (equal (append (if (atom xs) ys (cons (car xs) (append (cdr xs) ys)))
      zs) (append xs (append ys zs)))
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
      ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

We notice the nested occurrence of a check for `(atom xs)`, which is redundant. We eliminate it with the next proof step – ((A 1 1) (if-nest-A (**atom** xs)ys (**cons** (**car** xs)(append (**cdr** xs)ys)))) – where `if-nest-A` is an axiom from the J-Bob standard library. The resulting expression is:

```
(if (atom xs)
  (equal (append ys zs) (append xs (append ys zs)))
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
       ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

With 2 analogous proof steps we can also simplify the second call (append xs ...) and we get:

```
(if (atom xs) (equal (append ys zs) (append ys zs))
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
       ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

There is a call to `equal` with identical arguments, which we can simplify with another library axiom in the next step – ((A) (equal-same (append ys zs))):

```
(if (atom xs) 't
  (if (equal (append (append (cdr xs) ys) zs) (append (cdr xs) (append
       ys zs)))
    (equal (append (append xs ys) zs) (append xs (append ys zs))) 't))
```

At this point the answer-arm of the outermost if-expression (which corresponds to the base case of the induction) has been reduced to `'t`. After 12 more proof steps (not shown for brevity), we also reduce the else-arm to `'t`:

```
(if (atom xs) 't 't)
```

The last step of the proof – (() (if-same (**atom** xs)'t)) – reduces this last expression to `'t`, which completes the proof. We have shown – by using an induction scheme plus a sequence of elementary program transformations – that the statement of the theorem will always evaluate to `'t`, no matter what arguments we pass.

3 A Supercompiler for J-Bob

3.1 Overview

The example from the previous section shows that many steps in a typical J-Bob proof are "obvious" reductions of subexpressions. Such proof steps are very tedious to write by hand, especially as J-Bob – being a very minimalistic prover – insists on fully specifying all arguments of the axiom or theorem each step uses. On the other hand many of these steps coincide directly with the steps a typical supercompiler would take when presented with such an expression as input. So our first goal is to make a supercompiler automate – as much as possible – the tedious parts of a J-Bob proof. To support this goal, our supercompiler must return not only the resulting expression, but also the sequence of transformation steps used to convert the input to the output expression, similar to the certifying supercompiler of Klyuchnikov et al. [14]:

$$scp : Prg \times Exp \rightarrow Exp \times Steps$$

If the input expression is the current goal of an unfinished J-Bob proof, we can insert the returned steps inside the proof and change the goal to the final expression produced by the supercompiler – so such an interface to the supercompiler ensures that it can be used directly for J-Bob proof automation. The sequence of transformation steps is actually sufficient to reconstruct the final expression from the original one. Supposing a function for performing transformation steps (which already exists in J-Bob)

$$rewrite : Prg \times Exp \times Steps \to Exp$$

it should hold that:

$$scp(defs, e_1) = (e_2, steps) \to rewrite(defs, e_1, steps) = e_2.$$

This observation permits us to simplify the type of the supercompiler by returning only the necessary transformation steps:

$$scp' : Prg \times Exp \to Steps$$

This approach makes easier the construction of the supercompiler, but also – and more importantly – its correctness proof.

Our second goal is to see if, and to what extent, the proof automation provided by our supercompiler can help in its own correctness proof. Several observations follow from this goal:

- we must formalize the correctness proof inside J-Bob;
- the supercompiler itself must be written in the Lisp dialect that J-Bob can handle;
- the precise formal statement of what it means for the supercompiler to be correct must be compatible with the semantics of J-Bob.

While the first 2 points are trivial, the last one requires some elaboration. J-Bob tacitly assumes that Lisp programs are evaluated in accordance with some semantics, but this semantics is not fully and explicitly described. There are Lisp expressions, about which J-Bob cannot reason. For example, there is no rule in the standard library, which could tell the value of `(if (cons x y) a b)`, (unless both `x` and `y` are constant values), although it is expected that this value is always defined. We sidestep this lack of explicit program semantics by changing what we mean by supercompiler correctness: we simply require that the supercompiler return a valid sequence of transformation steps, which will not get stuck if applied to the original expression. To formalize this definition in a simple and explicit way, we need a modified version of the rewriting function, which also returns an explicit flag if all the steps have been performed successfully:

$$rewrite' : Prg \times Exp \times Steps \to Bool \times Exp$$

We can then define:

$$correct(scp') \equiv \forall defs \forall e(fst(rewrite'(defs, e, scp'(defs, e))) = true)$$

If we assume – or have a separate proof – that J-Bob transformation steps respect the underlying semantics of the language, then the above definition is equivalent to the traditional definition of correctness as semantics preservation.

To keep the supercompiler simple – both as implementation and as correctness proof, we reuse an architecture we have already applied in another simple supercompiler [15]. The implementation is split into layers:

- simplification by basic supercompilation transformations. This layer performs reductions on open expressions, such as if-expressions with a constant condition, or calls to built-in functions with all arguments being constant. This layer also performs if-lifting (to be defined precisely below) and information propagation. No unfolding or folding happens in this layer;
- unfolding layer, using different strategies.

The overall organization of the supercompiler is to perform a sequence of unfolding steps and a subsequence of basic transformations before and after each unfolding:

$$scp = (simplify)* \cdot (unfold \cdot (simplify)*)*$$

The following subsections explain in detail the transformations applied by each layer. They also explain what happens with the other typical supercompiler ingredients – *folding*, *generalization*, and *whistle* – which do not appear explicitly in the architecture as outlined above.

Before we continue, we show how a proof for the same property (list append associativity) can look like:

```
(dethm append-assoc (xs ys zs)
    (equal (append (append xs ys) zs) (append xs (append ys zs))))
    (list-induction xs)
    (scp 5 50)
    (expand ((append 5)) (equal-if (append (append (cdr xs) ys) zs) (
        append (cdr xs) (append ys zs))))
    (scp 0 50)
)
```

The proof has only 3 steps (versus 18 in the manual proof) – 2 calls to the supercompiler (with different options) and 1 manual step. These 3 macro-steps are expanded to 18 steps before being submitted to J-Bob to check – incidentally the same 18 steps as in the manual proof, but in a slightly different order.

3.2 Basic Supercompilation Transformations

The proof example from the previous section has already hinted that many of the elementary program transformations J-Bob performs are similar to those performed by supercompilation. A detailed analysis of all available transformations (mostly in the form of axioms in the standard library) confirms, that we have all the ingredients to simulate supercompiler actions by J-Bob proof steps. Table 1 lists a subset of the axioms suitable for our needs. The name and the definition are given exactly as found in the J-Bob standard library, while the remaining 2 columns illustrate the action of each rule on a typical expression.

We can roughly divide these transformation rules in 3 groups:

Table 1. Transformation rules

Name	Definition	Original expression	Result expression
atom/cons	(**equal** (**atom** (**cons** x y))'nil)	(**atom** (**cons** x y))	't
car/cons	(**equal** (**car** (**cons** x y))x)	(**car** (**cons** x y))	x
cdr/cons	(**equal** (**cdr** (**cons** x y))y)	(**cdr** (**cons** x y))	y
if−true	(**equal** (**if** 't x y) x)	(**if** 't x y)	x
if−false	(**equal** (**if** 'nil x y) y)	(**if** 'nil x y)	y
if−nest−A	(**if** x (**equal** (**if** x y z) y) 't)	(**if** x (... (**if** x y z) ...) ...)	(**if** x (... y ...) ...)
if−nest−E	(**if** x 't (**equal** (**if** x y z) z))	(**if** x ... (... (**if** x y z) ...))	(**if** x ... (... z ...))
if−same	(**equal** (**if** x y y) y)	(**if** x y y)	y
equal−same	(**equal** (**equal** x x)'t)	(**equal** x x)	't

- rules for performing evaluation of open expressions (`atom/cons`, `car/cons`, `cdr/cons`, `if-true`, `if-false`). They correspond to a form of simple partial evaluation.
- rules for nested repeated conditions (`if-nest-A`, `if-nest-E`). Their definition in J-Bob is less obvious, but it simply corresponds to the way J-Bob encodes conditional rewriting rules – as equalities inside if-expressions. These rules correspond to a form of information propagation (both positive and negative) as found in supercompilation.
- rules dealing with duplicated arguments in `equal` and `if` (`if-same`, `equal-same`). With a few exceptions, such transformations are usually not employed by supercompilers, but it is easy to include them in our supercompiler.

If we compare these rules to the basic transformations used in other simple supercompilers [15], there is one ingredient missing: the ability to lift if-expression appearing as conditions of other if-expressions. It turns out, however, that such *if-lifting* can be performed with a combination of existing rules by using the fact, that J-Bob rewriting rules can be applied in both directions:

```
    (if (if q a e) a' e')
=   (if q (if (if q a e) a' e') (if (if q a e) a' e'))   {if-same, right-to-left}
=   (if q (if a a' e') (if (if q a e) a' e'))            {if-nest-A, left-to-right}
=   (if q (if a a' e') (if e a' e'))                     {if-nest-E, left-to-right}
```

Actually, we can use the same trick to lift if-expressions outside of function calls. This is not performed by most supercompilers, because it is not safe in general (it can make a program less terminating). The unsafe step in our derivation above would be the use of `if-same` right-to-left. But J-Bob ensures totality of all expressions, so general if-lifting is a valid transformation in our case.

3.3 Unfolding and Generalization

Unfolding is possible as a basic transformation step in J-Bob, so we are free to add it to our supercompiler. The only question is when and in what order – since J-Bob requires totality proofs for all functions, we are not tied to any particular evaluation order. We can consider the decision whether to unfold a given call or not as a form of generalization. That is why we consider unfolding and generalization together. Experiments with early versions of the supercompiler for performing proofs have shown, that it is useful to have several strategies for unfolding:

- *best unfolding*: We tentatively perform each of the possible unfoldings, simplify the resulting term with the basic transformations from the previous subsection, and select the one resulting in the smallest final expression.
- *call-by-name*: We select the first possible unfolding in leftmost-outermost order.
- *call-by-value*: We select the first possible unfolding in leftmost-innermost order.

During each (extended) proof step, the proof author can decide which strategy is most appropriate, and select it by a switch.

3.4 Folding

As already mentioned, our supercompiler does not perform folding (yet). There are several reasons for this decision:

- Folding seems much less useful for a supercompiler aimed at producing proofs, than in a supercompiler tailored for program optimization or analysis. The reason is that the goal of a J-Bob proof is to reduce a given expression to `'t`. If we introduce new function definitions by folding, it means we have given up any chance to arrive to a final expression equal to `'t`.
- It is not obvious how to integrate arbitrary folding inside a J-Bob proof. J-Bob supports folding steps, but only for existing function definitions. It is not possible to introduce new function definitions on-the-fly inside a proof. Besides, even if there were a way, we would still have to produce a proof of termination for each newly introduced function definition, which is in general a non-trivial task.
- The lack of folding simplifies a lot the correctness proof of the supercompiler. As we describe just a proof-of-concept experiment, whose main goal is to have a supercompiler correctness proof performed with the help of the supercompiler itself, this simplification appears a worthy compromise.

Of course, if we want to use this supercompiler for other tasks beside proof automation, adding support for folding will be highly desirable. We leave this task for future work.

3.5 Whistle

Whistles are critical for the performance of supercompilers aimed at program optimization and analysis. They must not stop the transformation process too early, as opportunities for optimization might be lost. But they should not stop too late either – the supercompiler should not spend too much time producing a bloated result with a lot of duplication, or even fail to stop at all. In our case, however, the supercompiler can be called many times inside a single proof, with different goals in mind. So it is more important to provide better control to the user in the selection of a suitable transformation strategy in each case, than to rely on a sophisticated general whistle. To keep our implementation – and its proof – simple, we have currently settled for a basic whistle using 2 counters: one limiting the total number of unfoldings to perform and one limiting the number of basic transformation steps performed between 2 consecutive unfoldings.

4 Implementation Details

4.1 Coq Prototype

Before the actual implementation of the supercompiler in J-Bob Lisp was started, we implemented a prototype in Coq to study different approaches to the implementation and their impact on the correctness proof. We used Coq, because creating proofs of such scale and complexity by hand in J-Bob appeared so lengthy and tedious as to be completely impractical. Currently the Coq prototype contains implementations of the main supercompiler components – basic transformations and unfolding – as well as a re-implementation of the rewriting component of J-Bob itself. These implementations match very closely the corresponding code in the J-Bob version. The correctness proofs of the implemented supercompiler components are almost complete, and demonstrate the feasibility of formally verifying such a proof in full.

4.2 Implementation in J-Bob

J-Bob is distributed in 2 parallel versions: one that can run inside ACL2 and one that can run inside any Scheme implementation. The J-Bob sources themselves are almost identical in the 2 versions, but there are different thin wrappers, which emulate J-Bob Lisp on top of ACL2 and Scheme respectively. We have chosen to use the Scheme version and our implementation is in Scheme[2], although most parts of the code are in the restricted Lisp subset supported by J-Bob. The files[3] containing the different parts of the source code are briefly described in Table 2.

The source files contain a fair amount of deliberate code duplication, because we need to use many pieces of code in two different ways. We must execute

[2] The code was tested with Racket 6.4, in R5RS emulation mode, with redefinition of initial bindings allowed.

[3] `https://bitbucket.org/dkrustev/jbobscp`

Table 2. Source code files

File	Description
`Coq/JBobScp.v`	Coq prototype
`j-bob/*`	A copy of J-Bob sources, as a git submodule
`Scheme/j-bob-rewriter.scm`	Copies of some definitions of J-Bob itself, which are only needed for the supercompiler correctness proofs
`Scheme/j-bob-rewriter2.scm`	patched versions of some definitions of J-Bob. The main goal of the changes is to add an explicit return flag if a transformation step or a sequence of steps was successfully applied. All patched versions have the same names as the original definitions, with an added suffix "2"
`Scheme/j-bob-scp.scm`	Implementation of the supercompiler for J-Bob List programs
`Scheme/j-bob-expand-proofs.scm`	A "proof expander": taking a list of proofs with steps using an extended syntax, and expanding them to simple steps directly accepted by J-Bob
`Scheme/j-bob-scp-proofs.scm`	Proof of correctness for the supercompiler, using proof automation supplied by the supercompiler itself.

those parts directly (J-Bob itself, the supercompiler). We also need to reason about the same code inside J-Bob, which, as a minimum, requires to wrap each function definition with a termination proof and to package all such definitions in an environment, which can be passed to J-Bob at runtime. Simple text file comparison can convince us that the parallel versions of duplicated definitions are identical. For example, we can compare `j-bob-rewriter.scm` (used only in the proofs) with the original source of J-Bob. Similarly, we can compare `j-bob-scp.scm` (which is executed) to `j-bob-scp-proofs.scm` (which contains (a part of) the same definitions inside the proof environment).

Supercompiler Implementation The main functions of the supercompiler implementation are as follows:

- `(simplify-current fullscp eroot path e)` tries to find a sequence of suitable basic transformation steps (Sect. 3.2) for the current subexpression `e`, which is at position `path` inside the top-level expression `eroot`. The boolean flag `fullscp` indicates whether we want full supercompilation or just a simple form of partial evaluation.
- `(simplify-top fullscp e)` returns (if possible) a basic transformation sequence for a *single* subexpression of the top-level expression `e`.
- `(simplify* defs fullscp fuel e)` returns a sequence of basic transformation steps, which simplify up to `(length fuel)` subexpressions of the top-level expression `e`. `defs` is the list of current definitions.

- (unfold−steps−top defs e) returns a list of all possible unfolding steps inside the expression e.
- (choose−unfold−step defs fullscp whitelist blacklist simplfuel e) returns a single unfolding step according to the specified strategy:
  - if whitelist is not empty, the first unfolding (in outermost leftmost order) for a function in the white-list is returned;
  - if blacklist is not empty, the first unfolding (in innermost leftmost order) for a function *not* in the black-list is returned;
  - if both lists are empty, we select the unfolding step, which results in the smallest new expression (after simplification using simplify∗).
- (scp−steps defs fullscp unfoldfuel whitelist blacklist simplfuel e) is the top-level supercompiler function. It returns a sequence of transformation steps for the expression e, which contains up to (length unfoldfuel) unfolding steps, around each of which we can have up to (length simplfuel) basic simplification steps.

As we can deduce from these descriptions, the implementation follows closely the architecture outlined in Sect. 3.1.

Proof Expander On top of the supercompiler implementation we have built a preprocessor, which takes proofs with a richer set of possible steps, and expands them into a sequence of standard proof steps, which J-Bob can verify. With this organization the proofs produced by the supercompiler (or by other extended tactics) are always checked by J-Bob, therefore we do not need to trust them. Some of the new proof steps include:

- `(scp` [<*unfolding limit*> [<*simplification limit*> [<*unfolding white-list*> [<*unfolding black-list*>]]]]`)`. This is the step, which calls the supercompiler on the current goal, and pastes its result at the current point of the proof. There is also a variant starting with the keyword `simpl`, which performs a reduced set of basic transformations, roughly equivalent to simple partial evaluation. It is useful, for example, when we need to simplify a call where most of the arguments are constants, as it produces a shorter sequence of steps.
- `(expand (`<*extended path*>`)` <*transformation*>`)`. This step always corresponds to a single standard J-Bob step, but it permits an extended syntax for paths, which is easier to use: $(path_1\ (f\ n)\ path_2)$ corresponds to $(path_1\ path_3\ path_2)$, where $path_3$ is the path to the n-th occurrence of a call to f in the subexpression found at $path_1$.

Readers interested in examining the source code of the implementation may find it, in places, unnecessarily convoluted. Such complicated tricks are, however, necessary to overcome limitations of the J-Bob Lisp dialect: no higher-order functions, no mutual recursion, no let-expressions. While the first limitation was not felt heavily during the development of the supercompiler (which is, after all, just a few hundred lines), the combination of the last two restrictions proved to be a major hurdle.

Supercompiler Correctness Proof All the main supercompiler functions listed above, as well as many of the auxiliary definition they use, return a list of transformation steps. As a result, the structure of the correctness proof is quite simple, and follows the structure of the implementation itself: for each such function definition we have a lemma, stating that the returned list of steps – in a suitable context – can be successfully executed by J-Bob. As an example, here is the proof for the function simplify-current:

```
((dethm simplify-current-correct (fullscp eroot path e)
    (if (focus-is-at-path?2 path eroot)
      (if (equal (find-focus-at-path2 path eroot) e)
        (equal (car (rewrite/steps2 (axioms) eroot (simplify-current
            fullscp eroot path e))) 't)
        't)
      't))
    nil
    (scp 1 50 (simplify-current))
    (insert-Q (A A A) (equal (find-focus-at-path2 path eroot) (if-c (if.
        Q e) (if.A e) (if.E e))))
    ((A A A A 1) (simplify-if-correct fullscp eroot path (if.Q e) (if.A
        e) (if.E e)))
    ((A A A Q 2) (if?/if-c/if.Q/if.A/if.E e))
    (scp 0 50)
    ((A A E A 1) (simplify-app-correct eroot path e))
    (expand ((rewrite/steps2 1)) (rewrite/steps2 (axioms) eroot '()))
    (scp 0 50))
```

It contains appeals to some lemmas about auxiliary functions (simplify-if-correct, simplify-app-correct) and some other manual steps, interspersed with calls to the supercompiler to fill in the tedious parts of the proof.

There are a few important design decisions, which substantially simplified the formal proofs of supercompiler correctness. As the definition of correctness (Sect. 3.1) uses the J-Bob rewriting machinery as a reference point, we use the same machinery as much as possible in the implementation of the supercompiler as well. For example, we could implement positive/negative information propagation by keeping track of the set of conditions we know to be true/false, as we descend recursively inside subexpressions. The J-Bob rewriter uses, however, a different approach (likely more adapted to its own architecture). There are functions (prem-A?/prem-E? prem path e), which check if condition prem occurs positively/negatively somewhere on the given path inside the top-level expression e. So we chose to use the same functions for information propagation inside the supercompiler, which is the main reason to carry around the top-level expression eroot in most supercompiler functions. Another decision, explicitly aimed at simplifying the correctness proof, was to use a small-step-style implementation not only for the unfolding steps, but for the basic transformation steps as well. The reason can be explained with the following example. If we simplify the expression $(f\ e_1\ e_2)$ using a big-step style, we first compute recursively the simplification steps for e_1 and e_2 (say $steps_1$ and $steps_2$), and the result for the whole expression will be (append $steps_1$ $steps_2$). What is important is that we compute $steps_2$ in the context of the original expression, but the rewriting engine will have to apply them on the result of applying $steps_1$ to this initial expression. This mismatch prevents a simple inductive argument, because we cannot use directly the inductive hypothesis for e_2. In order to make such a proof feasible,

we would have to first formalize that the transformation steps for e_1 and e_2 are independent, as they treat disjoint subexpressions, which would complicate and lengthen the proof considerably.

The proof of the full J-Bob supercompiler is far from complete – mostly for reasons we discuss in the next section. Existing proofs cover many of the basic transformation steps, however, and clearly demonstrate the importance of supercompiler proof automation to make them feasible. Completing the formal proofs should simply be a matter of investing more time and solving some problems unrelated to the use of supercompilation in verification. The Coq prototype shows there are no important technical difficulties in the formal proofs themselves.

5 Performance Evaluation

Table 3 contains some statistics about the currently existing lemmas in the supercompiler correctness proof[4]. The proofs are classified – subjectively – in 3 categories:

1. "typical" proofs, which rely mostly on logical reasoning (analysis by cases, appeals to existing lemmas, rewriting, ...);
2. proofs by direct computation (which requires, however, many J-Bob standard steps);
3. a mixture of the above 2 categories – proofs that for the most part are like those in the first category, but also contain steps using computation over known values.

The statistics presented in the table support this classification – the values in the last two columns are very similar within the categories 1 and 2, but quite distinct between the two categories. Category 3 has more diverse values, but on average they are between those for category 1 and category 2.

The good news first: Even if we completely ignore the statistics of categories 2 and 3, we can conclude that supercompilation is of great help as a form of proof automation: category 1 has almost an order of magnitude of savings in the number of proof steps one has to enter manually (7.14 expanded steps per single original step). If we include all categories, the savings are even more impressive – almost two orders of magnitude.

The bad news is that the current implementation is too slow to be used in an interactive fashion. The expansion of all existing proofs (which are only a part of the full supercompiler correctness proof) takes almost 40 sec in this experiment; with the time J-Bob requires to verify the expanded proofs, the full time is almost 75 sec. As J-Bob reevaluates all existing proofs after each user modification of the current proof, working on the supercompiler correctness proof requires waiting for over a minute between each 2 interactive proof changes.

[4] Tests performed on a laptop with a Intel(R) Core(TM) i7-2640M CPU @ 2.80GHz, 8 GB RAM, OS Microsoft Windows 7 Pro 64 bit, using DrRacket 6.4 with debug info switched off.

Table 3. Correctness proof statistics

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
1	path-b-exp-induction	3	2	6	110	120	0	66.67%	2.00
1	focus-is-at-path/rewrite-focus-at-path	18	11	143	5475	5530	611	61.11%	7.94
1	focus-is-at-path/rewrite-focus-at-path/cons-true	6	1	28	187	190	31	16.67%	4.67
1	simplify-if-correct	14	7	87	3588	3590	764	50.00%	6.21
1	if?/if-c/if.q/if.a/if.e	8	2	58	250	260	0	25.00%	7.25
1	app?-cons/car+cdr	3	2	19	156	150	16	66.67%	6.33
1	simplify-app-quoted-correct	14	9	188	577	590	16	64.29%	13.43
1	list2?-expand-list	5	2	18	156	150	0	40.00%	3.60
1	simplify-app-not-quoted-correct	12	6	92	2668	2670	80	50.00%	7.67
1	simplify-app-correct	7	4	30	265	270	16	57.14%	4.29
1	simplify-current-correct	8	3	37	359	350	32	37.50%	4.63
2	lookup/if-true	1	1	324	422	430	32	100.00%	324.00
2	lookup/if-false	1	1	370	421	430	32	100.00%	370.00
2	lookup/if-nest-a	1	1	462	593	590	78	100.00%	462.00
2	lookup/if-nest-e	1	1	416	577	580	64	100.00%	416.00
2	lookup/atom/cons	1	1	48	172	180	47	100.00%	48.00
2	lookup/equal-same	1	1	186	296	290	15	100.00%	186.00
3	simplify-if-correct-if-true	16	12	1126	2840	2880	330	75.00%	70.38
3	simplify-if-correct-if-false	16	12	1135	3026	3030	392	75.00%	70.94
3	simplify-if-correct-if-nest-a	14	11	1722	5257	5380	623	78.57%	123.00
3	simplify-if-correct-if-nest-e	14	11	1003	3932	3950	376	78.57%	71.64
3	simplify-atom-correct	25	11	1180	5397	5410	95	44.00%	47.20
3	simplify-equal-correct	29	16	589	2434	2430	47	55.17%	20.31
1	Total by category	99	49	707	13916	13990	1566	49.49%	7.14
2	Total by category	6	6	1806	2481	2500	268	100.00%	301.00
3	Total by category	114	73	6755	22886	23080	1863	64.04%	59.25
	Total	219	128	9268	39283	39570	3697	58.45%	42.32
	Total time with proof checking				73726	74140	4335		

Column *Description*

(1) Lemma category

(2) Lemma name

(3) Total number of steps in original proof

(4) Number of supercompilation steps in original proof

(5) Total number of steps after proof expansion

(6)-(8) CPU/real/GC time (msec) for proof expansion (including supercompilation), as reported by Racket's `time` function

(9) Frequency of supercompilation steps in original proof

(10) Ratio of expanded to original proof steps

Initially the performance of the system was even worse, but we managed to improve it substantially by adding some new options to the supercompiler: a call-by-value unfolding strategy was added beside the existing ones ("best" unfolding and call-by-name); a flag was added to perform only a reduced set of basic transformation steps, corresponding to a form of simple partial evaluation. Table 4 demonstrates the effect of these options on the most time-consuming proofs in category 2. These proofs contain only a single call to the supercompiler, and they can be completed by using any combination of options, which makes them suitable for this comparison. The table shows – as expected – that CBV outperforms a lot CBN and that simple partial evaluation is a little bit better than full supercompilation. Of course, such comparisons are meaningful only when all options lead to the same result in the corresponding proof.

Table 4. Evaluation strategy statistics

	Full supercompilation				Simple partial evaluation			
	CBN		CBV		CBN		CBV	
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
lookup/if-true	1038	4960	498	592	716	2840	324	422
lookup/if-false	1226	6209	569	624	858	3478	370	421
lookup/if-nest-a	1632	8830	711	749	1172	4977	462	593
lookup/if-nest-e	1424	6989	640	717	1010	4134	416	577
lookup/atom/cons	120	390	72	203	74	327	48	172
lookup/equal-same	534	2028	285	406	350	1295	186	296

Column	*Description*
(1)	Total number of steps after proof expansion
(2)	CPU time (msec)

Even with these improvements we still need one "quick-and-dirty" trick to get around the performance problem. Proofs are split into smaller pieces (in the form of auxiliary lemmas). Especially those parts of a proof, which involve direct computation only, are extracted as separate lemmas whenever it is not too complicated. (This is the real reason for the existence of all lemmas in category 2.) After the proof of each such lemma is ready and checked, we shunt it by replacing it with a fake proof, which consists of only a single J-Bob standard step (using a fake axiom). Such shunting is especially useful for the lemmas in categories 2 and 3. When it is put in place, the total time for proof expansion and checking goes down to a little over 10 sec on the same machine, which is already (barely) acceptable for interactive proof editing. Still, the rechecking time between proof modifications will go up as we continue to make progress towards a full proof of supercompiler correctness (comparable to that in the Coq prototype). Given the current performance of the system, we decided to postpone the work towards completing the proof until we can achieve more improvements in its reactivity.

If we want better improvements in performance, we must first analyze the causes for the current long processing times. They seem mostly related to the underlying proof assistant:

- J-Bob does not have its own interactive editor or shell. It just provides a simple high-level API, which can be used directly from the Scheme REPL. As this API is stateless, it entails full rechecking of all current proofs after each user interaction.
- J-Bob only allows very elementary program transformations as proofs steps. Allowing even a simple form of partial evaluation as a built-in proof step (similar to what proof assistants like Coq and Agda provide) would eliminate a big source of inefficiency in the proofs listed in Table 3.
- The restrictions of J-Bob's Lisp subset – especially the lack of let-expressions – often make it too hard to write an efficient version of the algorithm one has in mind. Instances of this problem exist inside the sources of both J-Bob itself and our supercompiler: sometimes they perform multiple traversals or repeat some computations just because of the lack of let-expressions.

Of course, all these limitations stem from the goal of J-Bob: to be a minimalistic proof assistant used mostly for educational purposes. Solving some of these limitations – such as the introduction of a built-in partial evaluation step – would require modifying J-Bob itself, and making it bigger, more complex, and potentially less reliable. Some other limitations can probably be removed without touching the J-Bob core. Adding let-expressions can likely be done by a preprocessor. A dedicated J-Bob REPL (or even just a statefull API for the Scheme REPL) would avoid the need to recheck all proofs after each interaction. We leave the study of these possibilities for future work.

6 Related Work

Proof automation is a large and active research area, covering a broad range of methods. The bibliography of one recent book [7] has about 700 references. We shall therefore not attempt a thorough comparison of the current method to other existing methods for proof automation, limiting ourselves instead to just a few works we consider most relevant.

J-Bob is closely related to ACL2 [10] and Milawa [1], as they all follow the traditions of the early Boyer-Moore prover, Nqthm. But because of their different intended usage, these provers have important differences. While J-Bob is a minimalistic educational tool, with no proof automation at all, both ACL2 and Milawa have facilities for proof automation. ACL2 is an industrial-strength theorem prover with powerful methods for automatic proof search. Its architecture is monolithic, without a dedicated core, and bugs anywhere in the system can impact its soundness as a prover [1]. Milawa is another prover in the Nqthm family, which proposes an interesting solution to the soundness and trusted-core problems. Its minimal core proof checker has to be trusted, while a reflection mechanism allows a new proof checker to be installed, if its soundness can be

verified by the current checker. By repeatedly installing new proof checkers, which accept higher-level proof steps, the level of Milawa can be raised to one approaching in power ACL2 [1]. In our approach, we leave the core proof checker (J-Bob) untouched, and instead expand higher-level proof steps into sequences of steps it can check. Such proof expansion can lead to high processing-time requirements in larger proofs. On the other hand, Milawa requires trusting not only its core proof checker (which appears even simpler than J-Bob), but also its reflection mechanism. We leave a more detailed comparison of the two approaches for future work.

The idea to apply supercompilation for proof automation appears already in some of Turchin's early papers [23]. Different specific applications of verification by supercompilation have been studied [11,13,18,19]. In all these cases we have to rely on the correctness of the used supercompiler, or have it proven correct, in order to trust the results of verification. There is even a theorem prover based on distillation (a program transformation method closely related to supercompilation) – Poitín [6]. Again, it appears that distillation is closely integrated into the kernel of this prover, so that bugs in its implementation may impact the soundness of the proved results.

Klyuchnikov et al. [14] propose an elegant solution to avoid the necessity to trust that the supercompiler is bug-free. They introduce a *certifying* supercompiler, which produces – together with the resulting transformed program – a proof that it is equivalent to the input program. This proof may be verified by an independent proof checker (hopefully much simpler than the supercompiler, and so with lower probability of soundness-critical bugs). We use the same idea, with a shortcut: our supercompiler produces only a proof, and the resulting program can be recovered from this proof by an independent process. Another important difference is that the supercompiler of Klyuchnikov et al. is implemented in a language (Scala) very different from the one it can treat (a version of Martin-Löf type theory). So their supercompiler cannot be used directly for its own verification.

Self-application has long been a desirable – but also somewhat elusive – goal in the context of supercompilation and partial evaluation in general. This interest stems mostly from the possibility to apply the Futamura projections [4], which enable the production of compilers from interpreters, and of compiler generators. Such optimizing self-application has been demonstrated first with partial evaluation [9], and then extended to cover online partial evaluation [5]. It appears harder to achieve in the context of supercompilation – there is a single description of successful experiments of self-application with a version of the Refal supercompiler [20]. The supercompiler we describe is self-applicable – in a sense that it can process programs in the same language it is written in. It cannot hope to achieve Futamura-projection-like self-application, mostly because it currently lacks folding. As we have demonstrated, it is still powerful enough to be used in proofs reasoning about its own sources.

The formal verification of supercompilers has recently emerged as an interesting research topic. In earlier work [15] we have demonstrated – on a simple

supercompiler for a tiny imperative language – the feasibility of this task. The current work reuses some ideas of that previous verification effort, most notably the decomposition of the supercompilation process in several phases. Subsequent research on formal correctness proofs for supercompilers has mostly concentrated on providing general frameworks, which can simplify the verification of many different supercompilers [16, 17, 21]. In all these cases a general proof assistant is used (Coq, Agda), and no attempt is made to use a supercompiler as a proof automation tool for its own verification.

7 Conclusions and Future Work

We have described the design of a certifying supercompiler, which can work together with a proof assistant (J-Bob) and supply automatically generated proof fragments upon request by the proof assistant user. The supercompiler is also self-applicable, as it is written in the same first-order subset of Lisp, which it can process. This feature cannot currently be used for producing Futamura projections, as the system does not implement folding yet. Self-application, however, permits the supercompiler to supply proof automation for its own correctness proof. To the best of our knowledge, this is the first successful experiment, where a supercompiler can assist its own formal verification. We have quantified the amount of proof automation the supercompiler provides by measuring the ratio of high-level proof steps (relying on supercompilation) versus low-level proof steps that the proof checker can verify directly. This ratio shows an almost two-orders-of-magnitude improvement, when calculated on the ready part of the supercompiler correctness proof. We estimate that such improvement is sufficient as proof-of-concept for the applicability of the proposed approach.

An interesting feature of our approach is that the user is not forced to use the supercompiler in a one-shot, all-or-nothing fashion on a given problem. Instead, the user builds formal proofs in interaction with a proof assistant, and at each step of the proof she may try to call the supercompiler for help. It would be interesting to study if such an incremental approach can work in other domains (like program analysis).

To make the implemented system really practical for users of J-Bob, we need to solve the performance problems we have detected while working on the supercompiler correctness proof. Our analysis indicates most of these performance issues are ultimately related to limitations of J-Bob itself. We have outlined some possible solutions, which we may try in the future.

Another interesting possibility is to apply the same approach to different proof assistants, featuring different programming languages. As Klyuchnikov et al. [14] have demonstrated, it is possible to build a certifying supercompiler for a language with higher-order functions and dependent types, such as those found in proof assistants like Coq and Agda. The challenge will be to produce a similar supercompiler, which is self-applicable and integrated with the corresponding proof assistant.

Acknowledgments I would like to thank Sergei Romanenko, whose comments helped to substantially improve the presentation of this article.

References

1. Davis, J.C.: A Self-verifying Theorem Prover. Ph.D. thesis, University of Texas at Austin, Austin, TX, USA (2009)
2. Friedman, D.P., Eastlund, C.: J-bob source repository. `https://github.com/the-little-prover/j-bob`, accessed: 2015-04-15
3. Friedman, D.P., Eastlund, C.: The Little Prover. MIT Press (2015)
4. Futamura, Y.: Partial evaluation of computation process–an approach to a compiler-compiler. Higher-Order and Symbolic Computation 12(4), 381–391 (1999), `http://dx.doi.org/10.1023/A:1010095604496`
5. Glück, R.: A self-applicable online partial evaluator for recursive flowchart languages. Software: Practice and Experience 42(6), 649–673 (2012)
6. Hamilton, G.W.: Distilling programs for verification. Electr. Notes Theor. Comput. Sci. 190(4), 17–32 (2007)
7. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press (2009)
8. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)
9. Jones, N.D., Sestoft, P., Søndergaard, H.: An experiment in partial evaluation: The generation of a compiler generator. In: Jouannaud, J.P. (ed.) Rewriting Techniques and Applications: Dijon, France, May 20–22, 1985. pp. 124–140. Springer Berlin Heidelberg, Berlin, Heidelberg (1985)
10. Kaufmann, M., Moore, J.S.: The ACL2 home page. In: `http://www.cs.utexas.edu/users/moore/acl2/`. Dept. of Computer Sciences, University of Texas at Austin (2016)
11. Klimov, A.V., Klyuchnikov, I.G., Romanenko, S.A.: Automatic verification of counter systems via domain-specific multi-result supercompilation. In: Third International Valentin Turchin Workshop on Metacomputation. pp. 112–141 (2012)
12. Klyuchnikov, I., Krustev, D.: Supercompilation: Ideas and methods. The Monad Reader 23 (2014)
13. Klyuchnikov, I., Romanenko, S.: Proving the equivalence of higher-order terms by means of supercompilation. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) Perspectives of Systems Informatics: 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers. pp. 193–205. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), `http://dx.doi.org/10.1007/978-3-642-11486-1_17`
14. Klyuchnikov, I., Romanenko, S.: Certifying supercompilation for Martin-Löf's type theory. In: Voronkov, A., Virbitskaite, I. (eds.) Perspectives of System Informatics: 9th International Ershov Informatics Conference, PSI 2014, St. Petersburg, Russia, June 24-27, 2014. Revised Selected Papers. pp. 186–200. Springer Berlin Heidelberg, Berlin, Heidelberg (2015), `http://dx.doi.org/10.1007/978-3-662-46823-4_16`
15. 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)

16. Krustev, D.: Towards a framework for building formally verified supercompilers in Coq. In: Loidl, H.W., PeÃśa, 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
17. Krustev, D.N.: An approach for modular verification of multi-result supercompilers. In: Klimov, A., Romamenko, S. (eds.) Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation. pp. 177–193. University of Pereslavl Publishing House, Pereslavl-Zalessky, Russia (2014)
18. Lisitsa, A.P., Nemytykh, A.P.: Verification as a parameterized testing (experiments with the SCP4 supercompiler). Programming and Computer Software 33(1), 14–23 (2007), http://dx.doi.org/10.1134/S0361768807010033
19. Mendel-Gleason, G.: Types and verification for infinite state systems. PhD thesis, Dublin City University, Dublin, Ireland (2011)
20. Nemytykh, A.P., Pinchuk, V.A., Turchin, V.F.: A self-applicable supercompiler. In: Danvy, O., Glück, R., Thiemann, P. (eds.) Partial Evaluation: International Seminar Dagstuhl Castle, Germany, February 12–16, 1996 Selected Papers. pp. 322–337. Springer Berlin Heidelberg, Berlin, Heidelberg (1996), http://dx.doi.org/10.1007/3-540-61580-6_16
21. Reich, J.S.: Property-based Testing and Properties as Types: A hybrid approach to supercompiler verification. Ph.D. thesis, University of York (2013)
22. 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)
23. Turchin, V.: The use of metasystem transition in theorem proving and program optimization. In: de Bakker, J., van Leeuwen, J. (eds.) Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 85, pp. 645–657. Springer Berlin / Heidelberg (1980)
24. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (July 1986)

Simple Programs on Binary Trees – Testing and Decidable Equivalence

Dimitur Krustev

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

Abstract. We consider a class of simple (iteration/recursion-free) programs operating on unlabeled binary trees. We introduce a cumulative hierarchy of subclasses of programs, which cover the whole class, such that a finite adequate test set exists for each subclass. By taking the minimum subclass, in which a pair of programs live, we can decide just by testing if they are extensionally equivalent.

1 Introduction

Program testing and decidability of program equivalence are two topics with numerous theoretical and practical applications. These topics are closely related on a fundamental level [3]. Both problems – existence of an adequate finite test set and program equivalence – are undecidable not only for Turing-complete languages, but also for many smaller classes of programs [3]. So, discovering classes of programs, for which one or both problems are decidable, can be of great interest. Here we study one such specific class of programs operating on unlabeled binary trees. Programs are composed of operations for building trees, checking tree emptiness, and extracting subtrees. The language is variable-free, similar to Backus' FP [2].

The main results we present are:

- the definition of a cumulative hierarchy of subclasses of programs covering the whole class – Sect. 4.2 (based on a definition of program normal forms introduced previously by the author [8,10]);
- proofs of existence (Sect. 4.4) and optimality (Sect. 4.5) of finite adequate test sets for programs in each subclass;
- a decision procedure for program equivalence (Sect. 5), based on the existence of finite adequate test sets.

We start by introducing the class of simple programs we consider (Sect. 2). We then briefly review the notion of program normal forms (Sect. 3), which is obtained by program transformations directly inspired by supercompilation and deforestation [13,15,16]. Some definitions related to program testing are briefly introduced in Sect. 4.1. The rest of Sect. 4 is devoted to the description of our main result – the existence of finite adequate test sets.

We can illustrate the proposed method on a simple example. Consider the 2 programs in Table 1. The first one (I) simply returns the input tree unchanged. The second (ifnil(I, nil, cons(hd, tl))) returns an empty tree if the input tree is empty, otherwise it builds a new tree containing the left and right subtree of the input as left and right subtree correspondingly. Clearly both programs compute the identity function. We can prove this fact in many ways, but using the main result of this article (Theorem 1) we can just check it by direct computation – comparing the results of the 2 programs on input trees of depth ≤ 2.

Table 1. Example of deciding program equivalence by testing

input	I	ifnil(I, nil, cons(hd, tl))
nil	nil	nil
(nil . nil)	(nil . nil)	(nil . nil)
((nil . nil) . nil)	((nil . nil) . nil)	((nil . nil) . nil)
(nil . (nil . nil))	(nil . (nil . nil))	(nil . (nil . nil))
((nil . nil) . (nil . nil))	((nil . nil) . (nil . nil))	((nil . nil) . (nil . nil))

2 Simple Programs on Binary Trees

The programs we consider operate on unlabeled binary trees. We use textual Lisp-like notation for such trees, whose grammar is:

$$T \ ::= \ \texttt{nil} \mid (T \ . \ T)$$

As an example, the notation (nil . (nil . nil)) corresponds to the tree in Fig. 1.

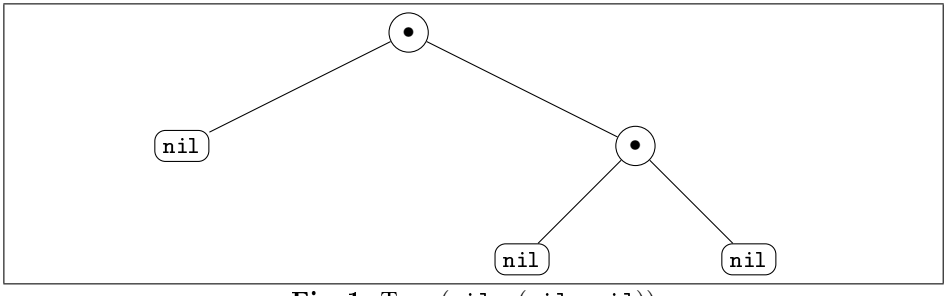


Fig. 1: Tree (nil . (nil . nil))

While this data structure is simple, it is universal, as we can encode arbitrary data as such trees. We give examples of such possible encodings below:

– $\lceil\bullet\rceil_{\text{Bool}} \ : \ \text{Bool} \to T$

$$\begin{aligned} \lceil \text{false} \rceil_{\text{Bool}} &= \text{nil} \\ \lceil \text{true} \rceil_{\text{Bool}} &= (\text{nil . nil}) \end{aligned}$$

– $\lceil\bullet\rceil_{\mathbb{N}} : \mathbb{N} \to T$

$$\begin{array}{lcl} \lceil 0\rceil_{\mathbb{N}} & = & \text{nil} \\ \lceil n+1\rceil_{\mathbb{N}} & = & (\text{nil} \mathbin{.} \lceil n\rceil_{\mathbb{N}}) \end{array}$$

– $\lceil\bullet\rceil_{\text{List}(X)} : \text{List}(X) \to T$

$$\begin{array}{lcl} \lceil[]\rceil_{\text{List}(X)} & = & \text{nil} \\ \lceil[x_1, x_2, \ldots, x_n]\rceil_{\text{List}(X)} & = & (\lceil x_1\rceil_X \mathbin{.} \lceil[x_2, \ldots, x_n]\rceil_{\text{List}(X)}) \end{array}$$

There are other, more popular universal data types in computer science – such as natural numbers and bit-strings. A distinctive advantage of binary trees is that they natively support pairing as a primitive operation, making the encoding of complicated data structures easier.

We can define the depth of a binary tree recursively in an obvious way and introduce sets of trees T_N of depth no more than $N \in \mathbb{N}$:

$$\begin{array}{lcl} \mathit{depth} & : & T \to \mathbb{N} \\ \mathit{depth}(\texttt{nil}) & = & 0 \\ \mathit{depth}(t_1 \mathbin{.} t_2) & = & 1 + \mathit{max}(\mathit{depth}(t_1), \mathit{depth}(t_2)) \\ T_N & = & \{t \in T \mid \mathit{depth}(t) \leq N\} \end{array}$$

The programs we consider are expressions (Fig. 2) built of:

– operations for constructing (`nil`, `cons`) and destructing (`hd`, `tl`) trees;
– conditional operation (`ifnil`);
– identity function (`I`) and function composition ($\circ$).

As these programs are just expressions, we shall use both terms interchangeably. Note that there are no variables in our language. This is not an important limitation, as the combination of built-in pairing and function composition permits us to encode an arbitrary set of variables [2, 7, 8]. The semantics of our language is defined in Fig. 3. To capture the possibility of errors during program execution, we use a domain extended with a new distinct element $\bot$: $T_\bot := T \cup \{\bot\}$. In the defining equations we use wild-cards ("_") to match arbitrary items not matched in any previous equation.

$$E ::= \texttt{I} \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{nil} \mid \texttt{cons}(E, E) \mid E \circ E \mid \texttt{ifnil}(E, E, E)$$

Fig. 2: Program syntax

Clearly, the class of programs we consider is far from Turing-complete, lacking any means for expressing iteration or recursion. As we are interested in decidable (extensional) equivalence, however, it is essential to consider restricted languages, as for more expressive languages equivalence is typically undecidable. This undecidability holds not only for Turing-complete languages, but even for relatively small subsets of the primitive-recursive functions, for example the class

$$
\begin{array}{ll}
[\![\bullet]\!] & : E \to T_\perp \to T_\perp \\
[\![\mathtt{I}]\!](x) & = x \\
[\![\mathtt{hd}]\!](t_1 \,.\, t_2) & = t_1 \\
[\![\mathtt{tl}]\!](t_1 \,.\, t_2) & = t_2 \\
[\![\mathtt{nil}]\!](x) & = \mathtt{nil} \\
[\![\mathtt{cons}(e_1, e_2)]\!](x) & = ([\![e_1]\!](x) \,.\, [\![e_2]\!](x)) \\
[\![e_1 \circ e_2]\!](x) & = [\![e_1]\!]([\![e_2]\!](x)) \\
[\![\mathtt{ifnil}(e_1, e_2, e_3)]\!](x) & = [\![e_2]\!](x) \text{ , if } [\![e_1]\!](x) = \mathtt{nil} \\
[\![\mathtt{ifnil}(e_1, e_2, e_3)]\!](x) & = [\![e_3]\!](x) \text{ , if } [\![e_1]\!](x) = (t_1 \,.\, t_2) \\
[\![\_]\!](\_) & = \perp
\end{array}
$$

Fig. 3: Program semantics

of elementary functions. Our class of simple programs is inspired by the simple programs introduced by Tsichritzis [14]. An important difference is that our domain consists of binary trees (and thus has a pairing operation), while Tsichritzis uses natural numbers as a domain and pairing is not definable.

3 Program Normal Forms

We can apply a number of simplifying transformations on expressions in the class we consider. Let $sel \in Sel := \{\mathtt{hd}, \mathtt{tl}\}$; the transformations we use are:

$$\mathtt{I} \circ e = e \circ \mathtt{I} = e \tag{1}$$

$$sel \circ \mathtt{cons}(e_1, e_2) = e_i \tag{2}$$

$$\mathtt{nil} \circ e = \mathtt{nil} \tag{3}$$

$$\mathtt{cons}(e_1, e_2) \circ e_3 = \mathtt{cons}(e_1 \circ e_3, e_2 \circ e_3) \tag{4}$$

$$e \circ \mathtt{ifnil}(e_1, e_2, e_3) = \mathtt{ifnil}(e_1, e \circ e_2, e \circ e_3) \tag{5}$$

$$\mathtt{ifnil}(e_1, e_2, e_3) \circ e = \mathtt{ifnil}(e_1 \circ e, e_2 \circ e, e_3 \circ e) \tag{6}$$

$$\mathtt{ifnil}(\mathtt{nil}, e_1, e_2) = e_1 \tag{7}$$

$$\mathtt{ifnil}(\mathtt{cons}(e_h, e_t), e_1, e_2) = e_2 \tag{8}$$

$$\mathtt{ifnil}(\mathtt{ifnil}(e_1, e_2, e_3), e_2', e_3') = \mathtt{ifnil}(e_1, \mathtt{ifnil}(e_2, e_2', e_3'), \mathtt{ifnil}(e_3, e_2', e_3')) \tag{9}$$

Transformations 1-6 permit to simplify instances of function composition (by either eliminating it completely or by pushing it inside subexpressions). Table 2 shows that these rules cover all cases of function composition, except for $sel_i \circ sel_j$. After these rules are applied to the condition of an if-expression, the remaining rules 7-9 allow to simplify it further.

If we exhaustively apply these transformations in a bottom-up manner, the resulting programs will be of the form shown in Fig. 4 (with an empty list of selectors being equivalent to I). We omit a detailed description of the algorithm $nf : E \to E^{nf}$ for producing normal forms, and the proofs of its properties

Table 2. Simplification rules for function composition

$\circ$	$\mathtt{I}$	$\mathtt{hd/tl}$	$\mathtt{nil}$	$\mathtt{cons}(\cdot,\cdot)$	$\mathtt{ifnil}(\cdot,\cdot,\cdot)$
$\mathtt{I}$	(1)	(1)	(1)	(1)	(1)
$\mathtt{hd/tl}$	(1)	–	$\bot$	(2)	(5)
$\mathtt{nil}$	(1)	(3)	(3)	(3)	(3)
$\mathtt{cons}(\cdot,\cdot)$	(1)	(4)	(4)	(4)	(4)
$\mathtt{ifnil}(\cdot,\cdot,\cdot)$	(1)	(6)	(6)	(6)	(5)

$$
\begin{array}{rcll}
E^{nf} & ::= & \mathtt{nil} \mid \mathtt{cons}(E^{nf}, E^{nf}) & \\
& & \mid sel_1 \circ \ldots \circ sel_n & (n \geq 0) \\
& & \mid \mathtt{ifnil}(sel_1 \circ \ldots \circ sel_n, E^{nf}, E^{nf}) & (n \geq 0)
\end{array}
$$

Fig. 4: Syntax of program normal forms

(shape of normal forms, semantics preservation), as both the algorithm and the proofs appear in previous works by the author [8,10].

We can illustrate the transformation of programs into normal form with a simple example – the composition of 2 Boolean negations. The result is, as expected, a program converting an arbitrary input tree into (an encoding of) a Boolean value, without negation.

$$
\begin{aligned}
& nf(\mathtt{ifnil}(\mathtt{I}, \mathtt{cons}(\mathtt{nil}, \mathtt{nil}), \mathtt{nil}) \circ (\mathtt{ifnil}(\mathtt{I}, \mathtt{cons}(\mathtt{nil}, \mathtt{nil}), \mathtt{nil}))) \\
& \quad = \mathtt{ifnil}(\mathtt{I}, \mathtt{nil}, \mathtt{cons}(\mathtt{nil}, \mathtt{nil}))
\end{aligned}
$$

The transformation rules described above are very similar to those used in supercompilation [13, 15] and deforestation [16]. In fact, we can consider the method for producing normal forms as a simple kind of supercompilation. As our language does not have loops or recursion, we do not need many of the complications involved in supercompilers for more powerful languages, such as folding, whistle, generalization.

4 Finite Adequate Test Sets for Simple Programs

4.1 Some Notions Related to Program Testing

We summarize here some definitions related to program testing used in the rest of the paper. We borrow most definitions from Budd et al. [3], but with slight differences in notation. In this subsection we consider an arbitrary set of programs P over a set of data D. The semantics of programs is given by an evaluation function $[\![\bullet]\!] : P \to D \to D$.

– Given a program $p \in P$, a *program neighborhood*[1] is any subset of programs $\Phi(p) \subseteq P$, such that $p \in \Phi(p)$.

[1] not to be confused with neighborhood analysis as a metacomputation technique

- A *test set* is a subset of data $T \subset D$ (usually tacitly assumed finite).
- A test set T is *adequate* for a program p (*relative* to a neighborhood $\Phi(p)$) if for any program $q \in \Phi(p)$ it holds:

$$(\forall d \in D, [\![p]\!](d) = [\![q]\!](d)) \leftrightarrow (\forall d \in T, [\![p]\!](d) = [\![q]\!](d))$$

The left-to-right direction in the last definition is trivial, as $T \subset D$; it is the right-to-left direction, which is important.

Relatively adequate test sets are often non-computable [3]:

- if $\Phi(p)$ are all programs in any Turing-complete language
- ... or all primitive recursive programs on $\mathbb{N}$
- ... or even all programs computing polynomials with integer coefficients
- ...

So, it is interesting to study classes of program neighborhoods, for which such tests are computable.

4.2 Subclasses of Simple Programs as Program Neighborhoods

Given some $N \in \mathbb{N}$ we define a subclass E_N^{nf} of expressions in normal form as those satisfying the following grammar:

$$\begin{aligned} E_N^{nf} \quad ::= \quad & \texttt{nil} \mid \texttt{cons}(E_N^{nf}, E_N^{nf}) \\ & \mid sel_1 \circ \ldots \circ sel_n \qquad\qquad (0 \le n \le N) \\ & \mid \texttt{ifnil}(sel_1 \circ \ldots \circ sel_n, E_N^{nf}, E_N^{nf}) \ (0 \le n < N) \end{aligned}$$

It is immediately obvious from this definition that these subclasses form a cumulative hierarchy covering the whole set of normal forms E^{nf}:

- $E_N^{nf} \subsetneq E_{N+1}^{nf}$;
- $\bigcup_{N \in \mathbb{N}} E_N^{nf} = E^{nf}$.

Note also that each subclass contains infinitely many programs. By extension, we classify any program $e \in E$ to be in subclass E_N^{nf} if $nf(e) \in E_N^{nf}$. The main intuition behind the introduction of these subclasses is that programs in E_N^{nf} can only "see" at depth not more than N inside the input tree. This intuition is made formal by the following statement, which is the main result of this article:

Theorem 1. *(NTrm_fixed_MaxSelCmpLen_testable*[2]*)* $\forall N \in \mathbb{N}, \forall e_1, e_2 \in E_N^{nf}$, $(\forall t \in T_{N+1}, [\![e_1]\!](t) = [\![e_2]\!](t)) \rightarrow \forall t \in T, [\![e_1]\!](t) = [\![e_2]\!](t)$

If we compare this result with the definitions from the previous subsection, we can see that the classes E_N^{nf} can serve perfectly as program neighborhoods: if $e \in E_N^{nf}$, and we set $\Phi(e) := E_N^{nf}$, Theorem 1 shows there is a computable adequate test set for e. We devote the following subsections to an overview of the proof of this theorem.

[2] The results of this article have been formally verified in Coq. In parentheses we give the corresponding names of the theorems/lemmas in the Coq sources – `https://github.com/dkrustev/SimpleTreeExprTests`

4.3 Tree Decomposition

In order to formalize the intuition about programs in E_N^{nf} "seeing" at depth at most N inside the input tree, we consider the decomposition of a tree t into (Fig. 5):

- a tree $t_1 \in T_N$, which is isomorphic to t up to depth N;
- trees $t_2, \ldots, t_n$ corresponding to all subtrees (if any) of t with roots at depth N.

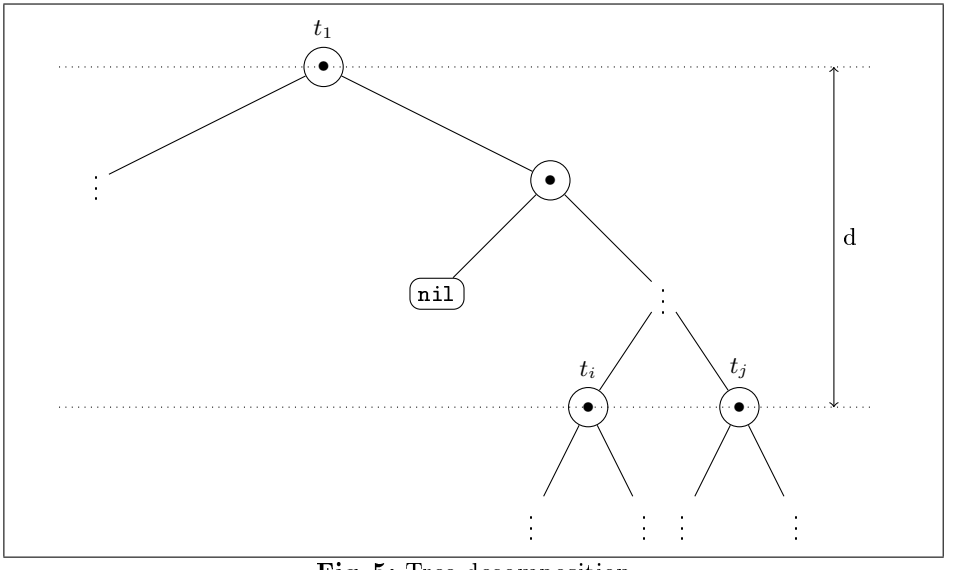


Fig. 5: Tree decomposition

To be able to recover the original tree t from its decomposition $t_1, t_2, \ldots, t_n$, we need to indicate the position of each t_i ($i \in \{2, \ldots, n\}$) inside t_1. One way to achieve it is to introduce trees with variables: given a (finite) set X of variables, the set T_X of trees with variables in X is given by the following grammar:

$$T_X \ ::= \ \texttt{nil} \mid (T_X \ . \ T_X) \mid x \qquad (x \in X)$$

The decomposition function is then $cutAt \ : \ \mathbb{N} \times T \ \to \ (X \ \to \ T) \times T_X$, $cutAt(d, t) = (\sigma, t_x)$, where:

- t_x is the tree t with all nodes at depth d replaced by variables from X
- σ is a substitution assigning the corresponding subtree to each of these variables

Example:

$$\begin{aligned} &cutAt(1, ((\texttt{nil} \ . \ \texttt{nil}) \ . \ ((\texttt{nil} \ . \ \texttt{nil}) \ . \ \texttt{nil}))) \\ &\quad = (\{x \mapsto (\texttt{nil} \ . \ \texttt{nil}), y \mapsto ((\texttt{nil} \ . \ \texttt{nil}) \ . \ \texttt{nil})\}, \\ &\qquad (x \ . \ y)) \end{aligned}$$

The action of variable substitutions is lifted to trees in the obvious way:

$$\begin{aligned} \texttt{nil}\sigma &= \texttt{nil} \\ (t_1 \ . \ t_2)\sigma &= (t_1\sigma \ . \ t_2\sigma) \\ x\sigma &= \sigma(x) \end{aligned}$$

The correctness of the decomposition function follows from the next lemma.

Lemma 1. *(vCutAt_mvSubst)* $\forall d \in \mathbb{N}, \forall t \in T, \forall \sigma, \forall t_x,\ cutAt(d,t) = (\sigma, t_x) \rightarrow t_x\sigma = t$.

As *cutAt* is defined by structural recursion over the input tree t, the proof of its correctness is by straightforward induction on t.

4.4 Existence of Adequate Test Sets

The proof of our main result relies on a couple of key observations. The first is that we can commute evaluation and substitution, provided the input tree with variables contains no variables at depth N or less. Before we write down this lemma, let us introduce some definitions. We can extend the evaluation function to work on trees with variables as well, denoted $[\![e]\!]_X \ : \ E \rightarrow T_{X_\perp} \rightarrow T_{X_\perp}$. We can use exactly the same definition as in Fig. 3, as we want the evaluation to return an error ($\perp$) whenever it encounters a variable as (top-level) input. The definition of minimum variable depth is equally straightforward:

$$\begin{aligned} minVarDepth(\texttt{nil}) &= \infty \\ minVarDepth(t_1 \ . \ t_2) &= 1 + min(minVarDepth(t_1), minVarDepth(t_2)) \\ minVarDepth(x) &= 0 \end{aligned}$$

Now our conditional commutativity property looks as follows:

Lemma 2. *(ntmvEval_ntEval)* $\forall N \in \mathbb{N}, \forall e \in E_N^{nf}, \forall X, \forall \sigma : X \rightarrow T, \forall t_x \in T_X,$ $N \leq minVarDepth(t_x) \rightarrow [\![e]\!](t_x\sigma) = ([\![e]\!]_X(t_x))\sigma$.

The proof is by induction on the structure of e. We use the condition $N \leq$ *minVarDepth* (t_x) in several cases to derive a contradiction.

The second key observation is that if we have a pair of syntactically different trees with variables, we can always build a "shallow" substitution, which – when applied to each of the 2 trees with variables – produces different ordinary trees. The substitution in question is shallow in the sense that it maps all variables to trees of depth 0 or 1.

Lemma 3. *(mvSubst_discrim)* $\forall X, \forall t_1, t_2 \in T_X, t_1 \neq t_2 \rightarrow \exists \sigma,\ (\forall x \in X,\ \sigma(x)$ $\in T_1) \wedge t_1\sigma \neq t_2\sigma$.

Proof sketch: there must be at least one pair of corresponding subtrees t_1' and t_2' with different root nodes

- if neither root is a variable, then the trivial substitution will do: $\sigma(x) =$ $\texttt{nil}, \forall x \in X$;

- if only one root is a variable, say y
  - if the other root is `nil`, then $\sigma(x) =$ if $x = y$ then (`nil . nil`) else `nil`;
  - if the other root is $(t_3\ .\ t_4)$, then $\sigma(x) =$ if $x = y$ then `nil` else (`nil . nil`);
- if $t'_1 = x, t'_2 = y$, then we can use:

$$\begin{aligned}\sigma(x) &= \texttt{nil}\\ \sigma(y) &= (\texttt{nil . nil})\\ \sigma(z) &= \texttt{nil} \qquad \forall z \in X, z \neq x, z \neq y.\square\end{aligned}$$

Armed with these observations, we can proceed with establishing the existence of finite adequate test sets:

Lemma 4. *(NTrm_fixed_MaxSelCmpLen_testable_aux)* $\forall N, \forall e_1, e_2 \in E_N^{nf}$, ($\exists\ t \in T, [\![e_1]\!](t) \neq [\![e_2]\!](t)) \rightarrow \exists t \in T_{N+1}, [\![e_1]\!](t) \neq [\![e_2]\!](t)$.

Proof sketch:

- let $t \in T$, s.t. $[\![e_1]\!](t) \neq [\![e_2]\!](t)$
- let $(\sigma, t_x) = cutAt(N, t)$
- then $[\![e_1]\!](t_x\sigma) \neq [\![e_2]\!](t_x\sigma)$ (by Lemma 1)
- commute evaluation and substitution: $([\![e_1]\!]_X(t_x))\ \sigma \neq ([\![e_2]\!]_X(t_x))\ \sigma$ (by Lemma 2)
  - possible because $cutAt(N, t) = (\sigma, t_x)$ ensures the required condition $N \leq minVarDepth(t_x)$
- so $[\![e_1]\!]_X(t_x) \neq [\![e_2]\!]_X(t_x)$
- the most interesting case is when both evaluation results are $\neq \perp$
- then (by Lemma 3) we can find σ' s.t. all $\sigma'(x) \in T_1$ and $([\![e_1]\!]_X(t_x))\sigma' \neq ([\![e_2]\!]_X(t_x))\sigma'$
- commute substitution and evaluation (again by Lemma 2): $([\![e_1]\!](t_x\sigma')) \neq ([\![e_2]\!](t_x\sigma'))$
- let $t' = t_x\sigma'$; we have $t' \in T_{N+1}$ and $[\![e_1]\!](t') \neq [\![e_2]\!](t')$. $\square$

Now it suffices to remark that Lemma 4 is just the contrapositive of Theorem 1, which concludes the proof of our main result.

4.5 Test Set Optimality

If we consider the whole set E_N^{nf} as a neighborhood of the program $e \in E_N^{nf}$, we cannot substantially improve the size of the adequate test set provided by Theorem 1, as the following theorem shows:

Theorem 2. *(undiscrTerms_exist)* $\forall N \in \mathbb{N}, \exists e_1, e_2 \in E_{N+1}^{nf}$, *such that* $\forall t \in T_{N+1}$, $[\![e_1]\!](t) = [\![e_2]\!](t)$ *and* $\exists t \in T, [\![e_1]\!](t) \neq [\![e_2]\!](t)$.

Proof sketch: it suffices to take

$$\begin{aligned}e_1 &= \texttt{hd} \circ e\\ e_2 &= \texttt{tl} \circ e, \qquad \text{where:}\\ e\ &= \underbrace{\texttt{hd} \circ \ldots \circ \texttt{hd}}_{N \text{ times}}\end{aligned}$$

$\square$

5 Decidability of Equivalence of Simple Programs

One direct application of the existence of finite adequate test sets is the decidability of equivalence for our class of simple programs. If we consider 2 programs $e_1, e_2 \in E$, we can proceed as follows:

- find the smallest N, such that $nf(e_1), nf(e_2) \in E_N^{nf}$;
- test if $[\![e_1]\!](t) = [\![e_2]\!](t)$ for all $t \in T_{N+1}$

If there is some t, for which the 2 programs return different results, they are clearly not equivalent. If, however, there is no such $t \in T_{N+1}$, then by Theorem 1 it follows that the 2 programs are equivalent.

The asymptotic complexity of this decision procedure is superexponential. Indeed, the number of unlabeled binary trees of depth no more than N is given by the following recurrence:

$$a_0 = 1$$
$$a_{N+1} = a_N^2 + 1$$

According to OEIS [12], $a_N \asymp c^{2^{N+1}}$ where $c = 1.2259\ldots$, which directly gives an superexponential bound for our algorithm. We leave as future work the search for algorithms of lower complexity. One idea that might work is to consider smaller subclasses of expressions of the following form:

$$\begin{aligned} E_S^{nf} \quad ::= \quad & \texttt{nil} \mid \texttt{cons}(E_S^{nf}, E_S^{nf}) \\ & \mid sel_1 \circ \ldots \circ sel_n \qquad\qquad (sel_1 \circ \ldots \circ sel_n \in S) \\ & \mid \texttt{ifnil}(sel_1 \circ \ldots \circ sel_n, E_S^{nf}, E_S^{nf}) \ (sel_1 \circ \ldots \circ sel_n \in S) \end{aligned}$$

If we can adapt our proof of existence of adequate test sets to this kind of smaller program neighborhoods, we can hope to get asymptotically smaller tests sets and as a result – a faster decision procedure for program equivalence.

6 Related Work

As already noted, the main results in this paper are very similar to – and to some extent inspired by – the work of Tsichritzis [14]. Tsichritzis starts with a subclass of primitive-recursive programs on natural numbers – namely, those having no nested loops – and first shows that all such programs can be represented as compositions of several simple operations. Then finite test sets are defined for this language, and – as a consequence – a decision procedure for program equivalence. We, on the other hand, start directly with a language consisting of expressions, which are composed of several simple operations on binary trees. Still, the languages considered are very similar in spirit, modulo differences in the data domains. The use of binary trees is an important advantage of our approach: as we have already noted, binary trees come with pairing as a built-in primitive, and it permits easy encoding of arbitrary data structures. The language, treated by Tsichritzis, is too weak to encode arbitrary pairing.

Binary trees have often been used in practical programming languages since the early days of Lisp. While most theoretical models of computation use either natural numbers or sequences over a fixed alphabet as the only data structure, several authors have noted the usefulness of binary trees in a more theoretical setting as well, for example Jones [7]. The variable-free nature of our language has its roots in Backus' FP [2], but Jones has proposed a similar language of a single variable [7].

The main results we report are enabled by the specific shapes of normal forms that we can produce by supercompilation-like program transformations. In this respect, the current work is an offshoot of some of the author's previous research on supercompilation [8,10]. Supercompilation – and metacomputation techniques in general – are the basis of several other methods for test generation [1, 9, 10]. Abramov's neighborhood testing [1] ensures strong adequacy properties for the generated test sets and covers arbitrary languages, but it is not guaranteed to terminate. Our skeleton testing method [10] produces adequate test sets for a Turing-complete functional language, but the program neighborhoods are finite and the test sets actually use program expressions instead of simple data values. We have also proposed a test-generation method based on metacomputation (used for program inversion) [9], which is more practically oriented, but without any formal adequacy guarantees.

The literature on program testing techniques is too big to review here. We just note several methods, which – similarly to ours – produce finite adequate test sets[3] for restricted classes of programs:

- the already discussed work of Tsichritzis [14] – primitive recursive programs on natural numbers without nested loops;
- programs computing multivariate polynomials of a known degree with integer coefficients [4, 6];
- programs computing multivariate polynomials of unknown degree with natural coefficients [5, 11].

7 Conclusions and Future Work

We have presented a class of simple programs operating on binary trees, which can be split into subclasses, such that for each subclass there exists a finite adequate test set. As a direct consequence, equivalence of programs in the whole class is decidable. The definition of the subclasses is made possible by converting programs into a normal form with specific shape, through supercompilation-like transformations.

The practical application of the equivalence decision procedure is hindered by its superexponential complexity. We have already mentioned some ideas that might help reduce this complexity, but more work is needed to flesh them out. The size of the test set (which is the cause for the decision procedure complexity) is also a hurdle for the practical application of the test generation method. A

[3] Possibly for a slightly different definition of "adequate"

potential approach for reducing the test set size, which we plan to explore, is to use trees with variables as test inputs and outputs. Another problem with applying the method for practical program testing is the use of a very restricted language, which does not by itself permit writing many interesting programs. One possibility is to study the use of the language of simple programs discussed here as the core of a Turing-complete language. For example, we can use a simple imperative language with while loops, similar to the ones used by Jones [7] and in our earlier work [8], with the language of simple programs being embedded as a sublanguage of expressions. The key research problem will be how to extend the test generation method from expressions to programs in the full language.

Another interesting problem, which we may consider in the future, is the more precise characterization of the expressiveness of the class of simple programs we have studied.

Acknowledgments I would like to thank Alexei Lisitsa for comments that helped improve the presentation of this article.

References

1. Abramov, S.M.: Metacomputation and program testing. In: Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging. pp. 121–135. Linköping University, Linköping, Sweden (1993)
2. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21(8), 613–641 (Aug 1978)
3. Budd, T.A., Angluin, D.: Two notions of correctness and their relation to testing. Acta Informatica 18(1), 31–45 (Nov 1982)
4. DeMillo, R.A., Lipton, R.J.: A probabilistic remark on algebraic program testing. Information Processing Letters 7(4), 193–195 (1978)
5. Govindarajan, K.: A note on polynomials with non-negative integral coefficients. Tech. rep., Department of Computer Science, State University of New York at Buffalo (1995)
6. Howden, W.E.: Algebraic program testing. Acta Informatica 10(1), 53–66 (1978), http://dx.doi.org/10.1007/BF00260923
7. Jones, N.D.: Computability and Complexity from a Programming Perspective. Foundations of Computing, MIT Press, Boston, London, 1 edn. (1997)
8. 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)
9. Krustev, D.: A metacomputation toolkit for a subset of F# and its application to software testing. In: Klimov, A.V., Romamenko, S.A. (eds.) Third International Valentin Turchin Workshop on Metacomputation. pp. 165–183 (2012)
10. Krustev, D.N.: Software test generation using program skeletons. PhD thesis Technical Report PRE 23/01, Wroclaw Polytechnic, Wroclaw, Poland (2001)
11. Maolepszy, J.: Reconstruction of extended polynomials from the finite number of examples. Tech. rep., Jagiellonian University, Institute Of Computer Science, Krakow (1996)

12. OEIS: sequence A003095. https://oeis.org/A003095, [Online; accessed 2016-05-08]
13. 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)
14. Tsichritzis, D.: The equivalence problem of simple programs. J. ACM 17(4), 729–738 (1970)
15. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (July 1986)
16. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. 73(2), 231–248 (1990)

On a Method of Verification of Functional Programs

Andrew M. Mironov

Moscow State University

amironov66@gmail.com

Abstract. In this paper the problem of verification of functional programs (FPs) over strings is considered, where specifications of properties of FPs are defined by other FPs, and a FP Σ_1 meets a specification defined by another FP Σ_2 iff a composition of functions defined by the FPs Σ_1 and Σ_2 is equal to the constant 1. We introduce a concept of a state diagram of a FP, and reduce the verification problem to the problem of an analysis of the state diagrams of FPs. The proposed approach is illustrated by the example of verification of a sorting program.

Keywords: functional program, state diagram, verification

1 Introduction

The problem of program verification is one of the main problems of theoretical computer science. For various classes of programs there are used various verification methods. For example, for a verification of sequential programs there are used Floyd's inductive assertions method [1], Hoare logic [2], etc. For verification of parallel and distributed programs there are used methods based on a calculus of communicating systems (CCS) and π-calculus [3], [4], a theory of communicating sequential processes (CSP) and its generalizations [5], [6], temporal logic and model checking [7], process algebra [8], Petri nets [9], etc.

Main methods of verification of functional programs (FPs) are computational induction and structural induction [10]. Disadvantages of these methods are related to difficulties to construct formal proofs of program correctness. Among other methods of verification of FPs it should be noted a method based on reasoning with datatypes and abstract interpretation through type inference [12], a model checking method to verify FPs [13], [14], methods based on flow analysis [11] methods based on the concept of a multiparametric tree transducer [15].

In this article we consider FPs as systems of algebraic equations over strings. We introduce a concept of a state diagram for such FPs and present the verification method based on the state diagrams. The main advantages of our approach in comparison with all the above approaches to verification of FPs are related to the fact that our approach allows to present proofs of correctness of FPs in the form of simple properties of their state diagrams.

The basic idea of our approach is the following. We assume that a specification of properties of a FP under verification Σ_1 is defined by another FP Σ_2, whose input is equal to the output of Σ_1, i.e. we consider FP $\Sigma_1 \circ \Sigma_2$, which is a composition Σ_1 and Σ_2. We say that a FP Σ_1 is correct with respect to the specification Σ_2 iff the input-output map $f_{\Sigma_1 \circ \Sigma_2}$, which corresponds to the FP $\Sigma_1 \circ \Sigma_2$ (i.e. $f_{\Sigma_1 \circ \Sigma_2}$ is a composition of the input-output maps corresponded to Σ_1 and Σ_2) has an output value 1 on all its input values. We reduce the problem of a proving the statement $f_{\Sigma_1 \circ \Sigma_2} = 1$ to the problem of an analysis of a state diagram for the FP $\Sigma_1 \circ \Sigma_2$.

The proposed method of verification of FPs is illustrated by an example of verification of a sorting FP. At first, we present a complete proof of correctness of this FP by structural induction. This is done for a comparison of the complexity of a manual verification of the FP on the base of the structural induction method, and the complexity of the proposed method of automatic verification of FPs. At second, we present a correctness proof of the FP by the method based on constructing its state diagram. The proof by the second method is significantly shorter, and moreover, it can be generated automatically. This demonstrates the benefits of the proposed method of verification of FPs in comparison with the manual verification based on the structural induction method.

2 Main concepts

2.1 Terms

We assume that there are given sets

- $\mathcal{D}$ of **values**, which is the union $\mathcal{D}_{\mathbf{C}} \cup \mathcal{D}_{\mathbf{S}}$, where
  - elements of $\mathcal{D}_{\mathbf{C}}$ are called **symbols**, and
  - elements of $\mathcal{D}_{\mathbf{S}}$ are called **symbolic strings** (or briefly **strings**), and each string from $\mathcal{D}_{\mathbf{S}}$ is a finite (maybe empty) sequence of elements of $\mathcal{D}_{\mathbf{C}}$,
- $\mathcal{X}$ of **data variables** (or briefly **variables**)
- $\mathcal{C}$ of **constants**,
- $\mathcal{F}$ of **functional symbols (FSs)**, and
- Φ of **functional variables**

where each element m of any of the above sets is associated with a **type** designated by the notation $type(m)$, and

- if $m \in \mathcal{D} \cup \mathcal{X} \cup \mathcal{C}$, then $type(m) \in \{\mathbf{C}, \mathbf{S}\}$,
- if $m \in \mathcal{F} \cup \Phi$, then $type(m)$ is a notation of the form $t_1 \times \ldots \times t_n \to t$, where $t_1, \ldots, t_n, t \in \{\mathbf{C}, \mathbf{S}\}$.

If $d \in \mathcal{D}_{\mathbf{C}}$, then $type(d) = \mathbf{C}$, and if $d \in \mathcal{D}_{\mathbf{S}}$, then $type(d) = \mathbf{S}$.

Each constant $c \in \mathcal{C}$ corresponds to an element of $\mathcal{D}_{type(c)}$, called a **value** of this constant. The notation ε denotes a constant of the type $\mathbf{S}$, whose value is an empty string. We assume that ε is the only constant of the type $\mathbf{S}$.

Each FS $f \in \mathcal{F}$ corresponds to a partial function of the form $\mathcal{D}_{t_1} \times \ldots \times \mathcal{D}_{t_n} \to \mathcal{D}_t$, where

$$type(f) = t_1 \times \ldots \times t_n \to t.$$

This function is denoted by the same symbol f.

Below we list some of the FSs which belong to $\mathcal{F}$. Beside each FS we point out (with a colon) its type.

1. $head : \mathbf{S} \to \mathbf{C}$. The function *head* is defined for non-empty string, it maps each non-empty string to its first element.
2. $tail : \mathbf{S} \to \mathbf{S}$. The function *tail* is defined for non-empty string, it maps each non-empty string u to a string (called a **tail** of u) derived from u by removal of its first element.
3. $conc : \mathbf{C} \times \mathbf{S} \to \mathbf{S}$. For each pair $(a, u) \in \mathcal{D}_{\mathbf{C}} \times \mathcal{D}_{\mathbf{S}}$ the string $conc(a, u)$ is obtained from u by adding the symbol a before.
4. $empty : \mathbf{S} \to \mathbf{C}$. Function *empty* maps empty string to the symbol 1, and each non-empty string to the symbol 0.
5. $=: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$. The value of the function $=$ on the pair (u, v) is equal to 1 if $u = v$, and 0 otherwise.
6. $\leq: \mathbf{C} \times \mathbf{C} \to \mathbf{C}$. We assume that $\mathcal{D}_{\mathbf{C}}$ is linearly ordered set, and the value of the function $\leq$ on the pair (u, v) is equal to 1 if $u \leq v$, and 0 otherwise.
7. Boolean FSs: $\neg : \mathbf{C} \to \mathbf{C}$, $\wedge : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$, etc., corresponding functions are standard boolean functions on the arguments 0 and 1 (i.e. $\neg(1) = 0$,, etc.) and are not defined on other arguments.
8. *if_then_else* $: \mathbf{C} \times t \times t \to t$, where $t = \mathbf{C}$ or $\mathbf{S}$ (i.e. the notation *if_then_else* denotes two FSs), and functions corresponding to both FSs are defined by the same way:

$$if\_then\_else\ (a, u, v) \stackrel{\text{def}}{=} \begin{cases} u, \text{ if } a = 1 \\ v, \text{ otherwise.} \end{cases}$$

A concept of a **term** is defined inductively. Each term e is associated with a certain type $type(e) \in \{\mathbf{C}, \mathbf{S}\}$. Each data variable and each constant is a term, a type of which is the same as the type of this variable or constant. If $e_1, \ldots, e_n$ is a list of terms and g is a FS or a functional variable such that

$$type(g) = \ type(e_1) \times \ldots \times type(e_n) \to t$$

then the notation $g(e_1, \ldots, e_n)$ is a term of the type t.

We shall denote terms

$$head(e),\ tail(e),\ conc(e_1, e_2),\ empty(e),$$
$$= (e_1, e_2),\ \leq (e_1, e_2),\ if\_then\_else\ (e_1, e_2, e_3)$$

in the form

$$e_h,\ e_t,\ e_1 e_2,\ [\![e = \varepsilon]\!],\ [\![e_1 = e_2]\!],\ [\![e_1 \leq e_2]\!],\ [\![e_1]\!]\, e_2 : e_3$$

respectively. Terms containing boolean FSs will be denoted as in mathematical texts (i.e. in the form $e_1 \wedge e_2$, etc.). Terms of the form $e_1 \wedge \ldots \wedge e_n$ can also be denoted as $[\![e_1, \ldots, e_n]\!]$.

2.2 A concept of a functional program over strings

A **functional program over strings** (referred below as a **functional program (FP)**) is a set Σ of functional equations of the form

$$\begin{cases} \varphi_1(x_{11}, \ldots, x_{1n_1}) = e_1 \\ \ldots \\ \varphi_m(x_{m1}, \ldots, x_{mn_m}) = e_m \end{cases} \tag{1}$$

where $\varphi_1, \ldots, \varphi_m$ are distinct functional variables, and for each $i = 1, \ldots, m$ $\varphi_i(x_{i1}, \ldots, x_{in_i})$ and e_i are terms of the same type, such that

$$X_{e_i} = \{x_{i1}, \ldots, x_{in_i}\}, \quad \Phi_{e_i} \subseteq \{\varphi_1, \ldots, \varphi_m\}$$

(where X_e and Φ_e are sets of all data variables and functional variables respectively occurred in the term e). We shall use the notation Φ_Σ for the set of all functional variables occurred in Σ.

FP (1) specifies a list

$$(f_{\varphi_1}, \ldots, f_{\varphi_m}) \tag{2}$$

of functions corresponding to the functional variables from Φ_Σ, which is the least (in the sense of an order on lists of partial functions, described in [10]) solution of (1) (this list is called a **least fixed point (LFP)** of the FP (1), all details related to the concept of a LFP can be found in chapter 5 of the book [10]). Values of these functions can be calculated by a standard recursion. We assume that for each FP under consideration all components of its LFP are total functions. First function in the list (2) (i.e. f_{φ_1}) is denoted by f_Σ, and is called a **function corresponding to** Σ. If Σ has the form (1), then $type(\Sigma)$ denotes the type $type(e_1)$.

3 Example of specification and verification of a FP

3.1 Example of a FP

Consider the following FP:

$$\begin{aligned} &\mathbf{sort}(x) = [\![x = \varepsilon]\!]\, \varepsilon : \mathbf{insert}(x_h, \mathbf{sort}(x_t)) \\ &\mathbf{insert}(a, y) = [\![y = \varepsilon]\!]\, a\varepsilon \\ &\qquad\qquad : [\![a \leq y_h]\!]\, ay \\ &\qquad\qquad\qquad : y_h\ \mathbf{insert}(a, y_t) \end{aligned} \tag{3}$$

This FP defines a function of string sorting. The FP consists of two equations, which define the following functions:

- **sort** : $\mathbf{S} \to \mathbf{S}$ is a main function, and
- **insert** : $\mathbf{C} \times \mathbf{S} \to \mathbf{S}$ is an auxiliary function, which maps a pair $(a, y) \in \mathbf{C} \times \mathbf{S}$ to the string derived by an insertion of the symbol a to the string y, with the following property: if the string y is ordered, then the string $\mathbf{insert}(a, y)$ also is ordered.
  (we say that a string is ordered, if its components form a nondecreasing sequence).

3.2 Example of a specification of a FP

One of correctness properties of FP (3) it the following: $\forall\, x \in \mathbf{S}$ the string $\mathbf{sort}(x)$ is ordered. This property can be described formally as follows. Consider a FP defining a function **ord** of string ordering checking:

$$\begin{aligned}\mathbf{ord}(x) = [\![x = \varepsilon]\!]\ 1 \\ : [\![x_t = \varepsilon]\!]\ 1 \\ : [\![x_h \le (x_t)_h]\!]\ \mathbf{ord}(x_t) : 0\end{aligned} \tag{4}$$

The function **ord** allows to describe the above property of correctness as the following mathematical statement:

$$\forall\, x \in \mathbf{S} \quad \mathbf{ord}(\mathbf{sort}(x)) = 1 \tag{5}$$

3.3 Example of a verification of a FP

The problem of verification of the correctness property of FP (3) consists of a formal proof of (5). This proposition can be proved like an ordinary mathematical theorem, for example using the method of mathematical induction. For example, a proof of this proposition can be the following.

If $x = \varepsilon$, then, according to first equation of system (3), the equality $\mathbf{sort}(x) = \varepsilon$ holds, and therefore

$$\mathbf{ord}(\mathbf{sort}(x)) = \mathbf{ord}(\varepsilon) = 1.$$

Let $x \neq \varepsilon$. We prove (5) for this case by induction. Assume that for each string y, which is shorter than x, the equality

$$\mathbf{ord}(\mathbf{sort}(y)) = 1$$

holds. Prove that this implies the equality

$$\mathbf{ord}(\mathbf{sort}(x)) = 1. \tag{6}$$

(6) is equivalent to the equality

$$\mathbf{ord}(\ \mathbf{insert}(x_h, \mathbf{sort}(x_t))) = 1 \tag{7}$$

By the induction hypothesis, the equality

$$\mathbf{ord}(\mathbf{sort}(x_t)) = 1$$

holds, and this implies (7) on the reason of the following lemma.

Lemma.
The following implication holds:

$$\mathbf{ord}(y) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, y)) = 1 \tag{8}$$

Proof.
We prove the lemma by induction on the length of y.
If $y = \varepsilon$, then the right side of(8) has the form

$$\mathbf{ord}(a\varepsilon) = 1$$

which is true by definition **ord**.

Let $y \neq \varepsilon$, and for each string z, which is shorter than y, the following implication holds:

$$\mathbf{ord}(z) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, z)) = 1 \tag{9}$$

Let $c \stackrel{\text{def}}{=} y_h$, $d \stackrel{\text{def}}{=} y_t$.
(8) has the form

$$\mathbf{ord}(cd) = 1 \quad \Rightarrow \quad \mathbf{ord}(\mathbf{insert}(a, cd)) = 1 \tag{10}$$

To prove the implication (10) it is necessary to prove that if $\mathbf{ord}(cd) = 1$, then the following implications hold:

(a) $a \leq c \quad \Rightarrow \quad \mathbf{ord}(a(cd)) = 1$,
(b) $c < a \quad \Rightarrow \quad \mathbf{ord}(c\,\mathbf{insert}(a, d)) = 1$.

(a) holds because $a \leq c$ implies

$$\mathbf{ord}(a(cd)) = \mathbf{ord}(cd) = 1.$$

Let us prove (b).

– $d = \varepsilon$. In this case, right side of (b) has the form

$$\mathbf{ord}(c(a\varepsilon)) = 1 \tag{11}$$

(11) follows from $c < a$.
– $d \neq \varepsilon$. Let $p \stackrel{\text{def}}{=} d_h$, $q \stackrel{\text{def}}{=} d_t$.
In this case, it is necessary to prove that if $c < a$, then

$$\mathbf{ord}(c\,\mathbf{insert}(a, pq)) = 1 \tag{12}$$

1. if $a \leq p$, then (12) has the form

$$\mathbf{ord}(c(a(pq))) = 1 \tag{13}$$

Since $c < a \leq p$, then (13) follows from the equalities

$$\begin{aligned}&\mathbf{ord}(c(a(pq))) = \mathbf{ord}(a(pq)) = \mathbf{ord}(pq) = \\ &= \mathbf{ord}(c(pq)) = \mathbf{ord}(cd) = 1\end{aligned}$$

2. if $p < a$, then (12) has the form

$$\mathbf{ord}(c(p\ \mathbf{insert}(a, q))) = 1 \tag{14}$$

Since, by assumption,

$$\mathbf{ord}(cd) = \mathbf{ord}(c(pq)) = 1$$

then $c \leq p$, and therefore (14) can be rewritten as

$$\mathbf{ord}(p\ \mathbf{insert}(a, q)) = 1 \tag{15}$$

If $p < a$, then

$$\mathbf{insert}(a, d) = \mathbf{insert}(a, pq) = p\ \mathbf{insert}(a, q)$$

therefore (15) can be rewritten as

$$\mathbf{ord}(\mathbf{insert}(a, d)) = 1 \tag{16}$$

(16) follows by the induction hypothesis for the Lemma (i.e., from the implication (9), where $z \stackrel{\text{def}}{=} d$) from the equality

$$\mathbf{ord}(d) = 1$$

which is justified by the chain of equalities

$$1 = \mathbf{ord}(cd) = \mathbf{ord}(c(pq)) = \quad \text{(since } c \leq p\text{)}$$
$$= \mathbf{ord}(pq) = \mathbf{ord}(d). \quad \blacksquare$$

From the above example we can see that even for the simplest FP, which consists of several lines, a proof of its correctness is not trivial mathematical reasoning, it is difficult to check it and much more difficult to construct it.

Below we present a radically different method for verification of FPs based on a construction of state diagrams for FPs, and illustrate it by a proof of (5) on the base of this method. This proof can be generated automatically, that is an evidence of advantages of the method for verification of FPs based on state diagrams.

4 State diagrams of functional programs

4.1 Concepts and notations related to terms

The following notations and concepts will be used below.

- $\mathcal{E}$ is a set of all terms.
- $\mathcal{E}_0$ is a set of all terms not containing functional variables.
- $\mathcal{E}_{conc}$ is a set of terms $e \in \mathcal{E}_0$, such that each FS occurring in e is *conc*.

- If Σ is a FP, then $\mathcal{E}_\Sigma$ is a set of terms, each of which is either a variable or has the form $\varphi(u_1, \ldots, u_n)$, where $\varphi \in \Phi_\Sigma$ and $u_1, \ldots, u_n \in \mathcal{E}_{conc}$.
- If $e \in \mathcal{E}$, $x_1, \ldots, x_n$ is a list of the different variables, and $e_1, \ldots, e_n$ are terms such that $\forall\, i = 1, \ldots, n$ $type(e_i) = type(x_i)$, then the notation

$$e(e_1/x_1, \ldots, e_n/x_n) \tag{17}$$

  denotes a term derived from e by replacement $\forall\, i \in \{1, \ldots, n\}$ of all occurrences of x_i in e with the term e_i.
- If e and e' are terms, then for each term e'', such that $type(e'') = type(e')$, the notation $e(e''/e')$ denotes a term derived from e by a replacement of all occurrences of e' in e with the term e''.
- An **assignment** is a notation of the form

$$u := e \tag{18}$$

  where $u \in \mathcal{E}_{conc}$, $e \in \mathcal{E}_\Sigma$, $type(u) = type(e)$.
- If $X \subseteq \mathcal{X}$, then an **evaluation** of variables occurring in X is a function ξ, which maps each variable $x \in X$ to a value $x^\xi \in \mathcal{D}_{type(x)}$. The set of all evaluations of variables occurring in X will be denoted by $X^\bullet$.
- For each $e \in \mathcal{E}_0$, each $X \supseteq X_e$ and each $\xi \in X^\bullet$ the notation e^ξ denotes an object called a **value** of e on ξ and defined by a standard way (i.e. if $e \in \mathcal{C}$, then e^ξ is equal to the value of the constant e, if $e \in \mathcal{X}$, then e^ξ is equal to the value of the evaluation ξ on the variable e, and if $e = f(e_1, \ldots, e_n)$, then $e^\xi = f(e_1^\xi, \ldots, e_n^\xi)$).
- We shall consider terms $e_1, e_2 \in \mathcal{E}_0$ as equal iff for each $\xi \in (X_{e_1} \cup X_{e_2})^\bullet$ the equality $e_1^\xi = e_2^\xi$ holds. We understand this equality in the following sense: values e_1^ξ and e_2^ξ either both undefined, or both defined and coincide.
- A term $e \in \mathcal{E}_0$ is called a **formula**, if all variables from X_e are of the type $\mathbf{C}$, and $\forall\, \xi \in X_e^\bullet \quad e^\xi \in \{0, 1\}$. The symbol $\mathcal{B}$ denotes the set of all formulas. The symbols $\top$ and $\bot$ denote formulas taking the values 1 and 0 respectively on each evaluation of their variables.

4.2 A concept of a state of a FP

Let Σ be a FP.

A **state** of Σ is a notation s of the form

$$[\![b]\!]\, u\, (\theta_1, \ldots, \theta_m) \tag{19}$$

components of which are the following:

- b is a formula from $\mathcal{B}$, called a **condition** of s,
- u is a term from $\mathcal{E}_{conc}$, called a **term related to** s, and
- $\theta_1, \ldots, \theta_m$ are assignments.

We shall use the following notations.

- S_Σ is the set of all states of Σ.
- If a state $s \in S_\Sigma$ is of the form (19), then we shall denote by b_s, u_s, Θ_s and $type(s)$ a formula b, a term u, a sequence of assignments (which can be empty) in (19), and a type $type(u)$, respectively.
  If $b_s = \top$, then the formula b in (19) will be omitted.
- If $s \in S_\Sigma$, then
  - X_s is a set of all data variables occurring in s,
  - each variable from X_s, occurring in the left side of an assignment from Θ_s, is called an **internal variable** of s, all other variables from X_s are called **input variables** of s,
  - $s^\bullet$ is a set of all $\xi \in X_s^\bullet$, such that $b_s^\xi = 1$, and $\forall\,(u_i := e_i) \in \Theta_s$
    - if $e_i \in \mathcal{E}_{conc}$, then $u_i^\xi = e_i^\xi$, and
    - if $e_i = \varphi(v_1, \ldots, v_n)$, then

$$u_i^\xi = f_\varphi(v_1^\xi, \ldots, v_n^\xi),$$

where f_φ is a corresponding component of a LFP of Σ.

A state $s \in S_\Sigma$ is said to be **terminal**, if Θ_s does not contain functional variables.

Given a pair of states $s_1, s_2 \in S_\Sigma$. We denote by the notation $s_1 \subseteq s_2$ the following statement: sets of input variables s_1 and s_2 are equal, and

$$\forall\,\xi_1 \in s_1^\bullet\ \exists\,\xi_2 \in s_2^\bullet : u_{s_1}^{\xi_1} = u_{s_2}^{\xi_2}.$$

Along with the states of FPs, we shall consider also **pseudo-states**, which differ from states only that their assignments have the form $u := e$, where $u \in \mathcal{E}_{conc}$, $e \in \mathcal{E}$. For each pseudo-state s the notations b_s, u_s and Θ_s have the same meaning as for states.

4.3 Unfoldinig of states

Let Σ be a FP, $s \in S_\Sigma$ be a state, $\theta \in \Theta_s$ be an assignment of the form

$$u := \varphi(v_1, \ldots, v_n)$$

and an equation in Σ that corresponds to φ has the form $\varphi(x_1, \ldots, x_n) = e_\varphi$.

Denote by s^θ a set, called an **unfolding** of the state s with respect to θ, and defined by the procedure of its construction, which consists of the steps listed below.

Step 1.

s^θ is assumed to be a singleton, which consists of a pseudo-state, derived from s by a replacement of θ with the assignment

$$u := e_\varphi(v_1/x_1, \ldots, v_n/x_n).$$

Step 2.

(This step can be performed several times until there is the possibility to perform it.)

If all the elements of the set s^θ are states from S_Σ, then the performance of this step ends, otherwise s^θ is modified in the following way.

We choose an arbitrary element $s' \in s^\theta$, which is not a state of S_Σ, and denote by θ' the first of the assignments, occurring in $\Theta_{s'}$, which has the form $u := e$, where $e \notin \mathcal{E}_\Sigma$. Consider all possible variants of the form of the term e, and for each of these variants, we present a rule of a modification of the set s^θ, according to this variant. Below, the phrase "a new variable" means "a variable that has no occurrences in the pseudo-state under consideration".

- $e \in \mathcal{C}$, in this case
  - if $u = e$, then remove θ' from s',
  - if $u \in \mathcal{X}$, then replace all occurrences of u in s' on e, and remove θ' from s',
  - otherwise remove s' from s^θ.
- $e = e'_h$, in this case replace θ' on the assignment
  - $u := e_1$, if e' has the form $e_1 e_2$,
  - $ux := e'$, where x is a new variable, otherwise.
- $e = e'_t$, in this case replace θ' on the assignment
  - $u := e_2$, if e' has the form $e_1 e_2$,
  - $xu := e'$, where x is a new variable, otherwise.
- $e = e_1 e_2$, in this case
  - if $u = u_1 u_2$, then replace θ' on a couple of assignments $u_1 := e_1$, $u_2 := e_2$,
  - if $u \in \mathcal{X}$, then replace all occurrences of u in s' on the term xy (where x and y are new variables), and θ' on the couple of assignments $x := e_1$, $y := e_2$,
  - otherwise remove s' from s^θ.
- $e = [\![e_1 = \varepsilon]\!]$, in this case
  - add to s^θ a copy of the state s' (denote it by s''),
  - replace
    - θ' in s' on the couple $u := 1$, $\varepsilon := e_1$, and
    - θ' in s'' on the couple $u := 0$, $xy := e_1$, where x and y are new variables.
- $e = [\![e_1 = e_2]\!]$, $e = [\![e_1 \leq e_2]\!]$, $e = [\![e_1 \wedge e_2]\!]$ etc., in this case
  - replace θ' on the couple $x_1 := e_1$, $x_2 := e_2$, where x_1, x_2 are new variables, and
  - add to $b_{s'}$ the conjunctive member $u = e'$, where e' is derived from e by a replacement of e_i with x_i $(i = 1, 2)$.
- $e = [\![e_1]\!]\, e_2 : e_3$, in this case add to s^θ a copy of s' (denote it by s''), and replace all occurrences
  - θ' in s' on the couple $1 := e_1$, $u := e_2$,
  - θ' in s'' on the couple $0 := e_1$, $u := e_3$.
- $e = \varphi(e_1, \ldots, e_k)$, $\exists\, i : e_i \notin \mathcal{E}_{conc}$, in this case, replace e_i in θ' on the new variable x, and add $x := e_i$ before θ'.

Step 3.

For each $s' \in s^\theta$

- if $\Theta_{s'}$ has a pair of the form $u := x$, $v := x$, where $x \in \mathcal{X}$, and u, v are of the form $u_1 \dots u_n$, $v_1 \dots v_m$ respectively, then there is executed an algorithm which consists of the following steps:
  (as a result of each of the these steps it is changed a form of these assignments, but we will denote the changed assignments by the same notation as original assignments):
  - if $n < m$, then in the case $u_n \in \mathcal{X}$ each occurrence of the variable u_n in s' is replaced on the term $v_n \dots v_m$, and in the case $u_n = \varepsilon$ we remove s' from s^θ,
  - analogously in the case $m < n$,
  - $\forall\, i = 1, \dots, n$:
    - if $u_i \in \mathcal{X}$, then replace all occurrences u_i in s' on v_i, and if $u_i \notin \mathcal{X}$, but $v_i \in \mathcal{X}$, then replace all occurrences v_i in s' on u_i,
    - if $u_i \neq v_i$, then remove s' from s^θ,
  - delete one of the considered assignments,
- if $b_{s'} = [\![b', x = u]\!]$, where $x \in \mathcal{X}$, $u \in \mathcal{X} \cup \mathcal{C}$, then $b_{s'}$ is replaced on b', and all occurrences x in s' are replaced on u,
- $b_{s'}$ is simplified by
  - a replacement of subterms without variables with corresponding constants, and
  - simplifying transformations related to boolean identities and properties of equality and linear order relations,
- if $b_{s'} = \bot$, then s' is removed from s^θ.

Theorem 1.
The above procedure for constructing of the set S^θ is always terminated. ■

A state $s \in S_\Sigma$ is **inconsistent**, if it is not terminal, and $\exists\, \theta \in \Theta_s$: either $s^\theta = \emptyset$, or all states in s^θ are inconsistent.

4.4 Substitution of states in terms

Let Σ be a FP, e be a term, $x_1, \dots, x_n$ be a list of different variables from $\mathcal{X}$, and $s_1, \dots, s_n$ be a list of states from S_Σ, such that $\forall\, i = 1, \dots, n \;\; type(s_i) = type(x_i)$. The notation

$$e(s_1/x_1, \dots, s_n/x_n) \tag{20}$$

denotes a state $s_e \in S_\Sigma$, defined by induction on the structure of e:

- if $e = x_i \in \{x_1, \dots, x_n\}$, then $s_e \stackrel{\text{def}}{=} s_i$,
- if $e \in \mathcal{X} \setminus \{x_1, \dots, x_n\}$ or $e \in \mathcal{C}$, then $s_e \stackrel{\text{def}}{=} e\,()$,
- if $e = g(e_1, \dots, e_k)$, where $g \in \mathcal{F} \cup \Phi$, and the states $s_{e_1}, \dots, s_{e_k}$ of the form (20), which are corresponded to terms $e_1, \dots, e_k$, are already defined, then s_e is defined as follows:

- internal variables of the states s_{e_i} are replaced on new variables by a standard way, so that all the internal variables of these states will be different, let $[\![b_i]\!]\, u_i(\Theta_i)$ $(i = 1, \ldots, k)$, be the resulting states,
- s_e is a result of an application of actions 2 and 3 from section (4.3) to the state

$$[\![b_1, \ldots, b_k]\!]\, (u_1, \ldots, u_k)\, (\Theta_1, \ldots, \Theta_k).$$

Term (20) will be denoted by the notation $e(s_1, \ldots, s_n)$, in that case, when the list of the variables $x_1, \ldots, x_n$ is clear from the context.

4.5 A concept of a state diagram of a FP

Let Σ be a FP, and left side of first equation in Σ has the form $\varphi(x_1, \ldots, x_n)$.

A **state diagram (SD)** of the FP Σ is a graph G with distinguished node n_0 (called an **initial node**) satisfying the following conditions.

– Each node n of the graph G is labelled by a state $s_n \in S_\Sigma$, and s_{n_0} has the form

$$y\,(y := \varphi(x_1, \ldots, x_n)), \quad \text{where } y \notin \{x_1, \ldots, x_n\}.$$

– For each node n of the graph G one of the following statements holds.

1. There is no an edge outgoing from n, and s_n is terminal.
2. There are two edges outgoing from n, and states s', s'' corresponding to the ends of these edges have the following property: $\exists\, x \in X_{s_n}$: $type(x) = \mathbf{S}$, there are no assignments of the form $u := x$ in Θ_{s_n}, and s', s'' are obtained from s_n by
   - a replacement of all occurrences of x with the constant ε and with the term yz respectively (where y and z are variables not occurring in X_{s_n}), and
   - if x is not occurring in the left side of any assignment from Θ_{s_n}, then – by adding assignments $\varepsilon := x$ and $yz := x$ to $\Theta_{s'}$ and $\Theta_{s''}$ respectively.
3. $\exists\, \theta \in \Theta_{s_n}$: a set of states corresponding to ends of edges outgoing from n, is equal to the set of all consistent states from s_n^θ.
4. u_{s_n} has the form $u_1 u_2$, and there is one edge outgoing from n labeled by $tail$, and the end n' of this edge satisfies the condition: $tail(s_n) \subseteq s_{n'}$.
5. There is an edge outgoing from n labelled by $<$, the end n' of which satisfies the condition:
   - $\exists\, n_1, n_2$: G contains an edge from n_1 to n_2 labelled by $tail$, and
   - $\exists\, e \in \mathcal{E}_\Sigma$, $\exists\, x \in X_e$:

$$s_n \subseteq e(tail(s_1)/x), \quad e(s_2/x) \subseteq s_{n'}.$$

We describe an informal sense of the concept of a SD. Each state s can be considered as a description of a process of a calculation of the value of the term u_s on concrete values of input variables of this state (by an execution of assignments from Θ_s, checking the condition b_s and a calculation of the value of the term

u_s on the calculated values of the variables occurring in this term). If all edges outgoing from the state n are unlabeled, then ends of these edges correspond to possible options for calculating the value of u_{s_n} (by detailization of a structure of a value of some variable from X_{s_n}, or by an equivalent transformation of any assignment from Θ_{s_n}). If there is an edge from n to n' labeled by *tail*, then this edge expresses a reduction of the problem of calculating of the tail of the value u_{s_n} to the problem of calculating the value of $u_{s_{n'}}$. If there is an edge from n to n' labeled by $<$, then this edge expresses a reduction of the problem of calculating the value u_{s_n} to the problem of calculating the value $u_{s_{n'}}$ on arguments on the smaller size.

We say that FP Σ **has a finite SD**, if there is a SD of Σ with finite set of nodes.

Theorem 2.

Let Σ_1 and Σ_2 have finite SDs, $\Phi_{\Sigma_1} \cap \Phi_{\Sigma_2} = \emptyset$, and left sides of first equations in Σ_1 and Σ_2 have the form $\varphi_1(x_1, \ldots, x_n)$ and $\varphi_2(y_1, \ldots, y_m)$ respectively, where $type(\Sigma_1) = type(y_1)$.

Then FP Σ such that

- its first equation has the form

$$\varphi(x_1, \ldots, x_n, y_2, \ldots, y_m) = \\ = \varphi_2(\varphi_1(x_1, \ldots, x_n), y_2, \ldots, y_m)$$

- and a set of other equations is $\Sigma_1 \cup \Sigma_2$

has a finite SD. ■

We do not give a description of the algorithm for the construction of a finite SD for Σ due to limitations on the size of the article. We note only that the SD is a union of a SD for Σ_1, a SD for Σ_2, and a SD, which is a Cartesian product of two previous SDs.

Theorem 3.

Let FP Σ has a finite SD, where terms, related to states corresponding to terminal nodes of this SD, which are reachable from an initial state, are equal to 1. Then f_Σ has value 1 on all its arguments. ■

The above theorems are theoretical foundation of a method of verification of FPs. This method consists in a constructing finite SDs

- for a FP Σ_1 under verification, and
- for a FP Σ_2 which represents some property of Σ_1.

If there are finite SDs for Σ_1 and Σ_2, then, according to Theorem 2, there is a finite SD for a superposition of Σ_1 and Σ_2. If this SD has the property indicated in Theorem 3, then the superposition of functions corresponding to Σ_1 and Σ_2, has the value 1 on all its arguments.

In the next section we present an example of this method.

For a constructing of SDs it is used a method of justification of statements of the form $s_1 \subseteq s_2$, which we did not set out here due to limitations on the size of the article. We only note that this method uses the concept of an unification of terms.

We shall use the following convention for graphical presentation of SDs: if a state s associated with a node of a SD has the form $[\![b]\!]\, u(\theta_1, \ldots, \theta_n)$, then this node is designated by an oval, over which it is drawn a notation $b.u$ (or u, if $b = \top$), and components of the list Θ_s are depicted inside the oval. An identifier of the node can be depicted from the left of the oval.

5 An example of verification of a FP by constructing of a state diagram

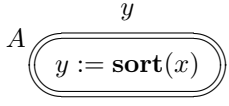
In this section we illustrate the verification method outlined above by an example of verification of FP of sorting, in this case $\Sigma_1 = (3)$ and $\Sigma_2 = (4)$.

We shall use the following convention: if nodes n_1 and n_2 of a SD are such that n_2 can be derived from n_1 by a performing of actions 2 and 3 from the definition of a SD, then we draw an unlabeled edge from n_1 to n_2 (i.e. unlabeled edges in a new understanding of a SD correspond to paths consisting of unlabeled edges in original understanding of a SD).

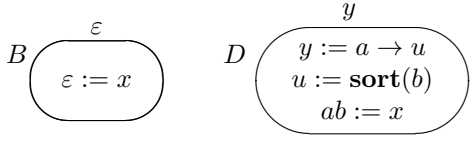
5.1 A state diagram for the FP of sorting

In this section we describe the process of building of a SD for FP (3). Terms of the form $\mathbf{insert}(a, y)$ we denote by $a \to y$.

An initial node of the SD for FP (3) (this node will be denoted by the symbol A) has the form



Two unlabeled edges can be drawn (corresponding to replacement of x with ε and with ab, and to an unfolding of one assignment) from this node to the nodes



Also it is possible to draw two unlabeled edges (corresponding to replacement of the variable u with constant ε and with the term cd) from D to nodes with labels

$$y\,(y := a \to cd, cd := \mathbf{sort}(b), ab := x), \tag{21}$$

and

$$y\,(y := a\varepsilon, \varepsilon := \mathbf{sort}(b), ab := x). \tag{22}$$

Also it is possible to draw two edges from the node labeled by (21) to nodes labeled by

$$\begin{array}{ll} C: & [\![a \le c]\!]\, acd\,(cd := \mathbf{sort}(b), ab := x), \\ G: & [\![c < a]\!]\, cz\,(z := a \to d, cd := \mathbf{sort}(b), ab := x) \end{array}$$

(by an unfolding of the first assignment).

It is possible to draw an edge labeled by *tail* from C to the initial node (the existence of such an edge is seen directly).

It is possible to draw two edges from the node labeled by (22) (replacing b to ε and to pq) to nodes, one of which is terminal and has the form

$$E: \quad a\varepsilon\ (a\varepsilon := x),$$

and the second node is inconsistent (that can be determined by additional unfoldings, which we do not present here).

It is possible to draw two unlabeled edges from G (corresponding to the replacement of b with ε and with pq) to nodes, one of which is inconsistent, and the second node is labeled by

$$[\![c < a]\!]\ cz \begin{pmatrix} z := a \to d, \\ cd := p \to w, \\ w := \mathbf{sort}(q), \\ apq := x \end{pmatrix}. \tag{23}$$

It is possible to draw two unlabeled edges from (23) (corresponding to the replacement of w with ε and ij):

– from the end of the first of these edges it can be drawn several unlabeled edges, but among ends of all these edges there is a unique consistent node labeled by

$$[\![c < a]\!]\, cz \begin{pmatrix} z := a \to \varepsilon, \\ c := p, \\ d := \varepsilon, \\ ap\varepsilon := x \end{pmatrix},$$

and there is a unique unlabeled edge from this node to a terminal node

$$H: \quad [\![c < a]\!]\, ca\varepsilon\,(ac\varepsilon := x),$$

– the end of second edge has a label

$$[\![c < a]\!]\ cz \begin{pmatrix} z := a \to d, \\ cd := p \to ij, \\ ij := \mathbf{sort}(q), \\ apq := x \end{pmatrix}. \tag{24}$$

It can be drawn a couple of edges from (24), the ends of which have labels

$$F: \quad [\![c < a, c \le i]\!]\ cz \begin{pmatrix} z := a \to ij, \\ ij := \mathbf{sort}(q), \\ acq := x \end{pmatrix},$$

$$I: \quad [\![c < a, c < p]\!]\ cz \begin{pmatrix} z := a \to d, \\ d := p \to j, \\ cj := \mathbf{sort}(q), \\ apq := x \end{pmatrix}.$$

It can be drawn an edge labeled by *tail* from F to the initial node (the existence of such an edge is seen directly).

A pair of nodes (D, I) is related to the pair of nodes (A, G) by the following relations:

$$tail(I) = e(tail(G)/h), \quad D = e(A/h) \tag{25}$$

where $e = a \to h$. In other words, labels of nodes I, D can be obtained from labels of nodes G, A by adding an assignment to the top. This fact can be used to justify an existence of an edge from G to A with label *tail*. We do not set out the detailed justification of an existence of such an edge, we describe only a scheme of such a justification. Let $\rho(x)$ be a partial function with the following property: if ρ is defined on a value α of the variable x, then it maps α to a string β, which has the property

$$u^{x \mapsto \alpha}_{tail(G)} = u^{x \mapsto \beta}_{A}.$$

The formula (25) directly implies the following property of the function ρ:

$$x \neq \varepsilon \ \Rightarrow\ \rho(x) \sqsupseteq x_h \rho(x_t) \tag{26}$$

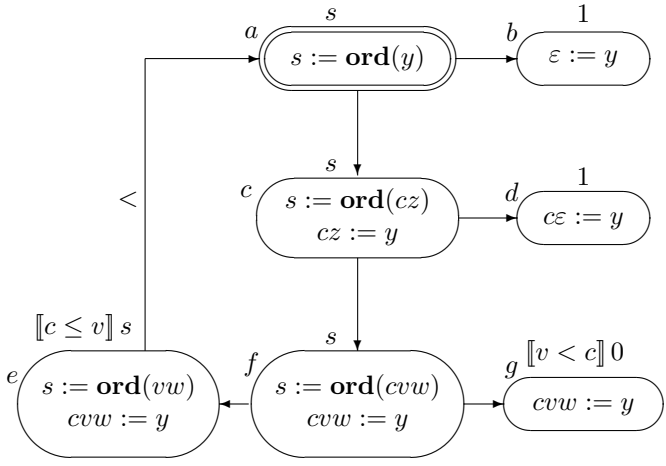
where the inequality $\sqsupseteq$ is understood as an order relation on the set of partial functions: if for some value of the variable x the right side of (26) is defined, then the left side also is defined for this value of x, and values of both parts are the same.

A property of totality of the function ρ is justified by the inequality (26) and by an analysis of a fragment of SD for (3) which is already built. Note that this justification can be generated automatically. A proof of correctness of this justification is based on the concept of unification of state pairs, it has a large volume, and we omit it.

The constructed SD for FP (3) is shown in Fig. 1. it can be simplified to the SD in Fig. 2 (we do not present here the detailed algorithm of this simplification).

5.2 A state diagram for the FP of cheking of string ordering

A fragment of a SD for FP Σ_2 (see (4)) (consisting of nodes reachable from the initial state) has the form



5.3 A state diagram for a superposition of the sorting FP and the FP of ordering checking

There is an algorithm based on Theorem 3, which can be applied to SDs for the FPs (3) and (4), which results the SD shown in Fig. 3. This SD has two terminal nodes, and labels of both these nodes have a value of 1. According to Theorem 3, this implies that the function **ord** ∘ **sort** has the value 1 on all its arguments.

In conclusion we note, that despite on the complexity of all of the above transformations and reasonings, all of them can be generated automatically. An attempt to justify an existence of edges with labels *tail* and $<$ can be executed automatically for each pair of nodes arising in the process of building of the SD. It can be seen from this example that the process of a construction of a SD is terminated fast enough.

6 Conclusion

We have proposed the concept of a state diagram (SD) for functional programs (FPs) and a verification method based on the concept of a SD. One of the problems for further research related to the concept of a SD has the following form: find a sufficient condition φ (as strong as possible) on a FP Σ such that if Σ meets φ then Σ has a finite SD.

References

1. 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.

2. C. A. R. Hoare: An axiomatic basis for computer programming. Communications of the ACM, 12(10): 576580, 583, October 1969.
3. R. Milner: A Calculus of Communicating Systems. Number 92 in Lecture Notes in Computer Science. Springer Verlag, 1980.
4. R. Milner: Communicating and Mobile Systems: the π-Calculus. Cambridge University Press, 1999.
5. Hoare, C. A. R.: Communicating sequential processes. Communications of the ACM 21 (8): 666677, 1978.
6. Separation Logic: A Logic for Shared Mutable Data Structures. John C. Reynolds. LICS 2002.
7. Clarke, E.M., Grumberg, O., and Peled, D.: Model Checking. MIT Press, 1999.
8. J.A. Bergstra, A. Ponse, and S.A. Smolka, editors: Handbook of Process Algebra. North-Holland, Amsterdam, 2001.
9. 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.
10. Z. Manna: Mathematical Theory of Computation. McGraw-Hill Series in Computer Science,1974.
11. N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. Theoretical Computer Science, 375:120136, 2007.
12. Ranjit Jhala, Rupak Majumdar, Andrey Rybalchenko: HMC: Verifying Functional Programs Using Abstract Interpreters, `http://arxiv.org/abs/1004.2884`
13. N. Kobayashi and C.-H. L. Ong. A type theory equivalent to the modal mu- calculus model checking of higher-order recursion schemes. In Proceedings of LICS 2009. IEEE Computer Society, 2009.
14. C.-H. L. Ong. On model- checking trees generated by higher order recursion schemes. In Proceedings 21st Annual IEEE Symposium on Logic in Computer Science, Seattle, pages 8190. Computer Society Press, 2006.
15. N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multiparameter tree transducers and recursion schemes for program verification. In POPL, pages 495508, 2010.

Fig. 1:

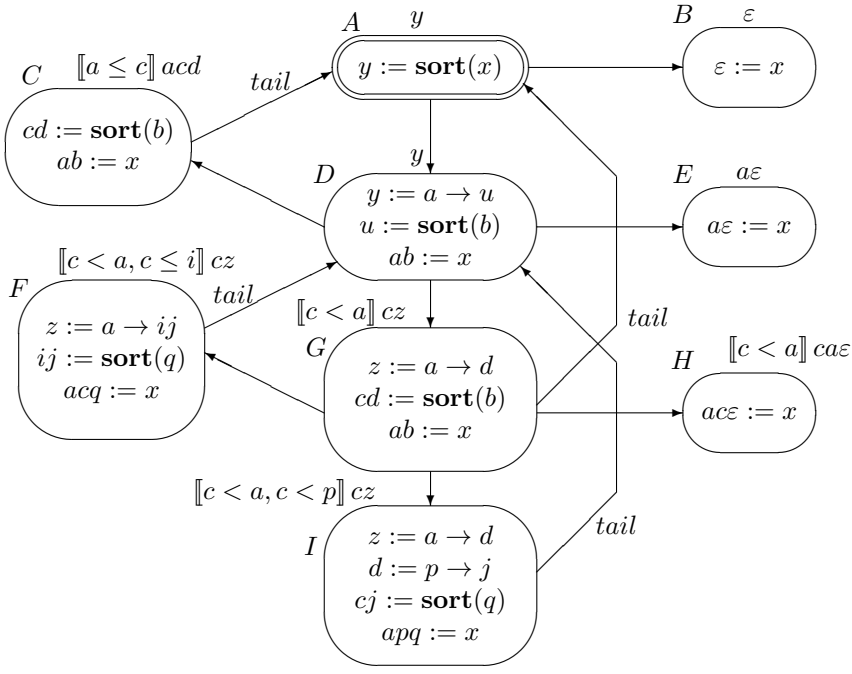

A
y
y := sort(x)
B
ε
ε := x
C
[a ≤ c] acd
cd := sort(b)
ab := x
tail
y
D
y := a → u
u := sort(b)
ab := x
E
aε
aε := x
[c < a, c ≤ i] cz
F
z := a → ij
ij := sort(q)
acq := x
tail
[c < a] cz
G
z := a → d
cd := sort(b)
ab := x
tail
H
[c < a] caε
acε := x
[c < a, c < p] cz
I
z := a → d
d := p → j
cj := sort(q)
apq := x
tail

Fig. 2:

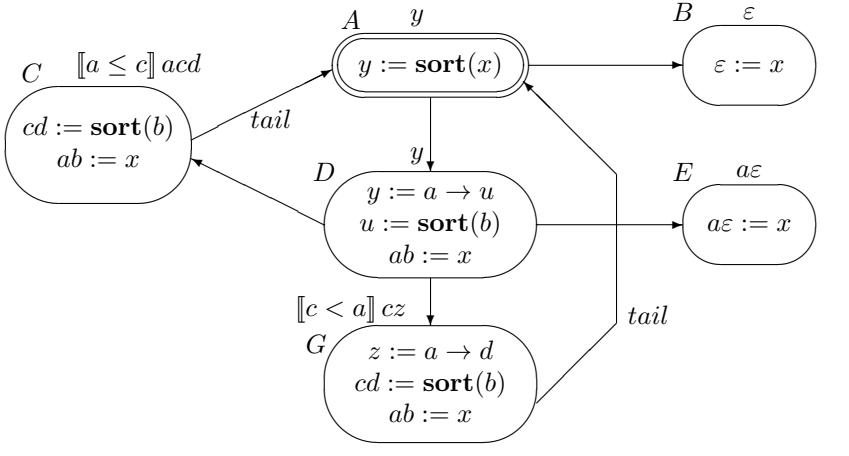
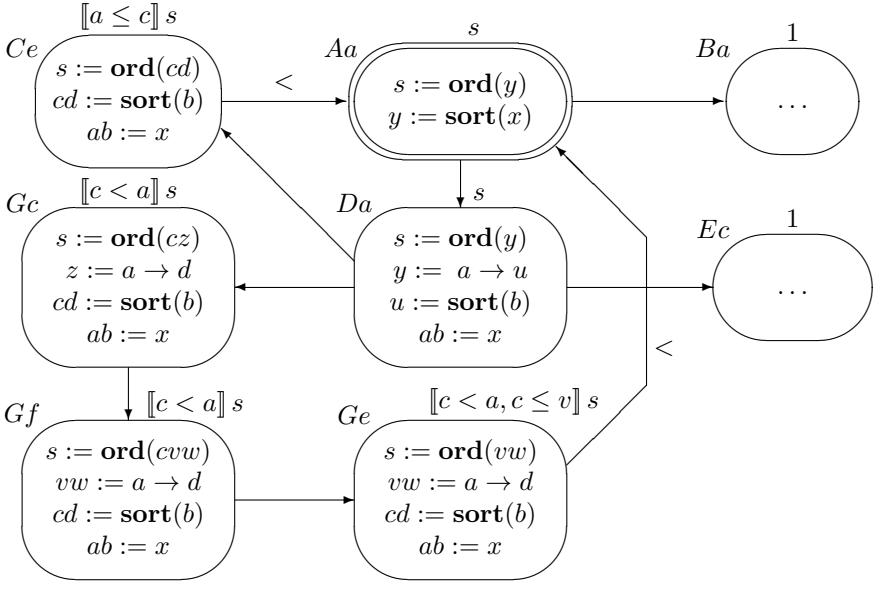

A
y
y := sort(x)
B
ε
ε := x
C
[a ≤ c] acd
cd := sort(b)
ab := x
tail
y
D
y := a → u
u := sort(b)
ab := x
E
aε
aε := x
[c < a] cz
G
z := a → d
cd := sort(b)
ab := x
tail

Fig. 3:


Ce
⟦a ≤ c⟧ s
s := ord(cd)
cd := sort(b)
ab := x
<
Aa
s
s := ord(y)
y := sort(x)
Ba
1
...
Gc
⟦c < a⟧ s
s := ord(cz)
z := a → d
cd := sort(b)
ab := x
Da
s
s := ord(y)
y := a → u
u := sort(b)
ab := x
Ec
1
...
<
Gf
⟦c < a⟧ s
s := ord(cvw)
vw := a → d
cd := sort(b)
ab := x
Ge
⟦c < a, c ≤ v⟧ s
s := ord(vw)
vw := a → d
cd := sort(b)
ab := x

Complexity of Turchin's Relation for Call-by-Name Computations

(Extended Abstract)

Antonina Nepeivoda
a_nevod@mail.ru

Program Systems Institute of Russian Academy of Sciences*
Pereslavl-Zalessky, Russia

Abstract. Supercompilation is a program transformation technique first described by V.F. Turchin in the 1970s. In supercompilation, Turchin's relation on call-stack configurations is used both for call-by-value and call-by-name semantics as a whistle. We give a formal grammar model of call-by-name stack behaviour and find the worst-case number of driving steps before the whistle using Turchin's relation is blown.

1 Introduction

Supercompilation is a program transformation method based on fold/unfold operations [2, 12, 14]. Given a program and its parametrized input configuration, a supercompiler partially unfolds the process tree of the program on the input configuration. Then it tries to fold the tree back into a graph, which presents the residual program. In general case, the process tree may be infinite. Thus, the following question appears: when is it reasonable to stop the unfolding in order to avoid going into an infinite loop?

One of the ways to solve this problem is based on "configuration similarity" relations. If a path in the tree contains two configurations, the latter of which resembles the former, that may be a sign that the path represents an unfolded loop. Thus, when a supercompiler finds two such configurations, it terminates unfolding of the path where they appear.

We recall an important relation property used for termination [5].

Definition 1 *Given a set T of terms and a set S of sequences of the terms from T, relation $R \subset T \times T$ is called* a well binary relation with respect to set S, *if every sequence $\{\Phi_n\} \in S$ such that $\forall i, j(i < j \Rightarrow (\Phi_i, \Phi_j) \notin R)$ is finite.*

A sequence $\{\Phi_n\}$ satisfying the property $\forall i, j(i < j \Rightarrow (\Phi_i, \Phi_j) \notin R)$ is called a bad sequence *with respect to R.*

* The reported study was partially supported by RFBR, research project No. 14-07-00133, and Russian Academy of Sciences, research project No. AAAA-A16-116021760039-0.

So, a well binary relation is "a well quasi-order without the order" (i. e., it is not necessarily transitive).

Any relation guaranteeing termination of the unfolding of a process tree must be a well binary relation with respect to the set of the traces generated in the tree. The relation most widely used for this aim, the homeomorphic embedding [1, 5, 10], is well binary with respect to arbitrary term sequences [4]. Some other relations used for termination in program transformations[1] are not well binary with respect to arbitrary term sequences. However, they are well binary with respect to the term sequences that can be generated on any computation path. In order to study termination properties of such relations, one must consider them together with their domain. Hence, the usual way of reasoning about the well-binariness of a relation as in [6][2] meets a couple of problems.

For Turchin's relation, well-binariness of which also can be proved only with respect to computation paths that appear during unfolding. That relation on call-stack configurations was the first well binary relation used for trace termination [14] (1986). Although Turchin's relation is a useful tool that helps to solve both termination and generalization problems [16], the proof of its well-binariness given by V. Turchin in [16] was presented in a semi-formal way. This work presents a formal approach to the theorem for computations in the call-by-name semantics. We introduce a notion of a multi-layer prefix grammar[3]. Elements of traces generated by such a grammar are models of call-stack configurations on computation paths in the call-by-name semantics. Based on the formalization, we could give a constructible proof of Turchin's theorem for the grammars being introduced. The constructible proof allowed us to find the upper bound on the bad sequence length with respect to Turchin's relation. The upper bound is Ackermanian.

In Section 2, we give an example showing how Turchin's relation can be efficiently used as a whistle. In Section 3 we give the formal definition for a class of grammars that model call stack behaviour for call-by-name computations. In Section 4 we very briefly show how such grammars can be used for modelling the call stack behaviour of programs in a simple functional language. Finally, in Section 5 we refine the definition of Turchin's relation for the new class of the grammars and state the result on the upper bound.

2 Turchin's Relation: an Example

The following program computes the sum of the squares starting from n down to 1 in a straightforward way.

[1] Among them is the relation used in supercompiler SCP4 [7] and the relation used in higher-order supercompiler HOSC [3].

[2] Including the "minimal bad sequence" method or methods using infinite Ramsey theorem.

[3] A precise description of the multi-layer grammars and their connection to call-stack configurations will be published in [9].

Example 1

A program computing $\sum_{k=1}^{n} k^2$
Start: $s(x)$;
$s(0) = 0;$ $s(x+1) = a(m(x+1, x+1), s(x));$
$a(0, y) = y;$ $a(x+1, y) = a(x, y) + 1;$
$m(0, y) = 0;$ $m(x+1, y) = a(y, m(x, y));$

When using a "core" homeomorphic embedding whistle, a supercompiler generates the residual program which repeats the initial program modulo renaming.

The program generated by a supercompiler which uses the composition of Turchin's relation and homeomorphic embedding as a whistle, is below.

Example 2

Residual program computing $\sum_{k=1}^{n} k^2$ *when Turchin's relation is used*
Start: $f(x, x, x)$;
$f(0, 0, 0) = 0;$ $f(0, 0, x+1) = f(x, x, x) + 1;$ $f(0, x+1, y) = f(y, x, y) + 1;$ $f(x+1, y, z) = f(x, y, z) + 1;$

The program given in Example 2 is more efficient than the initial one given in Example 1: on every step except the very last it adds 1 to the result, thus avoiding "zero" steps such as $m(0, y) = 0;$ or $a(0, y) = y;$. What properties of the refined whistle made this result possible?

The part of the process tree of the program in Example 1 is shown in Figure 1. When we use the homeomorphic embedding relation as a whistle when driving the initial configuration $s(x)$ to $s(x_1 + 1)$, the whistle is blown immediately after the substitution of the narrowing $x \to x + 1$ to the right-hand side of the definition $s(x+1) = a(m(x+1, x+1), s(x))$ because the subterm $s(x_1)$ repeats the initial configuration modulo the variables' names.

After the generalization to **let** $z = s(x_1)$ **in** $a(m(x_1 + 1, x_1 + 1), z)$, the process tree of the program is unfolded until the narrowing $x_1 \to x_2 + 1$ is done. Then the term $a(a(x_2, m(x_2 + 1, x_2 + 2)) + 1, z) + 1$, which is the result of the substitution of the narrowing in the term $a(a(x_1, m(x_1, x_1 + 1)), z) + 1$, embeds the parent term or its ancestor $a(m(x_1+1, x_1+1), z)$ (depending on the strategy of a supercompiler). The msg for both the pairs is $a(u, z)$. That is the cause why the residual program coincides with the one given in Example 1.

As opposed to the homeomorphic embedding relation, Turchin's relation considers only flat call-stack structure, ignoring all the passive data. In Figure 1,

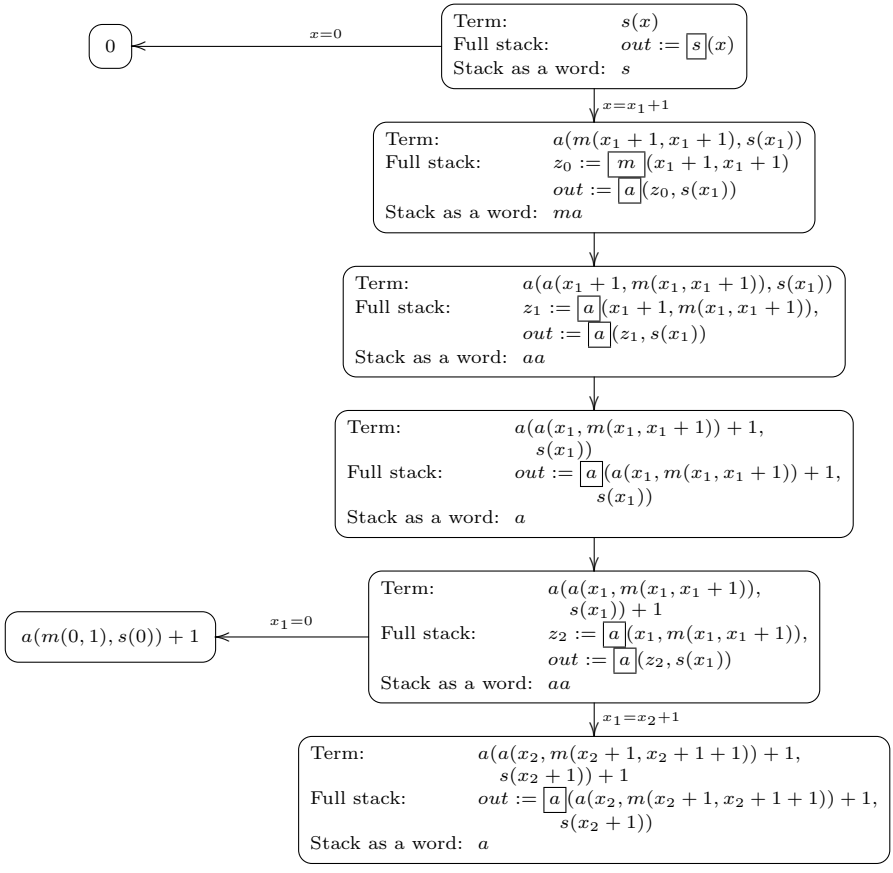


Fig. 1. A fragment of the process tree for the program of Example 1

the call-stack configurations are presented in the two forms: the full form that is constructed while interpreting the configuration, and the "word" form which contains *only* the names of the functions in the call-stack. Turchin's relation operates with the "word" forms. Namely, it checks whether the two "word forms" of the call stacks Δ_1 and Δ_2 on the path can be split into parts $[Top]$, $[Middle]$, and $[Context]$ such that $\Delta_1 = [Top][Context]$, $\Delta_2 = [Top][Middle][Context]$ and the part $[Context]$ is never changed on the path segment starting at Δ_1 and ending at Δ_2 (as it is shown in Figure 2).

Looking back to Figure 1, we can see that the first two configurations satisfying Turchin's relation are $a(a(x_1{+}1, m(x_1, x_1{+}1)), s(x_1))$ and $a(a(x_1, m(x_1, x_1{+}1)), s(x_1)){+}1$ (whose call-stacks in the "word form" look as aa). They are generalized by the most-specific generalization to $a(a(z, m(x_1, x_1 + 1)), s(x_1))$, which is much more specific than $a(u, z)$. We can notice that these two configurations

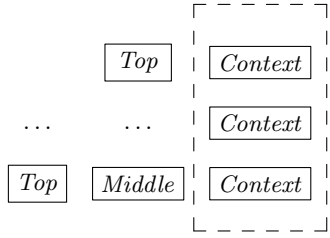


Fig. 2. Turchin's relation for call-stack configurations

do not satisfy the homeomorphic embedding relation, thus, when the composition of the relations is used as a whistle, another generalization is done. The first terms satisfying the both relations are $a(a(x_1, m(x_1, x_1 + 1)) + 1, s(x_1))$ and $a(a(x_2, m(x_2 + 1, x_2 + 1 + 1)) + 1, s(x_2 + 1)) + 1$. They are generalized to $a(a(u, m(w, w + 1)) + 1, s(w))$, which is also a good generalization with the substitutions containing no function calls. Further generalizations, which we do not discuss there, also allows a supercompiler to do driving on the whole term synchronously.

The examples above show that the composition of the homeomorphic embedding and Turchin's relation may be a better whistle than the homeomorphic embedding alone. On the one hand, they consider different properties of the configurations, which allows a supercompiler to build more specific generalizations. On the other hand, they have much in common, hence it is not likely that their composition will generate much longer bad sequences than the homeomorphic embedding alone. And even if they can sometimes generate very long bad sequences, one can have a desire that these situations will not appear too often.

3 Multi-layer Prefix Grammars

Based on the observations given in Section 2, we use the following assumptions to construct the grammar models for programs based on the call-by-name semantics.

1. A configuration can be considered as a tree of calls, and the active call stack — as a path in the tree. We use a set of labels $\mathbf{S}$ with partial order $\lhd$ for denoting the positions of the function calls in the tree.
2. Every call in the stack is modelled by a pair $<$ NAME, LABEL $>$, where LABEL $\in \mathbf{S}$.
3. Every configuration is represented as a word $\Gamma\$\Delta$ consisting of the two parts separated by the symbol \$. The structure of the active stack is placed in Γ and is linearly ordered w.r.t. labels, the function calls in the passive part of the configuration are placed in Δ.

Let Υ be a finite alphabet. Let $\mathbf{S}$ be *a label set* and $\lhd$ be a strict (non-reflexive) partial order relation over $\mathbf{S}$. We denote the labels from $\mathbf{S}$ by the

letters s, t (maybe with subscripts). Let us say that s_1 is a child of s_0 w.r.t. $\mathbf{S}' \subseteq \mathbf{S}$ (denoted by $s_1 = \text{child}(s_0)[\mathbf{S}']$) if $s_0 \lhd s_1$, $s_0 \in \mathbf{S}'$, $s_1 \in \mathbf{S}'$ and there is no such $s_2 \in \mathbf{S}'$ that $s_0 \lhd s_2$ and $s_2 \lhd s_1$. The inverse for the child relation is the parent relation. Given a set $\mathbf{S}' \subseteq \mathbf{S}$ and a label $t \in \mathbf{S} \setminus \mathbf{S}'$, we call t *a fresh label* w.r.t. $\mathbf{S}'$ if $\mathbf{S}'$ contains neither descendants nor ancestors of label t[4].

Henceforth, the set of finite sequences of pairs $\{\langle a, s_i\rangle | a \in \Upsilon \ \& \ s_i \in \mathbf{S}\}^*$ is denoted by $\mathsf{LW}(\Upsilon, \mathbf{S})$. Elements of $\mathsf{LW}(\Upsilon, \mathbf{S})$ are called *layered words*, and are denoted by Greek capitals Γ, Δ, Φ, Ψ, Ξ, Θ. If $\langle a_1, s_1\rangle \dots \langle a_n, s_n\rangle$ is a layered word, the corresponding *plain word* is defined as $a_1 \dots a_n$.

If Φ is a layered word, $|\Phi|$ stands for the number of the pairs in Φ and $\Phi[i]$ stands for the i-th pair. For the sake of brevity, layered word $\langle a_1, s_0\rangle \dots \langle a_n, s_0\rangle$ can be also written as $\langle a_1 \dots a_n, s_0\rangle$ (thus, $a\langle s_0\rangle$ is an equivalent form for $\langle a, s_0\rangle$).

Expression $\Phi\langle s_0\rangle$ denotes the maximal subsequence of Φ containing only the pairs labelled with s_0. The set of all labels in Φ is denoted by $\mathbf{S}_\Phi$.

Given a label s_i and natural numbers K_1 and K_2, we define a set of *layer functions* w.r.t. label s_i, $\mathfrak{F}^{s_i}_{K_1,K_2} : \mathsf{LW}(\Upsilon, \mathbf{S}) \to \mathsf{LW}(\Upsilon, \mathbf{S})$, as a minimal set of functions containing all compositions of K_1 elementary functions, which are:

1. Append $\text{App}^{s_j}[\Psi]$ (where $s_j \in \mathbf{S}$, $\Psi \in \Upsilon^*$): given a layered word Φ, the word $\text{App}^{s_j}[\Psi](\Phi)$ is the word $\Phi\Psi\langle s_j\rangle$ such that s_j is a child of s_i w.r.t. $\mathbf{S}_\Phi \cup \{s_j\}$, s_j is fresh w.r.t. $\mathbf{S}_\Phi \setminus \{s_i\}$, and $|\Psi| \leq K_2$.
   For example, if $\text{App}^{s_1}[g] \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \lhd s_1$, then
   $$\text{App}^{s_1}[g](\langle f, s_0\rangle\langle g, s_1\rangle) = \langle f, s_0\rangle\langle g, s_1\rangle\langle g, s_1\rangle$$
2. Insert $\text{Ins}^{s_j}[\Psi\langle s_k\rangle]$(where $s_j, s_k \in \mathbf{S}$, $\Psi \in \Upsilon^*$): given Φ with a non-empty $\Phi\langle s_j\rangle$, where s_j is a child of s_i w.r.t. $\mathbf{S}_\Phi$, $\text{Ins}^{s_j}[\Psi\langle s_k\rangle](\Phi)$ is the word $\Phi\Psi\langle s_k\rangle$ where $|\Psi| \leq K_2$ and s_k is a child of s_i w.r.t. $\mathbf{S}_\Phi \cup \{s_k\}$, s_k is fresh w.r.t $\mathbf{S}_\Phi \setminus \{s_i\}$ and s_j is a child of s_k w.r.t $\mathbf{S}_\Phi \cup \{s_k\}$.
   For example, if $\text{Ins}^{s_1}[gf\langle s_2\rangle] \in \mathfrak{F}^{s_0}_{1,1}$[5] and $s_0 \lhd s_1$, then
   $$\text{Ins}^{s_1}[\langle gf, s_2\rangle](\langle f, s_0\rangle\langle g, s_1\rangle) = \langle f, s_0\rangle\langle g, s_1\rangle\langle gf, s_2\rangle$$
   The insert operation differs from the append operation only by introduction of an unused child label s_k, which marks the newly appended word Ψ.
3. Deleting Del^{s_j} (where $s_j \in \mathbf{S}$): given Φ with a non-empty $\Phi\langle s_j\rangle$, $s_j = \text{child}(s_i)$ w.r.t. $\mathbf{S}_\Phi$, Del^{s_j} erases $\Phi\langle s_j\rangle$ from Φ together with all $\Phi\langle t\rangle$ for which $s_j \lhd t$.
   For example, if $\text{Del}^{s_{01}} \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \lhd s_{01}$, s_{02} is incomparable with s_{01}, then
   $$\text{Del}^{s_{01}}(\langle d, s_{01}\rangle\langle d, s_{02}\rangle) = \langle d, s_{02}\rangle$$
4. Copying Copy^{s_j} (where $s_j \in \mathbf{S}$): given Φ with a non-empty $\Phi\langle s_j\rangle$, $s_j = \text{child}(s_i)$ w.r.t. $\mathbf{S}_\Phi$, Copy^{s_j} appends $\Phi\langle s_k\rangle$ to Φ, where s_k is a child of s_i w.r.t.

[4] In most cases, we assume that $\mathbf{S}'$ is a set of all previously used labels, hence there is no need to write it in the square brackets in expressions like $\text{child}(s_0)[\mathbf{S}']$.

[5] This condition implies that $s_2 \lhd s_1$ and $s_0 \lhd s_2$.

$\mathbf{S}_\Phi \cup \{s_j\}$, s_j is fresh w.r.t. $\mathbf{S}_\Phi \setminus \{s_i\}$, and then it appends all subsequences $\Phi\langle s_l\rangle$ labelled by the children of s_j and labels them by fresh children of s_l and so on until all the sequences $\Phi\langle t\rangle$, where $s_j \triangleleft t$, are copied exactly once. For example, if $\mathrm{Copy}^{s_{01}} \in \mathfrak{F}^{s_0}_{1,1}$ and $s_0 \triangleleft s_{01}$, then

$$\mathrm{Copy}^{s_{01}}(\langle d, s_{01}\rangle) = \langle b, s_0\rangle\langle d, s_{01}\rangle\langle d, s_{02}\rangle,$$

where s_{02} is incomparable with s_{01}.

Definition 2 *Let us consider a tuple* $\mathbf{G} = \langle \Upsilon, \mathbf{S}, \mathbf{R}, \mathfrak{F}^v_{K_1,K_2}, \Gamma_0\$\Delta_0\rangle$ *where* Γ_0 *and* Δ_0 *are layered words over* $\Upsilon \times \mathbf{S}$ *such that for every* $\Gamma_0[i] = \langle a_i, s_i\rangle$ *and* $\Gamma_0[j] = \langle a_j, s_j\rangle$*, if* $j > i$ *then* $s_j \triangleleft s_i$ *or* $s_j = s_i$*,* \$ *is a special symbol,* $\$ \notin \Upsilon$*, and* $\mathfrak{F}^v_{K_1,K_2}$ *is a finite set of layer function forms where* v *runs over the label set* $\mathbf{S}$*. For every* $\mathbf{G}$*-word* $\Gamma\$\Delta$*, where* Γ *and* Δ *are words in* $\mathsf{LW}(\Upsilon, \mathbf{S})$*, we call* Γ the visible layer*, and we call* Δ the invisible layer *of* $\Gamma\$\Delta$*.*

Let all rewriting rules from $\mathbf{R}$ *have one of the following forms:*

- *Simple rule:*
$$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Phi\Theta\$F^{s_i}(\Psi),$$
where all the letters of Φ *are labelled either by* s_i *or by fresh descendants of* s_i*,* $F^{s_i} \in \mathfrak{F}^{s_i}$*.*
- *Pop rule: for* $\Psi\langle s_j\rangle$ *— the maximal subsequence of* Ψ *marked by some* $s_j = child(s_i) \in \mathbf{S}$*,*
$$\Xi\langle a, s_i\rangle\Theta\$\Psi \to \Psi\langle s_j\rangle\Phi\Theta\$F^{s_i}(\Psi),$$
where all the letters of Φ *are labelled either by* s_i *or by fresh descendants of* s_i*,* $F^{s_i} \in \mathfrak{F}^{s_i}$*. In a pop rule, we may specify* s_j*, but there are no ways to specify* $\Psi\langle s_j\rangle$*.*

Such a grammar $\mathbf{G}$ *is called a multi-layer prefix grammar.* K_2 *is called the maximal rewrite depth. A sequence of* $\mathbf{G}$*-words starting at* $\Gamma_0\$\Delta_0$ *that are transformed by the rules from* $\mathbf{R}$ *is called a trace of* $\mathbf{G}$*.*

If any rule of such a grammar changes only one letter of the visible layer, then the multi-layer prefix grammar is alphabetic*.*

Words on the traces generated by the alphabetic multi-layer grammars are models of the call-stack configurations that appear on the path of the process tree during the call-by-name computations. Some pointers to a way for constructing these models explicitly are given in the next section.

4 Modelling Call Stack Behaviour by Multi-layer Grammars

We borrow the notions of f-function and g-function from [11] and use them in the following sense. An f-function is a function whose definition consists of a one rule with the trivial patterns (e. g., if h_1 is defined as $h_1(x_1, x_2) = x_2 + h_2(x_1 + 1)$

then h_1 is an f-function). A g-function is a function with non-trivial patterns in the definition (e. g., $h_2(x+1) = h_2(x) + h_2(x)$ is a definition of the g-function).

In order to get a grammar from a program, we treat every configuration generated by the unfolding as a tree, whose nodes are named by function or constructor names and leaves contain no function calls. First, we mark every function name in the tree by a superscript depending on the state of the function call. If the call is ready to be unfolded without unfolding of other calls, the function name is marked as "ready" (by + in the superscript). Otherwise, the function name is marked as "unready". Hence, the call names of all f-functions are always marked as "ready", while the call names of g-functions are marked as "ready" if their pattern can be matched without evaluating another call[6].

Then we delete all the nodes containing static data[7]. The remaining nodes are given the layer labels. If some node T is a descendant of a node W, the label of T is greater than the label of W. Otherwise the labels are incomparable.

Finally, we find all the nodes containing the unready call names with a single child. The child of such a node is given the label of the node. And then, all the nodes with the same labels are merged: data from the ancestor nodes are placed in the merged node after the data from their descendants.

The resulting tree is a tree form of the corresponding layered word.

Example 3 *Given the term $a(m(x_1+1, x_1+1), s(x_1))$, we transform it to a layered word. All the steps of the transformation are given in Figure 3.*

First, we mark the calls as "ready"(with + in the superscript) and "unready"(with − in the superscript), and delete the nodes with the static data. The only function call in the configuration which is ready to be unfolded without unfolding is the outermost call of m. The call $s(x_1)$ requires unfolding (which generates the narrowing on x_1), but does not require unfolding of other calls, so it is also marked as ready. The remaining call a is marked as unready.

Then we assign the layer labels in the resulting tree of the marked call names. The tree below shows that $s_0 \lhd s_{01}$, $s_0 \lhd s_{02}$.

Finally, the nodes containing the names of the calls in the active stack are extracted. These names together with the node labels take a place in the visible part of the layered word; data from the remaining node in the tree take a place in the invisible part.

---

[6] There we always can determine all the calls that are ready to be unfolded due to simplicity of the patterns. In languages with complex pattern matching (e. g., Refal [15]), that can be done only if one knows the strategy of the pattern matching applied in the interpreter.

[7] In some cases, this action can transform the tree into a forest. For example, that can happen if the configuration is $cons(h_1(x), cons(h_2(x), Nil))$. To avoid these cases, we always assume that the transformed tree has a root, but the root is a "virtual" function call, which is always present in the **G**-word corresponding to the tree and is denoted by \$.

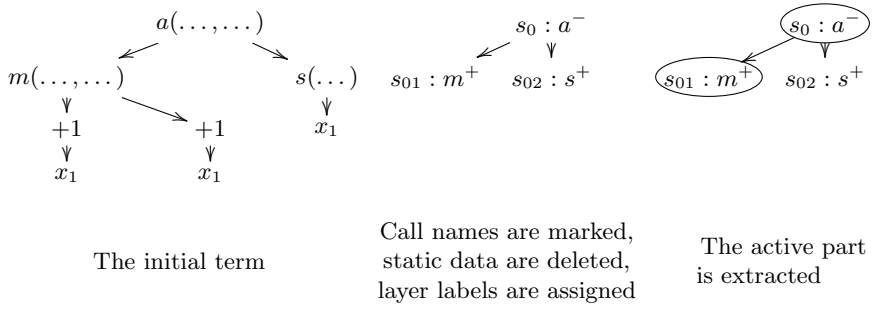


Fig. 3. Steps transforming $a(m(x_1 + 1, x_1 + 1), s(x_1))$ from a tree to layered word $\langle m, s_{01}\rangle\langle a, s_0\rangle\$\langle s, s_{02}\rangle$

5 Turchin's Relation and Multi-Layer Grammars

Definition 3 *Let* $\mathbf{G}$ *be a multi-layer prefix grammar with the set of rules that rewrite* N *letters in the visible layer. Given a trace* $\{\Gamma_k\$\Delta_k\}$ *and its segment* $[i, j]$*, suffix* Θ *of* Φ_i *is called* a permanently stable *suffix w.r.t. the segment* $[i, j]$ *if all the words* $\Gamma_k\$\Delta_k$*,* $i \le k < j$*, are of the form* $\Phi_k\Theta\$\Delta_k$ *where* Φ_k *is a layered word with the length not less than* N*, and* Γ_j *is of the form* $\Phi_j\Theta$*, where* Φ_j *may be* Λ [8]*. If* j *is not bounded,* Θ *is called* a permanently stable suffix *w.r.t.* i*.*

Informally, a permanently stable suffix is a suffix of the visible layer that is never changed in the trace segment starting at the i-th and ending by the j-th $\mathbf{G}$-word in the trace. In the terms of call stack behaviour, a permanently stable suffix corresponds to an unchanged context of the computation.

Definition 4 *Let* $\mathbf{G} = \langle \Upsilon, \mathbf{S}, R, \mathfrak{F}^v_{K_1,K_2}, \Gamma_0\$\Delta_0\rangle$ *be a multi-layer prefix grammar. Given two* $\mathbf{G}$*-words* $\Xi_i = \Gamma_i\$\Delta_i$*,* $\Xi_j = \Gamma_j\$\Delta_j$ *in a trace* $\{\Gamma_k\$\Delta_k\}$*, we say that the words form* a Turchin pair *(denoted as* $\Xi_i \preceq \Xi_j$*) if* $\Gamma_i = \Phi\Theta_0$*,* $\Gamma_j = \Phi'\Psi\Theta_0$*,* Φ *is equal to* Φ' *as a plain word (up to the layer labels) and the suffix* Θ_0 *is permanently stable w.r.t. segment* $[i, j]$*.*

Lemma 1 *Let us consider the Ackermann function defined as follows:*

$$B_K(M, 0) = 1$$
$$B_K(0, N) = N + 1$$
$$B_K(M, N) = B_K(M - 1, B_K(M, N - 1) * K)$$

An alphabetic multi-layer grammar $\mathbf{G}$ *can generate bad sequences w.r.t. Turchin's relation not longer than* $B_K(M, N)$*, where* K *is the maximal rewrite depth of* $\mathbf{G}$*,* M *is the number of rules in the set of rewriting rules of* $\mathbf{G}$ *and* N *is the total length of the initial word in the grammar.*

[8] In the case of alphabetic prefix grammars, when $N = 1$, the first condition implies the second.

Example 4 *A program that generates traces which are Ackermanian bad sequences w.r.t. Turchin's relation can be as follows. The input point of the program is* $A(< N, b(B(< 1, 0 >)) >)$ *(where* N *is an arbitrary fixed natural number).*

$$
\begin{aligned}
&A(< x_1 + 1, x_2 >) = a(A(< x_1, x_2 >));\\
&A(< 0, x_2 >) =< x_2 + 1, 0 >;\\
&a(< x_1 + 1, x_2 >) = x_1;\\
&B(< x_1 + 1, x_2 >) = c(c(< x_1 + 1, x_2 >));\\
&b(< x_1 + 1, x_2 >) = x_2;\\
&c(< x_1 + 1, x_2 >) =< B(b(< x_1, x_2 >)) + 1, c(c(< x_1, x_2 >)) >;\\
&c(< 0, x_2 >) =< B(b(< 1, 0 >)) + 1, c(c(< 0, 0 >)) >;
\end{aligned}
$$

The program never stops and its call-stack configurations form bad sequences of the exponential tower length (in N*) with respect to Turchin's relation (that is, of the length* $O(2^{2^{\cdot^{\cdot^{\cdot^{2}}}}\}N})$*). However, with respect to the homeomorphic embedding relation over the entire terms, a computation of this program for every* $N > 0$ *is terminated on the* $5 + N$*-th step.*

6 Conclusion

Turchin's relation for call-by-name computations is a strong and consistent branch termination criterion, which finds pairs of embedded terms on the trace of every infinite computation. It allows a program transformation tool to construct very long configuration sequences (i. e., traces) with no Turchin pairs in them, and although such sequences appear in real program runs almost never, the computational power of Turchin's relation shows that the relation can be used to solve some complex problems. In fact, the Ackermanian upper bound on the bad sequence length is rather a "good" property indicating that the relation is non-trivial, than a "bad" one indicating that the whistle using the relation will be blown too late. Example 4 shows a program that generates very long bad sequences. But the program contains an implicit definition of the Ackermanian function. From the practical point of view, such programs are very rare. If a program does not generate such complex structures, its call-stack configuration structures considered by Turchin's relation can be modelled by a grammar belonging to a smaller class than the whole class of the multi-layer prefix grammars. E.g., for the call-by-value semantics, this class coincides with the regular grammars [8]. Primitively recursive functions also cannot generate too complex stack structures: they are even incapable to compute such fast-growing functions as the Ackermanian function. Moreover, the homeomorphic embedding can produce even longer bad sequences than Turchin's relation [13, 17]. Thus, the property being described in this paper is not a thing Turchin's relation must be blamed of.

References

1. Albert, E., Gallagher, J., Gomes-Zamalla, M., Puebla, G.: Type-based homeomorphic embedding for online termination. Journal of Information Processing Letters 109(15), 879–886 (2009)
2. Hamilton, G.W., Jones, N.D.: Distillation with labelled transition systems, pp. 15–24. IEEE Computer Society Press (2012)
3. Klyuchnikov, I.: Inferring and proving properties of functional programs by means of supercompilation. Ph. D. Thesis (in Russian) (2010)
4. Kruskal, J.: Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. Transactions of the American Mathematical Society 95, 210–225
5. Leuschel, M.: Homeomorphic Embedding for Online Termination of Symbolic Methods, Lecture Notes in Computer Science, vol. 2566, pp. 379–403. IEEE Computer Society Press (2002)
6. Nash-Williams, C.S.J.A.: On well-quasi-ordering infinite trees. Proceedings of Cambridge Philosophical Society 61, 697–720 (1965)
7. Nemytykh, A.P.: The Supercompiler Scp4: General Structure. URSS, Moscow (2007), (In Russian)
8. Nepeivoda, A.: Turchin's relation and subsequence relation in loop approximation. In: PSI 2014. Ershov Informatics Conference. Poster Session. vol. 23, pp. 30–42. EPiC Series, EasyChair (2014)
9. Nepeivoda, A.: Turchin's relation for call-by-name computations: a formal approach. (to appear) (2016)
10. Sørensen, M.H., Glück, R.: An algorithm of generalization in positive supercompilation. In: Proceedings of ILPS'95, the International Logic Programming Symposium. pp. 465–479. MIT Press (1995)
11. Sørensen, M.: Turchin's supercompiler revisited. Ms.Thesis (1994)
12. Sørensen, M., Glück, R., Jones, N.D.: A positive supercompiler. Journal of Functional Programming 6, 811–838 (1996)
13. Touzet, H.: A characterisation of multiply recursive functions with higman's lemma. Information and Computation 178, 534–544 (2002)
14. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (1986)
15. Turchin, V.F.: Refal-5, Programming Guide and Reference Manual. New England Publishing Co., Holyoke, Massachusetts (1989), electronic version:`http://www.botik.ru/pub/local/scp/refal5/`
16. Turchin, V.: The algorithm of generalization in the supercompiler. Partial Evaluation and Mixed Computation pp. 341–353 (1988)
17. Weiermann, A.: Phase transition thresholds for some friedman-style independence results. Mathematical Logic Quarterly 53, 4–18 (2007)

Algorithm Design Patterns: Programming Theory Perspective

(Extended Abstract)

Nikolay V. Shilov

A.P. Ershov Institute of Informatics Systems, Russian Academy of Sciences
Lavren'ev av. 6, 630090 Novosibirsk, Russia
shilov@iis.nsk.su
http://persons.iis.nsk.su/en/person/shilov

Abstract. Design and Analysis of Computer Algorithms is a must of Computer Curricula. In particular it teaches algorithm design patterns like *greedy method*, *divide-and-conquer*, *dynamic programming*, *backtracking* and *branch-and-bound*. Unfortunately, all listed design patterns are taught, learned and comprehended by examples, while they can be formalized as design templates, rigorously specified, and mathematically verified. Greedy method is the only pattern that had been studied from rigour mathematical point of view in XX century. Later, the author published (in years 2010-2012 in separate papers) formalization, specification and verification for three more patterns, namely *dynamic programming*, *backtracking* and *branch-and-bound*. In the present extended abstract these studies are summarized and discussed from programming theory perspective using concepts and techniques used in *Abstract Data Types*, *Theory of Program Schemata*, *Partial and Total Correctness*, *program specialization*.

To commemorate 85 anniversary of A.P. Ershov (1931-1988), a Scientist that drew my interest to Theory of Programming.

1 Introduction

Algorithm design patterns (ADP) like *greedy method*, *divide-and-conquer*, *dynamic programming* (DYN), *backtracking* (BTR) and *branch-and-bound* (B&B) are usually considered as Classics of the Past (going back to days of R. Floyd and E. Dijkstra). However, ADP can be (semi)formalized as design templates, specified by correctness conditions, and formally verified either in the Floyd-Hoare methodology, by means of the Manna-Pnueli proof-principles, or in some other way.

Nevertheless until 2010 the only formalized method of algorithm design was greedy method (or greedy algorithms): it was proven in years 1971-1993 [5,13,15] that if the structure of the domain in an optimization problem is a *matroid* (or, is more general, *greedoid*), then application of greedy algorithm guarantees a global optimum for the problem.

Unfortunately, further progress with DYN, BTR and B&B techniques has degenerated into an extensive collection of "success stories" and "recipes" how they have been used in the context of particular combinatorial or optimization problems. This leads to educational situation when the most popular contemporary textbooks on the algorithm design and implementation look like Cooking Books [1, 4].

BTR and B&B are widely used in design of combinatorial algorithms for (virtual) graph traversing. In particular, most global optimization methods using interval techniques employ a branch-and-bound strategy [9] These algorithms decompose the search domain into a collection of boxes, arrange them into a tree-structure (according to inclusion), and compute the lower bound on the objective function by an interval technique. Basically the strategy is an algorithm design pattern that originates in graph traversal.

In general graph traversal refers to the problem of visiting all the nodes in a (di)graph to find particular nodes (vertices) that enjoy some property specified by some Boolean "criterion condition". A Depth-first search (DFS) is a technique for traversing a finite graph that visits the child nodes before visiting the sibling nodes. A Breadth-first search (BFS) is another technique for traversing a finite undirected graph that visits the sibling nodes before visiting the child nodes.

Sometimes it is not necessary to traverse all vertices of a graph to collect the set of nodes that meet the criterion function, since there exists some Boolean "boundary condition" which guarantees that child nodes do not meet the criterion function: Backtracking (BTR) is DFS that uses boundary condition, branch-and-bound (B&B) is DFS that uses boundary condition. Backtracking became popular in 1965 due to research of S.W. Golomb and L.D. Baumert [11], but it had been suggested earlier by D. H. Lehmer. Branch-and-bound was suggested by A.H. Land and A.G. Doig in 1960 [18].

Formalization and verification of backtracking and branch-and-bound ADP was attempted in years 2011-2012: a unified ADP for BTR and B&B was formalized as a design template, specified by correctness conditions, and formally verified by means of the Manna-Pnueli proof-principles first [20] and later in the Floyd-Hoare methodology [23].

Dynamic Programming was introduced by Richard Bellman in the 1950s [3] to tackle optimal planning problems. At this time, the noun *programming* had nothing in common with more recent *computer programming* and meant *planning* (compare: *linear programming*). The adjective *dynamic* points out that Dynamic Programming is related to a *change of state* (compare: *dynamic logic*, *dynamic system*). *Bellman equations* are recursive functional equalities for the objective function that express the optimal solution at the current state in terms of optimal solutions at changed states. They formalize the following *Bellman Principle of Optimality*: an optimal program (or plan) remains optimal at every stage.

At the same time, according to [7], R. Bellman, speaking about the 50s, explains:

> *An interesting question is, "Where did the name,* dynamic programming*, come from?" The 1950s were not good years for mathematical research.*

(...) Hence, I felt I had to do something to shield [the Secretary of Defense] and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. (...) Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

A preliminary formalization of Dynamic Programming has been published in [21] and then in [22]. The approach

- formalizes *descending* Dynamic Programming ADP by a *recursive program scheme* (with variable arity of functional symbols),
- formalizes *ascending* Dynamic Programming as a *standard program scheme* (also with variable arity of functional symbols) augmented by *generic dynamic array* to compute the least fix-point (according to the Knaster-Tarski theorem),
- proves functional equivalence of both schemes, and proves that in general case we can't rid of *dynamic memory* when implement the ascending dynamic programming pattern.

This extended abstract represents

in the next section 2: the unified template for Backtracking and Branch-and-Bound, its (semi-)formal specification and main correctness statements (as in [20, 23]);

in the section 3: the unified templates for descending and ascending Dynamic Programming, their functional equivalence statement (as in [21, 22]).

The final section 4 discusses in brief examples of use of the templates and further research topics (in particular, *dynamic memory* issues and *partial evaluation* perspective).

2 Template for Backtracking and Branch-and-Bound

2.1 Abstract Data Type "Teque"

Let us define a special temporal abstract data type (ADT) "theque[1]" for the unified representation of BTR and B&B. Theque is a finite collection (i. e. a set) of values (of some background data type) marked by disjoint "time-stamps". The time stamps are readings of "global clock" that counts time in numbers of "ticks", they (time-stamps) never change and always are not greater than current reading of the clock. Let us represent an element x with a time-stamp t

[1] Theque – storage (Greek: theke), e. g. "discotheque".

by the pair (x, t). Readings of the clock as well as time-stamps are not "visible" for any "observer". Let us assume that this "tick" is indivisible, every action takes a positive (integer) number of ticks, and the clocks never resets or restarts.

ADT theque inherits some set-theoretic operations: the *empteq* (i. e. "empty teque")) is simply the empty set ($\emptyset$), set-theoretic equality ($=$) and inequality ($\neq$), subsumption ($\subset$, $\subseteq$). At the same time ADT theque has its own specific operations, some of these operations are time-independent, some others — time-sensitive, and some are time-dependent. Let us enumerate below time-independent operations, and describe time-dependent and time-sensitive operations in the next paragraphs.

- Operation Set: for every teque T let $Set(T)$ be $\{x \ : \ \exists t((x,t) \in T)\}$ the set of all values that belongs to T (with any time-stamp).
- Operations In and Ni: for every teque T and any value x of the background type let $In(x, T)$ denote $x \in Set(T)$, and let $Ni(x, T)$ denote $x \notin Set(T)$.
- Operation $Spec$ (*specification*): for every teque T and any predicate $\lambda x.Q(x)$ of values of the background type let teque $Spec(T, Q)$ be the following subteque $\{(x,t) \in T \ : Q(x)\}$.

The unique time-dependent operation is a synchronous addition $AddTo$ of elements to teques. For every finite list of teques T_1, ... T_n ($n \geq 1$) and finite set $\{x_1, \ ... \ x_m\}$ of elements of the background type ($m \geq 0$), let execution of $AddTo(\{x_1, \ldots x_m\}, \ T_1, \ldots T_n)$ at time t (i. e. the current reading of the clock is t) returns n teques T'_1, ... T'_n such, that there exist m moments of time (i. e. readings of the clock) $t = t_1 < ... < t_m = t'$ such that t' is the moment of termination of the operation, and for every $1 \leq i \leq n$ the teque T'_i expands T_i by $\{(x_1, t_1), \ ... \ (x_m, t_m)\}$, i. e. $T'_i = \ T_i \cup \ \{(x_1, t_1), \ ... \ (x_m, t_m)\}$. Let us observe that this operation is non-deterministic due to several reasons: first, the set of added elements $\{x_1, \ ... \ x_m\}$ can be sorted in different manners; next, time-stamps $t_1 < ... < t_m$ can be arbitrary (starting at the current time). Let us write $AddTo(x, T_1, \ldots T_n)$ instead of $AddTo(\{x\}, T_1, \ldots T_n)$ in the case of a singleton set $\{x\}$.

There are three pairs of time-sensitive operations: Fir and $ReMFir$, Las and $RemLas$, Elm and $RemElm$. Let T be a teque. Recall that all values in this teque have disjoint time-stamps.

- Let $Fir(T)$ be the value of the background type (i. e. without a time-stamp) that has the smallest (i. e. the first) time-stamp in T, and let $RemFir(T)$ be the teque that results from T after removal of this element (with the smallest time-stamp).
- Let $Las(T)$ be the value of the background type (i. e. without a time-stamp) that has the largest (i. e. the last) time-stamp in T, and let $RemLas(T)$ be the teque that results from T after removal of this element (with the largest time-stamp).

We also assume that $Elm(T)$ is "some" element of T (also without any time-stamp) that is defined according to some "procedure" (unknown for us) and $RemElm(T)$ is the teque that results from T after removal of this element (with its time-stamp).

2.2 Unified Template

Let us introduce some notation that unifies representation of BTR and B&B by a single template for graph traversing: let FEL and REM stay either for Fir and $ReMFir$, or for Las and $RemLas$, or for Elm and $RemElm$. It means, for example, that if we instantiate Fir for FEL, then we must instantiate Fir for FEL and $RemFir$ for REM throughout the template. Instantiation of Fir and $RemFir$ imposes a queue discipline "first-in, first-out" and specializes the unified template to B&B template; instantiation of Las and $RemLas$ imposes a stack discipline "first-in, last-out" and specializes the template to BTR template; instantiation of Elm and $RemElm$ specializes the unified template to "Deep Backtracking" or "Branch and Bounds with priorities" templates.

Let us say that a (di)graph is concrete, if it is given by the enumeration of all vertices and edges, or by the adjacency matrix, or in any other explicit manner. In contrast, let us say that a (di)graph G is virtual, if the following features are given:

- a type $Node$ for vertices of G, the initial vertex ini (of this type) such that every vertex of G is reachable from ini;
- a computable function $Neighb: \ Node \to 2^{Node}$ that for any vertex of G returns the set of all its neighbors (children in a digraph).

In this notation a unified template for traversing a virtual graph G with the aid of "easy to cheque"

- a boundary condition $B: \ 2^{Node} \times Node \ \to BOOLEAN$,
- and a decision condition $D: \ 2^{Node} \times Node \ \to BOOLEAN$

for collecting all nodes that meet a "hard to cheque"

- criterion condition $C : Node \to BOOLEAN$

can be represented by the following pseudo-code.

```
VAR U: Node;
VAR S: set of Node;
VAR Visit, Live, Out: teque of Node;
Live, Visit:= AddTo(ini, empteq, empteq);
Out:= empteq; IF D({ini}, ini) THEN Out:= AddTo(ini, Out);
WHILE Live ≠ empteq
      DO U:= FEL(Live); Live:= REM(Live);
        S:= {W ∈ Neighb(U) : Ni(W, Visit) & ¬B(Set(Visit), W)};
        Live, Visit:= AddTo(S, Live, Visit);
        Out:= Spec(Out, λx.D(Set(Visit), x));
        IF D(Set(Visit), U) THEN Out:= AddTo(U,Out);
      OD
```

2.3 Correctness

An algorithm without specification is a tool without manual: no idea how to use it and what to expect. A specified algorithm without correctness proof is a non-certified tool, it can be dangerous in use. So we have to specify and prove correctness of our unified template. We would like to use Floyd – Hoare approach to algorithm specification and proof [2, 10]. In this approach an algorithm is specified by a precondition and a postcondition for input and output data, correctness is proved with the aid of loop invariants by induction.

The postcondition is simple: Teque Out consists of all nodes of the graph G (with time-stamps) that meet the criterion condition C, and each of these nodes has a single entry (occurrence) in Out.

The precondition is more complicated, and can be presented as a conjunction of the following clauses.

1. G is a virtual (di)graph, ini is a node of G, $Neighb$ is a function that computes for every node the set of all its neighbors so, that all nodes of G can be reached from ini by iterating $Neighb$.
2. For every node x of G the boundary condition $\lambda S.B(S,x)$ is a monotone function: $B(S_1,x) \Rightarrow B(S_2,x)$ for all sets of nodes $S_1 \subseteq S_2$ (i. e. if a node is ruled-out, then it is ruled-out forever).
3. For all nodes x and y of G, for any set of nodes S, if y is reachable from x, then $B(S,x)$ implies $B(S,y)$ (i. e. if a node is ruled-out then all its successors are ruled out also).
4. For every node x of G, the decision condition $\lambda S.D(S,x)$ is an anti-monotone function: $D(S_2,x) \Rightarrow D(S_1,x)$ for all sets of nodes $S_1 \subseteq S_2$ (i. e. a candidate node may be discarded later).
5. For every set of nodes S, if $S \cup \{x \in G : B(S,x)\}$ is equal to the set of all nodes of G, then $D(S,x) \Leftrightarrow C(x)$ (i. e. the decision condition D applied to a set with "complete extension" is equivalent to the criterion condition C).

Proposition 1. *The unified template is partially correct with respect to the above precondition and postcondition, i. e. if the input data meet the precondition and a particular algorithm instantiated from the template terminates on the input data, then it terminates with the output that meets the postcondition.*

Proposition 2. *If the input graph is finite then the unified template eventually terminates, i. e. every particular algorithm instantiated from the template always halts traversing the graph after a finite number of steps.*

The above two propositions imply the following total correctness statement for Backtracking and Branch-and-Bound.

Corollary 1.
If the input data (including the boundary, decision and criterion conditions B, D and C) meet the precondition, and the virtual graph G for traversing is finite, then every particular algorithm instantiated from the template terminates after $O(|G|)$ iterations of the loop, so that upon termination the set $Set(Out)$ will consist of all nodes of the graph G that meet the criterion condition C.

Remark 1. For proofs please refer papers [20, 23].

3 Templates for Dynamic Programming

3.1 Recursive Descending Dynamic Programming

If to analyse Bellman principle then it is possible to suggest the following *recursive scheme* as a general pattern for Bellman equations:

$$G(x) \;=\; if\ p(x)\ then\ f(x)\ else\ g(x,\ h_i(G(t_i(x)),\ i \in [1..n(x)])), \qquad (1)$$

where

- G is a function variable to represent an objective function from some domain X to some range Y that are to be optimized;
- p is a predicate symbol to represent a known predicate over the same domain X as above;
- f is a function symbol to represent a known function from the same domain X to the same range Y;
- g is a function symbol to represent a known operation with a variable arity on the same domain X (i.e. a function from X^* to X);
- all h_i and t_i, $i \in \mathbb{N}$, are functional symbols to represent known functions from the range Y to the domain X (in case of h_i) and to represent known operations on the domain X (in case of t_i).

Here we understand the recursive scheme in the sense of the *theory of program schemata* [8, 16, 19]. Let us refer the above recursive scheme as a *recursive template for descending Dynamic Programming.*

3.2 Iterative Ascending Dynamic Programming

Let us consider a function $G : X \to Y$ that is defined by the interpreted recursive scheme (1) of Dynamic Programming. For every argument value $v \in X$, such that $p(v)$ doesn't hold, let *base* be the following set $bas(v)$ of values $\{t_i(v) : i \in [1..n(v)]\}$. Let us remark that for every argument value v, if $G(v)$ is defined, $bas(v)$ is finite. Let us also observe that if the objective function G is defined for some argument value v, then it is possible to *pre-compute* (i.e. compute prior to the computation of $G(v)$) the *support* for this argument value v, i.e. the set $spp(v)$ of all argument values that occur in the recursive computation of $G(v)$, according to the following recursive algorithm

$$spp(x) \;=\; if\ p(x)\ then\ \{x\}\ else\ \{x\} \cup (\bigcup_{y \in bas(x)} spp(y)). \qquad (2)$$

Another remark is that for every argument value v, if $G(v)$ is defined, then $spp(v)$ is finite (since computation of $G(v)$ terminates). Let us say that a function $SPP : X \to 2^X$ is an *upper support approximation* if for every argument value v, the following conditions hold:

- $v \in SPP(v)$,

– $spp(u) \subseteq SPP(v)$ for every $u \in SPP(v)$,
– if $spp(v)$ is finite then $SPP(v)$ is finite.

Let us consider the case when some upper approximation is *easier to compute*, i.e. the (time and/or space) complexity of the available algorithm to compute it is better than the complexity of the available algorithm that computes G. Then it makes sense to use *iterative ascending Dynamic Programming*.

Ascending Dynamic Programming comprises the following steps.

1. Input argument value v and compute $SPP(v)$. Let G be an array (in Pascal style) $var\ G:\ Y\ array\ of\ SPP(v)$ of Y-values indexed by values in $SPP(v)$. Then compute and save in the array G values of the objective function G for all arguments $u \in SPP(v)$ such that $p(u)$: $G[u] := f(u)$.
2. Expand the set of saved values of the objective function by values that can be immediately computed on the basis of the set of saved values: for every $u \in SPP(v)$, if $G(u)$ has not been computed yet, but for every $w \in bas(u)$ the value $G(w)$ has already been computed and saved in $G[w]$, then compute and save $G(u)$ in $G[u]$: $G[u] :=\ g(u,\ (h_i(G(t_i(u))),\ i \in [1..n(u)]))$.
3. Repeat Step 2 until the moment when the value of the objective function for the argument v is saved.

The ascending Dynamic Programming is an imperative iterative procedure.

3.3 Formalization

Let us formalize iterative ascending Dynamic Programming by means of an imperative pseudo-code annotated by precondition and postcondition [10].

Precondition:
D is a non-empty set of argument values,
S and P are "trivial" and "target" subsets in D,
$F : 2^D \to 2^D$ is a call-by-value total monotone function,
$\rho : 2^D \times 2^D \to Bool$ is a call-by-value total function monotone on the second argument.

Peseudo-code:

```
VAR U= S, V: subsets of D;
REPEAT V:= U; U:= F(V)∪S UNTIL (ρ(P,U) or U=V)
```

Postcondition: $\rho(P, U) \Leftrightarrow \rho(P, T)$,
where U is the final value of the variable `U` upon termination
and T is the least set of D such that $T = (F(T) \cup S)$.

Here the initialization `U=` S corresponds to the first step of the informal description of ascending Dynamic Programming, the second assignment `U:=` F`(V)` in the loop body corresponds to the second step, and the loop condition ρ`(P,U)` corresponds to the condition at the third step; an auxiliary variable `V`, the first assignment `V:= U` and condition `U=V` are used for termination in case when no further progress is possible.

We would like to refer to this formalization as the *iterative ascending Dynamic Programming template*.

3.4 Correctness and Equivalence

Proposition 3. (Knaster-Tarski fix-point theorem [14])
Let D be a non-empty set, $G : 2^D \to 2^D$ be a total monotone function, and R_0, R_1, ... be the following sequence of D-subsets: $R_0 = \varnothing$ and $R_{k+1} = G(R_k)$ for every $k \geq 0$. Then there exists the least fix-point $T \subseteq D$ of the function G and $R_0 \subseteq R_1 \subseteq R_2 \subseteq \ldots R_k \subseteq R_{k+1} \subseteq \ldots \subseteq T$.

The following two propositions are trivial consequences of the above one.

Proposition 4.
Iterative ascending Dynamic Programming template is partially correct.

Proposition 5. *Assume that for some input data the precondition of the iterative ascending Dynamic Programming template is valid and the domain D is finite. Then the algorithm generated from the template terminates after $|D|$ iterations of the loop.*

In turn they imply the following (functional) equivalence statement for recursive (descending) and iterative (ascending) Dynamic Programming.

Corollary 2.
Let G be an arbitrary function defined by interpreted recursive scheme 1. Let

- *D be a* generic dynamic array $\{(u, G(u)) : \ u \in SPP(v)\}$*, where $SPP(v)$ is an upper support approximation;*
- *S be $\{(u, f(u)) : \ p(u)$ and $u \in SPP(v)\}$ and P be a singleton $\{(v, G(v))\}$;*
- *F be $\lambda Q \subseteq D.\ \{(u, w) \in D \mid n = n(u),$*
  $$\exists w_1, \ldots w_n : (t_1(u), w_1), \ldots (t_n(u), w_n) \in Q,$$
  $$\text{and } w = g(u, h_1(w_1), \ldots h_n(w_n))\};$$
- *ρ be $\lambda Q.(P \subseteq Q)$ (that is equivalent to $\lambda Q \subseteq D.(\exists w : (v, w) \in Q)$).*

Then, the algorithm resulting from the iterative ascending Dynamic Programming template after the specified specialization is totally correct, the final value U of the variable `U` contains a single pair (v, y), and y in this pair equals to $G(v)$.

Remark 2. For proofs please refer paper [22].

4 Conclusion

We have represented in this extended abstract the unified template for Backtracking and Branch-and-Bound, and templates for recursive (descending) and iterative (ascending) Dynamic Programming, specified these templates by means of (semi-formal) precondition and postcondition, stated the total correctness and functional equivalence (for Dynamic Programming) for the templates. "Manual" formal proofs of propositions and corollaries stated in this extended abstract can be found in papers [20–23].

In the cited papers [20–23] one can find the following examples of specialisation of the presented templates:

- in papers [20, 23] unified BTR and B&B template was specialized to solve Discrete Knapsack Problem,
- in papers [21, 22] Dynamic Programming templates were specialised to solve a toy sample Dropping Bricks Puzzle as well as for solving finite position games and to Cocke-Younger-Kasami algorithm for parsing of context-free language.

Basically, the primary purpose of the *specified* and *verified* templates for algorithm design patterns is to use them for (semi-)automatic *specialization* of the patterns to generate correct but more *efficient* algorithms to solve concrete problems. One may observe that this purpose is closely related to purpose of *Mixed Computations* [6] and/or *Partial Evaluation* [12]; the difference consists in level of consideration: in our case we speak about *algorithm design* and use *pseudo-code* while in Mixed Computations and Partial Evaluation *programming languages* and *program code* are in use. Nevertheless further studies of algorithm design templates from Mixed Computations and Partial Evaluation perspective may be an interesting research topic. In particular, it may be interesting to try to building-in algorithm design templates into an educational IDE (Integrated Development Environment) to support (semi-)automatic algorithm generation.

Another interesting topic for theoretical research is a need of *dynamic memory* either for teque or generic array implementation. In particular, we demonstrated that every function defined by the recursive scheme of Dynamic Programming may be computed by an iterative program with a *generic dynamic array*. The advantage of the translation is the use of an *array* instead of a *stack* generally required to translate recursion. Nevertheless a natural question arises: may *finite static memory* suffices for computing this function? Unfortunately, in general case the answer is negative according to the following proposition proved by M.S. Paterson and C.T. Hewitt [16, 19].

Proposition 6. *The following special case of general recursive scheme of descending Dynamic Programming (1)*

$$F(x) = \ if\ p(x)\ then\ x\ else\ f(F(g(x)), F(h(x)))$$

is not equivalent to any standard program scheme (i.e. an uninterpreted iterative program scheme with finite static memory).

This statement does not mean that dynamic memory is *always* required; it just means that for *some* interpretations of *uniterpreted* symbols p, f, g and h the size of required memory depends on the input data. But if p, f, g and h are *interpreted*, it may happen that function F can be computed by an iterative program with a finite static memory. For example, it is possible to prove that Dropping Bricks Puzzle [22] can be solved by an iterative algorithm with two integer variables. Two other examples of this kind are the factorial function and Fibonacci numbers

$$Fac(n) = \ if\ n = 0\ then\ 1else\ n \times Fac(n-1),$$

$$Fib(n) = \ if\ n = 0\ or\ n = 1\ then\ 1\ else\ Fib(n-2) + Fib(n-1);$$

they both match the pattern of scheme in the above proposition, but three integer variables suffice to compute them by iterative programs. So, the next problem for further research is about those of functions that can be computed with finite static memory (i.e. like the optimal number of bricks droppings, factorial values or Fibonacci numbers) by iterative imperative algorithms generated from ascending dynamic programming.

Another research topic is about use of different fix-points in the Dynamic Programming context. A unified logical approach to a variety of fix points can be found in [17]; a natural question follows: what if to use for algorithm design other fix-points (studied in the cited paper) than the least one?

References

1. Aho A.V., Hopcroft J. E., Ullman J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
2. Apt K.R., de Boer F.S., Olderog E.-R. *Verification of Sequential and Concurrent Programs.* Third edition. Springer, 2009.
3. Bellman, R. *The theory of dynamic programming.* Bulletin of the American Mathematical Society, 1954, v.60, p.503-516.
4. Cormen T.H., Leiserson C.E., Rivest R.L., and Stein C. *Introduction to Algorithms.* Third edition. The MIT Press, 2009.
5. Edmonds J. *Matroids and the greedy algorithm.* Mathematical Programming, 1971, v.1, p.127113.
6. Ershov A.P. *Mixed computation: potential applications and problems for study.* Theor. Comp. Sci., 1982, v18(1), p.41-67.
7. Gimbert, H. *Games for Verification and Synthesis.* Slides for 10th School for young researchers about Modelling and Verifying Parallel processes (MOVEP). — `http://movep.lif.univ-mrs.fr/documents/marta-slides1.pdf`.
8. Greibach, S.A. *Theory of Program Structures: Schemes, Semantics, Verification.* Springer, 1975. (Lecture Notes in Computer Science, v.36.)
9. Gray P., Hart W., Painton L., Phillips C., Trahan M., Wagner J. *A Survey of Global Optimization Methods.* Sandia National Laboratories, 1997, available at `http://www.cs.sandia.gov/opt/survey/main.html`.
10. Gries D. *The Science of Programming.* Springer, 1987.
11. Golomb S.W. and Baumert L.D. *Backtrack Programming.* Journal of ACM, 12(4), 1965, p.516-524.
12. Jones J.D., Gomard C.K., and Sestoft P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993, available at `http://www.itu.dk/people/sestoft/pebook/`.
13. Helman P., Moret B.M.E., Shapiro H.D. *An exact characterization of greedy structures.* SIAM Journal on Discrete Mathematics, 1993, v.6(2), p.274283.
14. Knaster, B., Tarski, A. *Un theoreme sur les fonctions d'ensembles.* Ann. Soc. Polon. Math., 1928, n.6, p.133-134.
15. Korte B., Lovasz L. *Mathematical structures underlying greedy algorithms.* Fundamentals of Computation Theory: Proceedings of the 1981 International FCT-Conference, Szeged, Hungaria, August 2428, 1981, Lecture Notes in Computer Science, 1981, v.117, p.205209.

16. Kotov V.E., Sabelfeld V.K. *Theory of Program Schemata.* (Teoria Skhem Programm.) Science (Nauka), 1991. (In Russian.)
17. Lisitsa A. *Temporal Access to the Iteration Sequences: A Unifying Approach to Fixed Point Logics.* Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck , Germany, September 12-14, 2011. IEEE Computer Society, 2011, p.57-63.
18. Land A. H. and Doig A. G. *An automatic method of solving discrete programming problems.* Econometrica, 28(3), 1960, p.497-520.
19. Paterson M.S., Hewitt C.T. *Comperative Schematology.* Proc. of the ACM Conf. on Concurrent Systems and Parallel Computation, 1970, p.119-127.
20. Shilov, N.V. *Algorithm Design Template base on Temporal ADT.* Proceedings of 18th International Symposium on Temporal Representation and Reasoning, 2011. IEEE Computer Society. P.157-162.
21. Shilov N.V. *Inverting Dynamic Programming.* Proceedings of the Third International Valentin Turchin Workshop on Metacomputation, 2012. Publishing House University of Pereslavl. P.216-227.
22. Shilov N.V. *Unifying Dynamic Programming Design Patterns.* Bulletin of the Novosibirsk Computing Center (Series: Computer Science, IIS Special Issue), v.34, 2012, p.135-156.
23. Silov N.V. *Verification of Backtracking and Branch and Bound Design Templates.* Automatic Control and Computer Sciences, 2012, v.46(7), p.402409.

Author Index

Научное издание

Труды конференции

Сборник трудов Пятого международного семинара по метавычислениям имени В. Ф. Турчина, г. Переславль-Залесский, 27 июня – 1 июля 2016 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. **10,2**. Усл. печ. л. **10,7**. Подписано к печати **10.06.2016**. Ответственный за выпуск: *С. М. Абрамов*.

Издательство университета города Переславля

Отпечатано в ООО «Регион». Печать цифровая. Бумага офсетная. Тираж 150 экз. Заказ 4.
152025 Ярославская область, г. Переславль-Залесский, ул. Строителей, д. 41.