Russian Academy of Sciences
Ailamazyan Program Systems Institute

Third International Valentin Turchin Workshop on Metacomputation

Proceedings
Pereslavl-Zalessky, Russia, July 5–9, 2012

Издательство Университетский город Переславль
Pereslavl-Zalessky

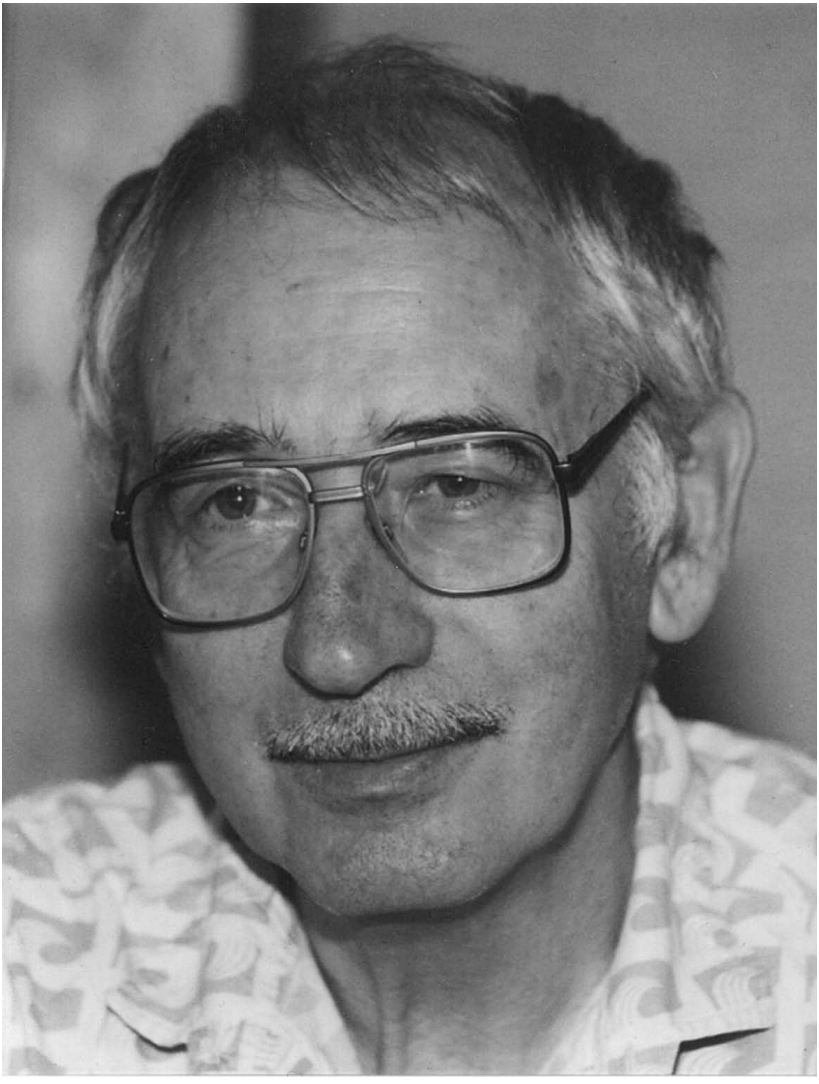
УДК 004.42(063)
ББК 22.18

Т66

Third International Valentin Turchin Workshop on Metacomputation // Proceedings of the Third International Valentin Turchin Workshop on Metacomputation. Pereslavl-Zalessky, Russia, July 5–9, 2012 / *Edited by A. V. Klimov and S. A. Romamenko.* — Pereslavl Zalessky: Publishing House "University of Pereslavl", 2012, **260** p. — ISBN 978-5–901795-28-6

Третий международный семинар по метавычислениям имени В. Ф. Турчина // Сборник трудов Третьего международного семинара по метавычислениям имени В. Ф. Турчина, г. Переславль-Залесский, 5–9 июля 2012 г. / *Под редакцией А. В. Климова и С. А. Романенко.* — Переславль-Залесский: Изд-во «Университет города Переславля», 2012, **260** с. *(англ).* — ISBN 978-5–901795-28-6

ISBN 978-5–901795-28-6



Valentin Turchin
(1931–2010)

Preface

The proceedings contain the papers presented at the Third International Valentin Turchin Workshop on Metacomputation, META 2012, held on July 5–9, 2012 in Pereslavl-Zalessky.

The workshop belongs to a series of workshops organized biannually by Ailamazyan Program Systems Institute of the Russian Academy of Science and Ailamazyan University of Pereslavl in an ancient Russian town Pereslavl-Zalessky.

The first workshop in 2008 was blessed by the creator of supercompilation Valentin Turchin, who did not come personally, but was very glad to hear that supercompilation and other metacomputation topics attracted more and more attention of the computer science community.

The second workshop in 2010 was held soon after Valentin Turchin's departure and was devoted to his memory. At that time we observed a burst of work on supercompilation and related techniques. We are pleased to mention that since then two authors who presented their work at META 2010, Jason Reich and Ilya Klyuchnikov, have completed and defended their PhD theses on the topics of further development of supercompilation.

From now on this series of workshops is named after Valentin Turchin.

Metacomputation is a research area related to deep program analysis and transformation. The primary topics of the META 2012 workshop are multi-result supercompilation and distillation, which are further developments of supercompilation. Other papers and talks address program inversion and its logics, research into practical metacomputation and applications as well as metaprogramming in Scala.

Multi-result supercompilation realizes the idea that while solving various problems by supercompilation from program optimization to proving properties of programs, there should be considered a set of residual programs equivalent to the source one, rather than a single one produced by using a fixed supercompilation strategy.

- Ilya G. Klyuchnikov and Sergei A. Romanenko. *Formalizing and Implementing Multi-Result Supercompilation.* Explains the principles of multi-result supercompilation and discusses some details related to its implementation.
- Sergei A. Grechanik. *Overgraph Representation for Multi-Result Supercompilation.* Suggests how to make multi-result supercompilation more efficient with the use of a compact representation of graphs of configurations.
- Andrei V. Klimov, Ilya G. Klyuchnikov and Sergei A. Romanenko. *Automatic Verification of Counter Systems via Domain-Specific Multi-Result Supercompilation.* Demonstrates how multi-result supercompilation improves previous results on application of supercompilation to verification of models expressed as counter systems.
- Andrei V. Klimov. *Why Multi-Result Supercompilation Matters: Case Study of Reachability Problems for Transition Systems.* Revisits and analyses the

method of solving the reachability problems by supercompilation and another method codenamed "Expand, Enlarge and Check" from the viewpoint of multi-result supercompilation.

Distillation is a result of a metasystem transition from plain supercompilation to a hierarchy of program transformers.

- Neil D. Jones and G. W. Hamilton. *Superlinear Speedup by Program Transformation.* Poses the problem of going beyond the classic positive supercompilation which can achieve only linear speedup and states the goal of finding an essential "inner core" of distillation, which gives rise to this improvement.
- G. W. Hamilton. *A Hierarchy of Program Transformers.* Presents the main idea of distillation as building a hierarchy of program transformers in which the transformer at each level of the hierarchy builds on top of the transformers at lower levels.
- Michael Dever and G. W. Hamilton. *A Comparison of Program Transformation Systems.* Compares positive supercompilation, distillation and higher-level supercompilation.

Program inversion is one of the favorite topics of metacomputation.

- Nikolai N. Nepejvoda. *Reversivity, Reversibility and Retractability.* Analyses three essentially different but usually mixed notions of program invertibility and presents an outline of their logics.
- Nikolay Shilov. *Inverting Dynamic Programming.* Considers Dynamic Programming as a case of inversion of a specific class of programs.

Towards practical metacomputation and applications

- Dimitur Krustev. *A Metacomputation Toolkit for a Subset of F# and Its Application to Software Testing: Towards Metacomputation for the Masses.* Presents an on-going experiment to develop a practical metacomputation toolkit for F# and also addresses a potential practical application — automatic generation of software test sets.
- Neil D. Jones. *Obfuscation by Partial Evaluation of Distorted Interpreters (Invited Talk).* Investigates the possibility of program obfuscation by program specialization.
- Gavin E. Mendel-Gleason and G. W. Hamilton. *Development of the Productive Forces.* Demonstrates how the productivity of a program, which is required in theorem provers such as Coq or Agda, can be derived by supercompiling the program and checking that the residual program satisfies a syntactic property referred to as guardedness condition.
- Alexei Adamovich. *Optimization of Imperative Functional Parallel Programs with Non-local Program Transformations.* Presents a method of nonlocal optimizing transformation of a typical parallel application solving massively parallel problem in the T-System being under development in Program Systems Institute.

Practical metaprogramming in Scala

– Eugene Burmako and Martin Odersky. *Scala Macros, a Technical Report.* Presents a minimalistic macro system for Scala, integrated with the static type system of the host language.
– Alexander A. Slesarenko. *Lightweight Polytypic Staging of DSLs in Scala.* Describes Lightweight Polytypic Staging, a new approach to the implementation of deep embedding of DSLs in Scala, and demonstrates its usefulness for efficient implementation of nested data parallelism as flat data parallelism.

The files of the papers and presentations of this and previous workshops on metacomputation 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/

June 2012

Andrei Klimov
Sergei Romanenko

Organization

Workshop Chair

Sergei Abramov, Ailamazyan Program Systems Institute of RAS, Russia

Program Committee Chairs

Andrei Klimov, Keldysh Institute of Applied Mathematics of RAS, Russia
Sergei Romanenko, Keldysh Institute of Applied Mathematics of RAS, Russia

Program Committee

Mikhail Bulyonkov, A. P. Ershov Institute of Informatics Systems of RAS, Russia
Robert Glück, University of Copenhagen, Denmark
Geoff Hamilton, Dublin City University, Republic of Ireland
Arkady Klimov, Institute for Design Problems in Microelectronics of RAS, Russia
Dimitur Krustev, IGE+XAO Balkan, Bulgaria
Alexei Lisitsa, Liverpool University, Great Britain
Gavin Mendel-Gleason, Dublin City University, Republic of Ireland
Neil Mitchell, Standard Charted, United Kingdom
Andrei Nemytykh, Ailamazyan Program Systems Institute of RAS, Russia
Johan Nordlander, Lulea University of Technology, Sweden
Peter Sestoft, IT University of Copenhagen, Denmark
Alexander Slesarenko, Keldysh Institute of Applied Mathematics of RAS, Russia
Morten Sørensen, Formalit, Denmark
Walid Taha, Computer and Electrical Engineering Halmstad University, Sweden

Invited Speaker

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

Sponsoring Organizations

Russian Academy of Sciences

Russian Foundation for Basic Research (grant № 12-07-06044-г)

Table of Contents

Optimization of Imperative Functional Parallel Programs with Non-local Program Transformations

Alexei Adamovich

Ailamazyan Program Systems Institute of Russian Academy of Sciences, RCMS, Pereslavl-Zalessky, Russia
lexa@botik.ru

Abstract. Functional programming is among paradigms used for the real-world parallel application development. PSI of RAS during a considerable period of time develops the T-System approach based on functional programming. This paper briefly describes the architecture of the ACCT compiler system targeted at performing transformations of the T-System programs. An algorithm for nonlocal optimizing transformation of a typical parallel application solving massively parallel problem in the T-System is presented. The author finally mentions several opportunities of other possible applications of the proposed architecture.

Keywords: functional programming, parallel application development, T-System, program transformation

1 Introduction

Advance in the field of parallel computations is one of the modern trends. The advance is due not only to the implementation of supercomputers with over 10 PFlops of performance. A substantial reason is that multicore processor architecture has become the dominant on the desktop PCs.

The current state of software tools for parallel applications development implies a coexistence of a number of paradigms. Specifically, the functional programming paradigm is being developed as a base for implementing the real-world parallel applications. In this paradigm, the possibility of automatic parallelization and dynamic load balancing is very attractive.

From the first half of 90s, the Aylamazyan Program Systems Institute of the Russian Academy of Sciences (PSI of RAS) develops an approach to parallel program development based on the functional paradigm which is called now the T-System [1]. Today, in PSI of RAS, there are made several different implementations of the T-System [2,3]. However, for all of the implementations one common disadvantage is the lack of tools for deep analysis and transformation of programs.

In the paper, the author presents general principles of the T-System. The paper also describes the architecture of the compiler version for the T-System

(with a codename ACCT) being implemented in PSI of RAS. ACCT allows to analyze programs given it at input and to execute their optimizing transformations. Then, the author outlines an algorithm for non-local transformations of a typical parallel application solving massively parallel problems in the T-System. The paper further gives several examples of other possible applications of the proposed architecture for increasing the efficiency of parallel applications.

2 Compiler Design

2.1 Basic Properties of the Input Language and the Model of Computation

For ACCT, the input language is an extended restriction of the C programming language (cT) [4]. Function bodies are written with a conventional imperative style. Function interaction is possible only within the framework of the functional programming paradigm, without the side effects: the information from outside can only be received via function arguments, and the transfer of information outwards is performed by sending function results (there may be a number of results).

When the T-function is called a T-process is created – which is a user-level thread. A new started T-process is potentially capable of being executed in parallel with the initial T-process. For enabling a parallel execution at the T-process launch, the variables that are receiving their values as a result of the functional call take special values – so called "non-ready values". A non-ready value is replaced with normal (ready) one after the T-process completed sending a corresponding result of the T-function call.

Non-ready values are located in special variables (outer variables). A non-ready value may easily participate in assignment of a value of one outer variable to another outer variable of the same type. If the T-process needs an outer variable value for executing a nontrivial operation (such as computation of the result of an arithmetic operation or transformation of an integral value into a float point value), the execution of such a T-process will be suspended until the outer variable takes a ready value.

It must be noted that T-function bodies may contain the calls of conventional functions (C functions), which requires to limit the effect of such a call by the T-process on the background of which the call is executed (there should be no side effects as far as other T-processes are concerned).

2.2 Compiler Architecture

The compiler consists of the following main components: front end, a set of transform passes, and back end.

Front end transforms the program module from an input language into an intermediate representation (IR). After the transformation is complete, the intermediate representation obtained as a result is stored in a separate file or special program library.

Each transform pass is able to transfer IR from the file or program library into RAM and somehow modify it. After that, a new version of IR is stored back on the external storage. Since all application modules are available to the transform pass, the performed transformations have a potential possibility to rely on the use of complete information about the application code as a whole.

The compiler back end reads IR from the file or program library and forms the resulting assembly (or C) code for further transformation into an executable program.

There also exists a compiler driver – a control program, which is needed for to call all the passes described above in the proper order.

A similar structure of compiling systems is used in a number of program transformation systems, such as SUIF [5], LLVM [6], OPS [7], etc. The ACCT implementation is heavily based on the C front end of the GCC compiler.

Hereinafter, we'll give an example of a non-local transformation of an application program. Such transformation may be implemented with the proposed program architecture.

3 Example of Program Transformation

3.1 Initial Problem

The proposed program transformation is suitable for the applications solving massively parallel problems. As an example of a program being transformed, we use a special modification of a standard iterative ray tracing algorithm. In case of ray tracing, a variable parameter in a massively parallel problem is a pair of coordinates of a point on the image plane.

The upper level of the modified algorithm implies a bisection of the rectangular part of the image plane containing the image. The division recursively proceeds until, after some step in the recursion, the resulting rectangles become sufficiently small. Thereafter, each of the resulting small rectangles is filled with image pixels by means of a standard tracing algorithm. Such small rectangular fragments are then assembled into a composite image.

Each small image fragment may be built independently of the others, which allows a parallel implementation of the problem. The recursive method of image fragmenting permits to bypass (e.g. executing the task on a cluster) the computation sequence which is typical for the so-called "task farm" paradigm and to avoid the appropriate performance penalties.

The implementation of the algorithm on cT may be represented as the following three functions:

1. The `render_scene` function (which is a C function) is destined for filling small rectangles with the RGB intensity values for each point of the fragment contained within such a rectangle.
2. The `render_scene_ut` T-function recursively bisects the rendering area. It also calls the `render_scene` function – in case the size limit of the area is reached (that is the base case).

3. `Tmain`. The launch of the T-process of the `TMain` function starts the execution of any application written in cT. `TMain` reads the scene description from the file and then launches the T-process with the first call to `render_scene_ut`. After that, `TMain` solves the problem of breadth-first traversal of the binary tree built by `render_scene_ut` and assembles a composite image from the fragments located inside the leaves of the tree, in parallel with the computation of individual fragments performed by `render_scene_ut`/`render_scene` calls.

In this paper, we'll consider the `render_scene_ut` T-function. The code of the function is as follows:

```
01 [void safe * sh]
02 render_scene_ut (double f_ulx,f_uly,f_stepx,f_stepy,
03                  int nx, ny, 04 void safe * sh_scene) {
05  void safe * utsh_res;
06
07  if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
08   int ny1, ny2;
09
10   ny1 = ny / 2;
11   ny2 = ny - ny1;
12   utsh_res = tnew (void safe * [2]);
13   utsh_res [0] =
14    render_scene_ut (f_ulx,f_uly,f_stepx,f_stepy,
15                     nx, ny1, sh_scene);
16   utsh_res [1] =
17    render_scene_ut (f_ulx,f_uly + f_stepy * ny1,
18                     f_stepx, f_stepy, nx, ny2,
19                     sh_scene);
20   sh <== utsh_res;
21  } else {
22   utsh_res =
23    tnew (char[sizeof (frag_dsc) +
24          CHAR_PER_POINT * nx * ny] outer);
25   render_scene
26    (f_ulx, f_uly, f_stepx, f_stepy, nx, ny,
27     ((char *) &(utsh_res.C)) + sizeof(frag_dsc));
28   sh <== utsh_res;
29  }
30 }
```

The function arguments are the parameters of the image fragments on the plane (the coordinates of upper left vertex of the rectangle, step size for each axis, the numbers of steps) and the scene description. As a result, the function returns a special-kind pointer called holder.

The line 7 of the function code above checks whether the bisection of the fragment must be continued. If bisection must be performed the resulting holder being returned (line 20) keeps (points to) a pair of similar holders (with initially non-ready values) returned in their turn by the recursive calls (lines 12 through 19). Otherwise, the function will return the holder (line 28) keeping the image fragment calculated by `render_scene` regular C call (lines 22 through 27).

Figure 1.a illustrates the sequence of the T-processes launched which starts when the `TMain` function calls the `render_scene_ut` T-function. As the picture indicates, the sufficient part of the T-processes recursively launches `render_scene_ut` and builds intermediate vertices of the binary tree fragments. The other part (building the leaves of the tree) computes the image fragments and returns them as the results. This means that almost a half of the T-processes are lightweight and the multiprocessor resources are underused as a consequence.

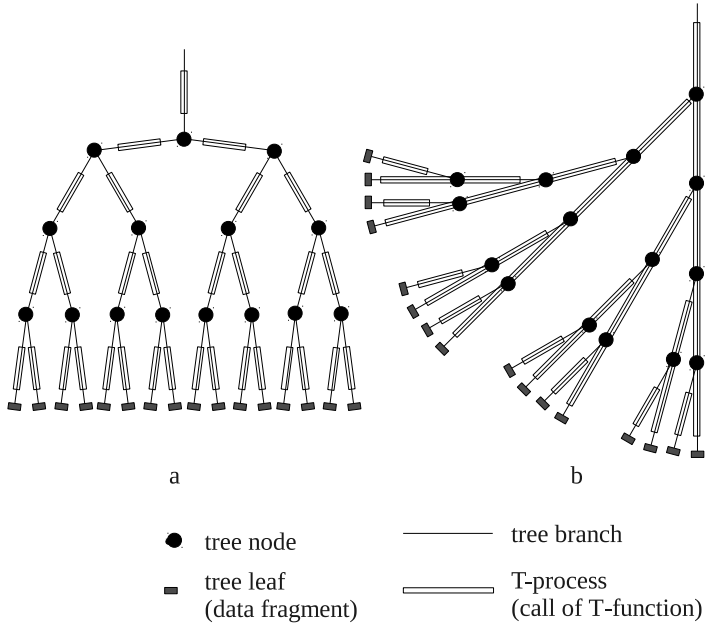


Fig. 1. Building a data fragments tree by parallel T-function calls: a – initial implementation scheme; b – scheme of implementation after modification.

Figure 1.b represents another scheme of building the tree of image fragments. On the 1.b scheme, each of the T-processes builds an image fragment located inside the tree leaf. Also one or more intermediate nodes of the tree may probably be built by the same T-process. This method of solving the problem permits to

avail the computational power of a multiprocessor efficiently since each of the T-processes becomes rather heavy computationally.

It is possible to obtain such parallel implementation of an application by changing the if-part of the conditional statement of the `render_scene_ut` function as follows:

```
06 ...
07  if (nx * ny > MIN_POINTS_PER_FRAG && ny >= 2) {
08   int ny1, ny2;
09'  void safe * utsh_w;
10'
11'  ny1 = ny / 2;
12'  ny2 = ny - ny1;
13'  utsh_res = tnew (void safe * [2]);
14'  utsh_w = utsh_res;
15'  for (;;) {
16'   utsh_w [0]
17'    = render_scene_ut
18'        (f_ulx, f_uly, f_stepx, f_stepy,
19'         nx, ny1, sh_scene);
20'   f_uly = f_uly + f_stepy * ny1;
21'   if (nx * ny2 <= MIN_POINTS_PER_FRAG
22'       || ny2 < 2)
23'    break;
24'   ny1 = ny2 / 2;
25'   ny2 = ny2 - ny1;
26'   utsh_w [1] = tnew (void safe * [2]);
27'   utsh_w = utsh_w [1];
28'  }
29'  utsh_w [1] =
30'   tnew (char[sizeof (frag_dsc) +
31'         CHAR_PER_POINT * nx * ny2] outer);
32'   render_scene
33'    (f_ulx, f_uly, f_stepx, f_stepy, nx, ny2,
34'     ((char *) (utsh_w[1].C))+sizeof(frag_dsc));
35'    sh <== utsh_res;
21  } else {
22 ...
```

The numbers of new (subject to changes) lines have a stroke. One can see that one (the second) of the recursive calls has been removed from the if-part of the conditional statement and the remaining (the first) call has been moved into a loop (lines 16' through 19'). This remaining call is responsible for launching the building of the left-upper branch (in terms of Fig. 1) in each intermediate node of the tree. All right-lower branches are computed by a single T-process during the loop execution. As the function exits the loop, it builds a tree leaf and returns the result (lines 29' through 35').

To find a way to generalize the mentioned transformation for solving an arbitrary massively parallel task is the core of the problem.

3.2 Solution: Sequence of Stages

Figure 2 illustrates a simplified scheme of the internal representation of the `compute_it_ut` function which implements the recursive part of the algorithm solving generalized massively parallel problems. Transformations consist in a partial replacement of recursion by iteration. Specifically, one of two recursive calls in the upper-left branch of the final conditional statement is to be replaced with iteration.

A transformation object is an inner representation of a given upper-left branch of a conditional statement. A transformation algorithm consists of three stages:

1. Substitution. The function body is subject to a special form of inlining – it is substituted into the second recursive call of the `compute_it_ut` function implementing the recursion step.
2. Looping. The looping stage is executed in several steps. The execution of all the three steps allows to considerably reduce the number of lightweight parallelism granules.
3. Final cleaning of variables and assignments.

Hereinafter, an overview of each step is presented.

Substitution. The second recursive function call – implementing the recursion step – is substituted with a copy of the function inner representation. Such substitution copies corresponding environments, including call arguments, and also assigns corresponding initial values to them.

As a result of the substitution stage, the inner representation of a recursive branch of the final conditional statement will contain three instead of two recursive calls to the `compute_it_ut` function. After further transformation at the looping stage, two of three recursive calls will be deleted but the remaining one will be executed in the loop body.

Looping. The two (of three) last recursive calls are completely eliminated at the looping stage. As a substitution to the eliminated recursive calls, the `compute_it` C function – loop structure and recursion base – is inserted into the recursive branch of the `compute_it_ut` function. The given procedure may be implemented as a following sequence of steps:

1. A working holder ("outer" pointer) with a unique name – indicated here as `utsh_w'` – is introduced into the `compute_it_ut` T function environment:

```
void safe * utsh_w';
```

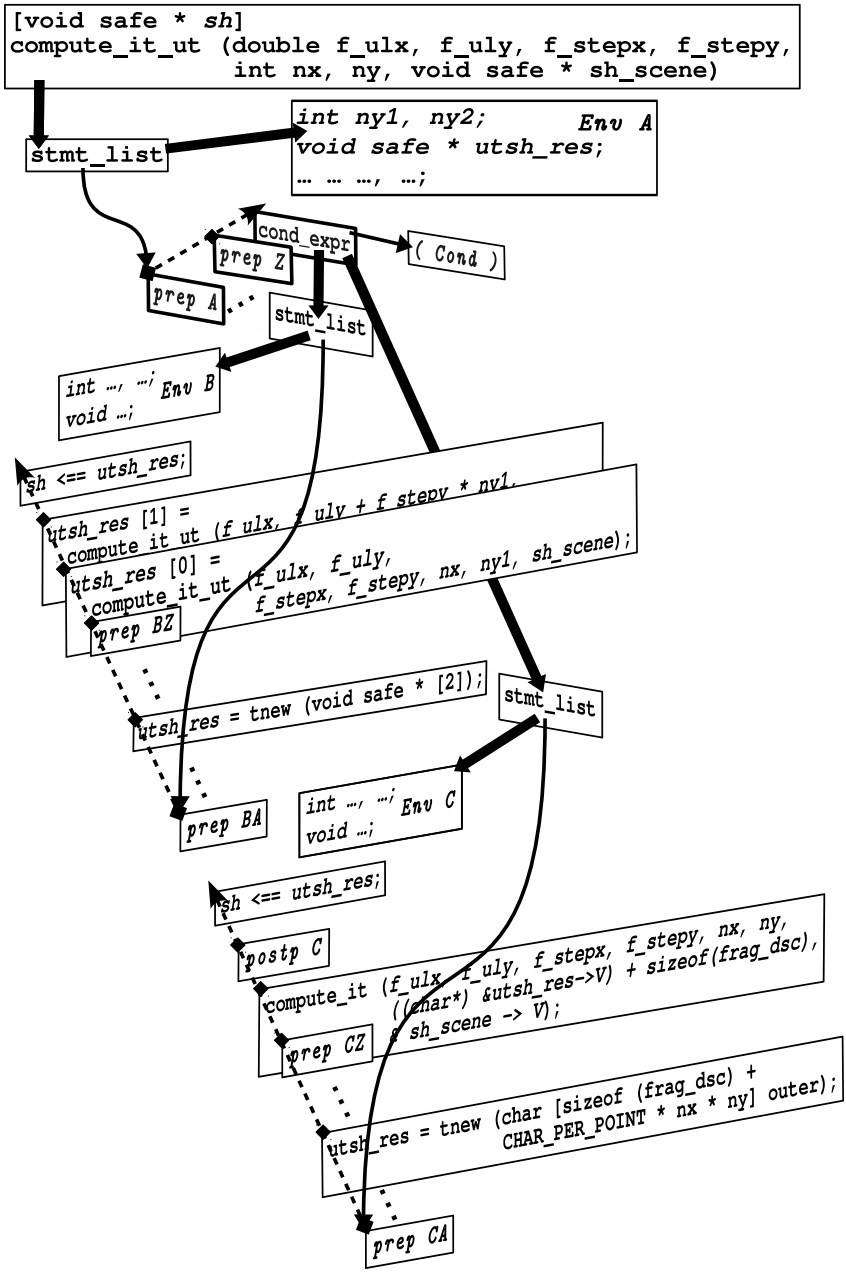


Fig. 2. Scheme of the internal representation of the function implementing the recursive part of the algorithm solving a generalized massively parallel problem

This holder will be used as leading and keep the current tree node – the node which is built in the loop at a current iteration step). The built root of a subtree returned as a result serves as an initial value of the leading pointer:

```
utsh_w' = utsh_res;
```

All subsequent occurrences of the `utsh_res` variable in the transformable code, except for the result sending final statement, should be substituted with occurrences of the `utsh_w'` variable.

2. The list of statements added at the substitution stage as a body of a substituted function is transformed into a loop statement body. The first recursive call of the `compute_it_ut` T-function is also moved into the loop body as the first statement.
3. After that, the nested (situated in the loop body) conditional statement is transformed. The non-recursive branch (the else-part containing the recursion base) is taken out of the loop body. The condition is reversed (the initial condition is denied). The break statement is placed into the conditional statement instead of the recursive branch. The recursive branch of the conditional statement is placed into the list of statements immediately after the conditional statement.

Two final recursive calls (added into the intermediate representation at the substitution step) are then deleted from the loop body. Thus, one initial call remains. A set of statements is added to the end of the loop, which – before the next operation starts – brings the variable environment to a state which is "equivalent" to the state it initially had after entering the called function and before performing the initial recursive call. In other words, the variables of the `A+B` (see Fig. 2) environment are reinitialized on the basis of variables of the `A'+B'` environment introduced during the substitution stage. The value transfer from the `A'+B'` environment to the `A+B` environment is made by reassignment; for example:

```
f_ulx = f_ulx';  f_uly = f_uly';
f_stepx = f_stepx';  f_stepy = f_stepy';
```

To complete the reinitialization, a new value is assign to the leading index:

```
utsh_w' = utsh_w' [1];
```

As Figure 3 illustrates, after the looping stage, the resulting scheme of the intermediate representation of the recursive branch is rather bulky. It should be noted that the scheme contains some excessive assignments and even some excessive variables that will be deleted at the following stage of transformation – at the cleaning stage.

Cleaning. As stated above, after mechanically implemented transformations, the intermediate representation of the recursive branch has a number of odd assignments and T-variables. For example, if we apply the above transformation steps to the `render_scene_ut` function, the result will contain the following definition:

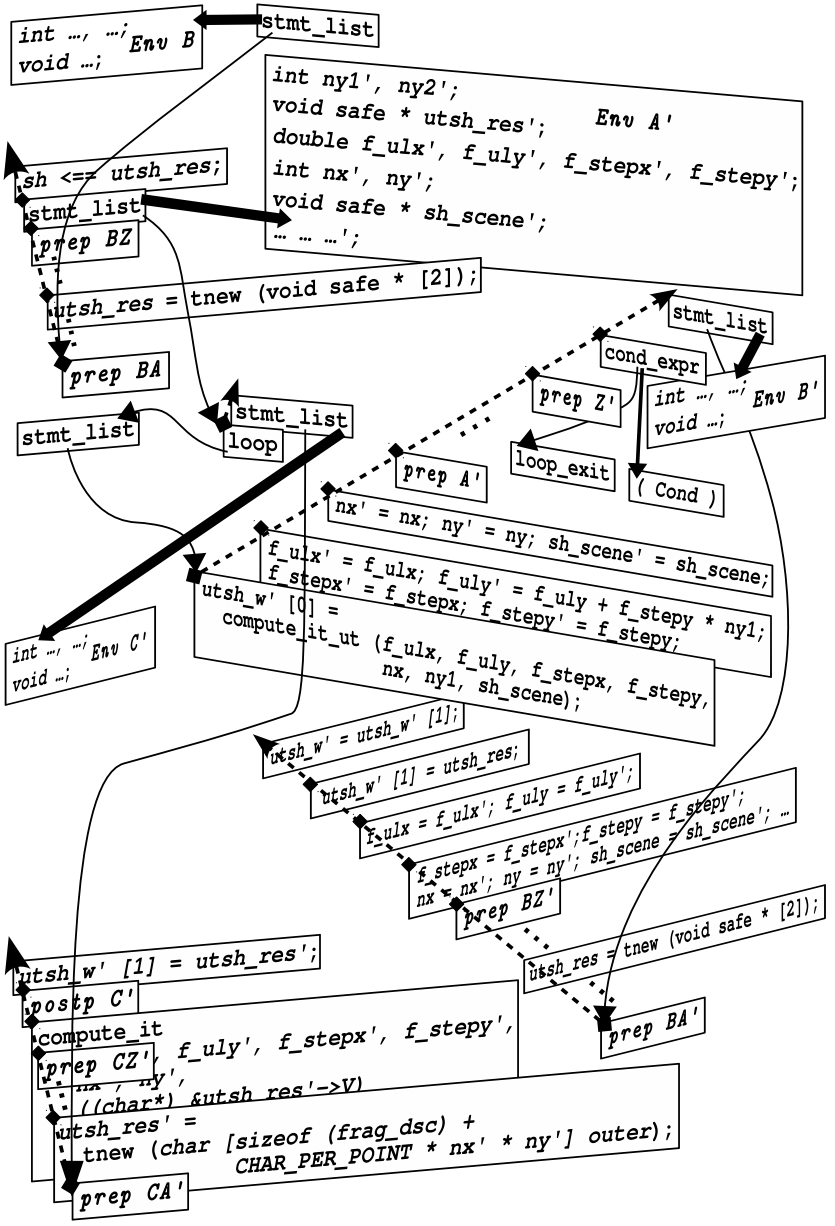


Fig. 3. Scheme of the intermediate representation of the recursive branch of the final conditional statement of the `compute_it_ut` function after the looping stage

```
void safe * sh_scene';
```

and a pair of assignments, such as

```
sh_scene' = sh_scene;
```

and

```
sh_scene = sh_scene';
```

with no other assignments to these variables are performed, which means thatthe assignments and the `sh_scene'` variable itself may be removed from the intermediate representation with substituting its other occurrences with the `sh_scene` variable references.

The optimizations being performed on the cleaning stage we have just mentioned are rather simple. However, they will not be automatically performed by the back-end since it merely converts outer variables and related actions into C correspondent data structures. After conversion, the information about semantics of outer variables will be lost.

4 Conclusion

Obviously, the transformation described above is not the only possible within the framework of the proposed ACCT architecture. The author hopes to make it possible to implement transformational passes based on more sophisticated techniques developed in the realm of functional programming (partial evaluations [8], supercompilation [9] etc.).

In addition, a set of transformational passes as tools for to support efficient implementation of a T-System runtime could be of value.

The author also expresses hope that the implementation of ACCT will permit to strengthen the position of the functional paradigm in the list of numerous modern programming paradigms being used in the area of parallel program development.

References

1. S. M. Abramov, A. I. Adamowitch, I. A. Nesterov, S. P. Pimenov, Y. V. Shevchuck. Autotransformation of evaluation network as a basis for automatic dynamic parallelizing, Proc. The 6th NATUG meeting, NATUG'1993 Spring Meeting "Transputer: Research and Application", May 10-11, 1993, IOS Press, Vancouver, Canada, pp. 333–344
2. S. M. Abramov, A. I. Adamovich, , and M. R. Kovalenko. T-System–Environment Supporting Automatic Dynamic Parallelization of Programs: An Example of the Implementation of an Image Rendering Algorithm Based on the Tracing Method, Programmirovanie, 1999, no. 2, pp. 100–107 (in Russian)

3. Sergey Abramov, Alexei Adamovich, Alexander Inyukhin, Alexander Moskovsky, Vladimir Roganov, Elena Shevchuk, Yuri Shevchuk, and Alexander Vodomerov. OpenTS: An Outline of Dynamic Parallelization Approach. Parallel Computing Technologies: 8th International Conference, PaCT 2005, Krasnoyarsk, Russia, September 5-9, 2005. Proceedings. Editors: Victor Malyshkin - Berlin etc. Springer, 2005. - Lecture Notes in Computer Science: Volume 3606, pp. 303–312
4. Alexey I. Adamovich. cT: An Imperative Language with Parallelizing Features Supporting the Computation Model "Autotransformation of the Evaluation Network". Proceedings of the 3rd International Conference on Parallel Computing Technologies (PaCT '95), St. Petersburg, Russia, September 1995, pp. 127–141
5. Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Monica S. Lam. Interprocedural parallelization analysis in SUIF. Transactions on Programming Languages and Systems (TOPLAS), Volume 27, Issue 4, July 2005, pp. 662–731
6. Chris Lattner, Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004, pp. 75–88
7. B. Steinberg, E. Alimova, A. Baglij, R. Morilev, Z. Nis, V. Petrenko, R. Steinberg. The System for Automated Program Testing. / Proceedings of IEEE East-West Design & Test Symposium (EWDTS'09). Moscow, Russia, September 18-21, 2009, pp. 218–220
8. Neil D. Jones, Carsten K. Gomard, Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice-Hall International Series in Computer Science, Prentice-Hall, 1993, 400 pages.
9. Valentin F. Turchin. Program transformation by supercompilation. Ganzinger H., Jones N.D. (ed.), Programs as Data Objects (Copenhagen, Denmark). Lecture Notes in Computer Science, Vol. 217, pp. 257–281, Springer-Verlag, 1986

Scala Macros, a Technical Report

Eugene Burmako and Martin Odersky

École Polytechnique Fédérale de Lausanne (EPFL)
first.last@epfl.ch

Abstract. Metaprogramming is a powerful technique of software development, which allows to automate program generation. Applications of metaprogramming range from improving expressiveness of a programming language via deep embedding of domain-specific languages to boosting performance of produced code by providing programmer with fine-grained control over compilation. In this report we introduce macros, facility that enables compile-time metaprogramming in the Scala programming language.

Keywords: Compile-time Metaprogramming, Macros, Multi-stage Programming, Language Virtualization

1 Introduction

As its name suggests, Scala (which stands for "scalable language" [1]) has been built from the ground up with extensibility in mind. Such features as abstract type members, explicit selftypes and modular mixin composition enable the programmer to compose programs as systems of reusable components [2].

The symbiosis of language features employed by Scala allows the code written in it to reach impressive levels of modularity [3], however there is still room for improvement. For example, the semantic gap between high-level abstractions and the runtime model of Java Virtual Machine brings performance issues that become apparent in high-performance scenarios [5]. Another example is state of the art in data access techniques. Recently established standards in this domain [4] cannot be readily expressed in Scala, which represents a disadvantage for enterprise software development.

Compile-time metaprogramming has been recognized as a valuable tool for enabling such programming techniques as: *language virtualization* (overloading/overriding semantics of the original programming language to enable deep embedding of DSLs) [6], *program reification* (providing programs with means to inspect their own code) [8,10], *self-optimization* (self-application of domain-specific optimizations based on program reification) [11,12], *algorithmic program construction* (generation of code that is tedious to write with the abstractions supported by a programming language) [7,8].

Our research introduces new concepts to Scala programming languages enabling metaprogramming techniques that address modern development challenges in an approachable and structured way [9].

2 Intuition

Here is a prototypical macro definition in our macro system:

```
def m(x: T): R = macro implRef
```

At first glance macro definitions are equivalent to normal function definitions, except for their body, which starts with the conditional keyword `macro` and is followed by a possibly qualified identifier that refers to a macro implementation method.

If, during type-checking, the compiler encounters an application of the macro `m(args)`, it will expand that application by invoking the corresponding macro implementation method, with the abstract-syntax trees of the argument expressions `args` as arguments. The result of the macro implementation is another abstract syntax tree, which will be inlined at the call site and will be type-checked in turn.

Example 1. The following code snippet declares a macro definition assert that references a macro implementation `Asserts.assertImpl`.

```
def assert(cond: Boolean, msg: Any) =
  macro Asserts.assertImpl
```

A call `assert(x < 10, "limit exceeded")` would then lead at compile time to an invocation:

```
assertImpl(c)(<[ x < 10 ]>, <[ "limit exceeded" ]>)
```

where `c` is a context argument that contains information collected by the compiler at the call site (receiver of the macro invocation, symbol tables for enclosing lexical scopes, etc.), and the other two arguments are abstract syntax trees representing the two expressions `x < 10` and `"limit exceeded"`.

In this document <[`expr`]> denotes the abstract syntax tree that represents the expression `expr`, but this notation has no counterpart in our extension of the Scala language. The canonical way to construct abstract syntax trees is to use the types in the compiler library, which for the two expressions above looks like this:

```
Apply(
  Select(Ident(newTermName("x")), newTermName("$less"),
  List(Literal(Constant(10))))

Literal(Constant("limit exceeded"))
```

The core of our macro system is described in sections 3 through 6 and is inspired by the notions from LISP [13], Scheme [14] and Nemerle [8]. Sections 7 through 9 describe a peculiar feature of Scala macros that makes use of staging to bootstrap macros into a hygienic and quasiquoting metaprogramming system. Subsequent sections conclude the report.

3 Baseline

Let us examine a possible implementation of the assert macro mentioned in *Example 1* to explore the foundations of Scala macros:

```
object Asserts {
  def assertionsEnabled = ...
  def raise(msg: Any) = throw new AssertionError(msg)
  def assertImpl(c: Context)
    (cond: c.Expr[Boolean], msg: c.Expr[Any])
    : c.Expr[Unit] =
  if (assertionsEnabled)
    <[ if (!cond) raise(msg) ]>
  else
    <[ () ]>
}
```

As the listing shows, a macro implementation takes several parameter lists. First comes a single parameter, of type `Context`. This is followed by a list of parameters that have the same names as the macro definition parameters. But where the original macro parameter has type `T`, a macro implementation parameter has type `c.Expr[T]`. `Expr[T]` is a type defined in `Context` that wraps an abstract syntax tree of type `T`. The result type of the `assertImpl` macro implementation is again a wrapped tree, of type `c.Expr[Unit]`.

Parameters of a macro implementation are dependently typed, being a part of a dependent method type [15]. Such type annotations statically ensure that artifacts passed into a macro belong to the context that services a macro expansion. This type-checking facility is important from a practical standpoint, as it prevents accidental mix-up of compilation stages. For example, without dependent typing it would be possible to inadvertently refer to runtime trees and types (obtained from a reflection context) in a compile-time macro (that uses the compiler context).

The macro being discussed is static, in a sense that it has a statically known receiver (such receivers are called "objects" in Scala parlance). It is possible, however, to define instance macros and use them in a prefix fashion, analogously to instance methods, e.g. `receiver.a_macro(args)`. In that case, abstract syntax tree corresponding to `receiver` is passed to the macro implementation in `Context`.

4 Expression Trees

An expression tree of type `Expr[T]` encapsulates an abstract syntax tree of type `T` together with its type. Heres the definition of `Expr` as a member of the compiler library exposed to macro implementations:

```
case class Expr[T: TypeTag](tree: Tree) {
  def eval: T = ...
  lazy val value: T = eval
}
```

Implicit in the contract for `Expr` is that the type of the reified tree conforms to the type parameter `T` (which is also reified by the virtue of the `TypeTag` context bound, as described in subsequent sections). `Expr` values are typically created by the compiler, which makes sure that this contract is kept.

Note that the method `eval` which when called on a value of type `Expr[T]` will yield a result of type `T`. The `eval` method and the `value` value play a special role in tree splicing as described in subsequent sections.

5 Polymorphic Macros

Macro definitions and macro implementations may both be polymorphic.

Type parameters in an implementation may come with `TypeTag` context bounds [16]. In that case the corresponding `TypeTags` describing the actual type arguments instantiated at the application site will be passed along when the macro is expanded.

Example 2. The code below declares a polymorphic macro definition `Queryable.map` that references a polymorphic macro implementation `QImpl.map`:

```
class Queryable[T] {
  def map[U](p: T => U): Queryable[U] = macro QImpl.map[T, U]
}

object QImpl {
  def map[T: c.TypeTag, U: c.TypeTag]
         (c: Context)
         (p: c.Expr[T => U])
         : c.Expr[Queryable[U]] = ...
}
```

As outlined in [16], context bounds provide a concise notation for declaring implicit parameter sections that captures suitable type class instances from lexical scope. For example, method `QImpl.map` is desugared into the following form:

```
object QImpl {
  def map[T, U]
         (c: Context)
         (p: c.Expr[T => U])
         (implicit evidence$1: c.TypeTag[T],
          implicit evidence$2: c.TypeTag[U])
         : c.Expr[Queryable[U]] = ...
}
```

Now consider a value `q` of type `Queryable[String]` and the following macro call (the explicit type argument `[Int]` can be omitted, in which case it will be inferred by the compiler):

```
q.map[Int](s => s.length)
```

The call is expanded to the following macro invocation:

```
QImpl.map(c)(<[ s => s.length ]>)
  (implicitly[c.TypeTag[String]], implicitly[c.TypeTag[Int]])
```

The `implicitly` function is used to summon suitable (i.e. marked as implicit and having a conformant type signature) type tags from the lexical scope of the call site. Shortly put, implicit search starts with the innermost enclosing scope and proceeds from the inside out (details about the implicit resolution algorithm may be found in [17]).

Of course, macro runtime does not expect the programmer to know about macro contexts, to create the type tags manually and to put them into local variables visible from macro call sites. In a common case when type tags are not declared explicitly, implicit search will fall back to the outermost scope, declared in the standard library. This outermost scope hosts implicit macros that are capable of materializing type tags for arbitrary types.

6 Type Tags

A value of type `TypeTag[T]` encapsulates a representation of type `T`. A `TypeTag` value simply wraps a Scala type, while a `ConcreteTypeTag` value is a type tag that is guaranteed not to contain any references to type parameters or abstract types.

```
case class TypeTag[T](tpe: Type) { ... }
class ConcreteTypeTag[T](tpe: Type) extends TypeTag[T](tpe)
```

Implicit in the contract for all `Tag` classes is that the reified type represents the type parameter `T`. Tags are typically created by the compiler, which makes sure that this contract is kept. The creation rules are as follows:

1) If an implicit value of type `TypeTag[T]` is required, the compiler will summon it from the enclosing lexical scope or make one up on demand using the implicit search algorithm described in the previous section.

2) The implicitly created value contains a value of type `Type` that is a reified representation of `T`. In that value, any occurrences of type parameters or abstract types `U` which come themselves with a `TypeTag` are represented by that `TypeTag`. This is called type splicing.

3) If an implicit value of type `ConcreteTypeTag[T]` is required, the compiler will make one up on demand following the same procedure as for `TypeTags`. However, if the resulting type still contains references to type parameters or abstract types, a static error results.

As an example that illustrates type splicing, consider the following function:

```
def f[T: TypeTag, U] = {
  type L = T => U
  implicitly[TypeTag[L]]
}
```

Then a call of `f[String, Int]` will yield a result of the form:

```
TypeTag(<[ String => U ]>)
```

Note that `T` has been replaced by `String`, because it comes with a `TypeTag` in `f`, whereas `U` was left as a type parameter.

Type splicing plays an important role in the design of the metaprogramming system, because Scala uses the erase-types model to compile polymorphic types [18]. In this model, when the program is compiled down to executable form, type arguments of the invocations are removed, and type parameters are replaced by their upper bounds. With implicit parameters it becomes possible to capture type arguments during the compile-time to retain them at runtime [3], and type splicing scales this technique to complex types.

7 Quasiquoting and Hygiene

The macro scheme described so far has the advantage that it is minimal, but also suffers from two inconveniences: tree construction is cumbersome and hygiene is not guaranteed. Consider a fragment of the body of `assertImpl` in *Example 1*:

```
<[ if (!cond) raise(msg) ]>
```

To actually produce the abstract syntax tree representing that expression one might write something like that:

```
c.Expr(
If(Select(cond, newTermName("unary_$bang")),
  Apply(Ident(newTermName("raise")), List(msg)),
  Literal(Constant(()))))
```

Cumbersome enough as this is, it is also wrong. The tree produced from a macro will be inlined and type-checked at the macro call site. But that means that the identifier `raise` will be type-checked at a point where it is most likely not visible, or in the worst case they might refer to something else. In the macro literature, this insensitivity to bindings is called non-hygienic [19,8].

In the case of `assertImpl`, the problems can be avoided by generating instead of an identifier a fully qualified selection

```
Select(Ident(newTermName("Asserts")), newTermName("raise"))
```

(to be completely sure, one would need to select the full path starting with the `_root_` package). But that makes the tree construction even more cumbersome and is very fragile because it is easily forgotten.

However, it turns out that macros themselves can be used to solve both these problems. A corner-stone of the technique is a macro called `reify` that produces its tree one stage later.

8 Reify

The reify macro plays a crucial role in the proposed macro system. Its definition as a member of `Context` is:

```
def reify[T](expr: T): Expr[T] = macro ...
```

Reify accepts a single parameter `expr`, which can be any well-typed Scala expression, and creates a tree that, when compiled and evaluated, will recreate the original tree `expr`. So `reify` is like time-travel: trees get re-constituted at a later stage. If `reify` is called from normal compiled code, its effect is that the abstract syntax tree passed to it will be recreated at run time. Consequently, if `reify` is called from a macro implementation, its effect is that the abstract syntax tree passed to it will be recreated at macro-expansion time (which corresponds to run time for macros). This gives a convenient way to create syntax trees from Scala code: pass the Scala code to `reify`, and the result will be a syntax tree that represents that very same code.

Moreover, `reify` packages the result expression tree with the types and values of all free references that occur in it. This means in effect that all free references in the result are already resolved, so that re-typechecking the tree is insensitive to its environment. All identifiers referred to from an expression passed to `reify` are bound at the definition site, and not re-bound at the call site. As a consequence, macros that generate trees only by the means of passing expressions to `reify` are hygienic.

So in that sense, Scala macros are self-cleaning. Their basic form is minimal and unhygienic, but that simple form is expressive enough to formulate a `reify` macro, which can be used in turn to make tree construction in macros concise and hygienic.

Example 3: Here is an implementation of the assert macro using `reify`.

```
object Asserts {
  def assertionsEnabled = ...
  def raise(msg: Any) = throw new AssertionError(msg)
  def assertImpl(c: Context)
    (cond: c.Expr[Boolean], msg: c.Expr[Any])
    : c.Expr[Unit] =

    if (assertionsEnabled)
      c.reify(if (!cond.eval) raise(msg.eval))
    else
      c.reify(())
}
```

Note the close correspondence with the meta-notation of *Example 1*.

9 Splicing

`Reify` and `eval` are inverses of each other. `Reify` takes an expression and produces a tree that, when evaluated with `eval`, yields the same result as the original expression. This is also expressed by their types. `reify` goes from `T` to `Expr[T]`, and `eval` goes from `Expr[T]` back to `T`.

The `reify` macro takes advantage of this relationship by short-circuiting embedded calls to `eval` during the process that we call tree splicing (compare this with type splicing described above):

```
reify(expr.eval) translates to expr
```

This principle is seen in action in *Example 3*. There, the contents of the parameters `cond` and `msg` are spliced into the body of the `reify`.

Along with `eval`, `value` also gets special treatment:

```
reify(expr.value) also translates to expr
```

Similar to `eval`, the `value` value also makes `reify` splice its tree into the result. The difference appears when the same expression gets spliced into multiple places inside the same reify block. With `eval`, `reify` will always insert a copy of the corresponding tree (potentially duplicating side-effects), whereas `value` will splice itself into a temporary variable that will be referred by its usages.

The notion of splicing also manifests itself when `reify` refers to a type that has a `TypeTag` associated with it. In that case instead of reproducing the types internal structure as usual, `reify` inserts a reference to the type tag into its result.

```
reify(expr: T) translates to expr typed as TypeTag[T].tpe
```

Tagging a type can be done either automatically, by writing a `TypeTag` context bound on a type parameter of a macro implementation, or manually, by introducing an implicit `TypeTag` value into the scope visible by `reify`.

Note the close resemblance of type splicing in `reify` and type splicing during `TypeTag` generation. In fact, here we are talking about the same algorithm. When producing a `TypeTag` for a type, corresponding implicit macros call `reify` (which, in turn, calls `TypeTag` generators to resolve potential splices using the implicit search algorithm).

10 Related Work

The history of compile-time metaprogramming dates back to the times of LISP [13], which was introduced in 1950s. Since then a fair amount of languages: statically typed [7] and dynamically typed [20], minimalistic [14] and having rich syntax [8] - have adopted macros. Our research builds on this notion of compile-time program transformation.

Hygiene is an important idea brought up in Scheme. The problem of inadvertent name clashes between the application code and generated macro expansions has been well-known in the Lisp community. Kohlbecker et al. [19] have solved this problem by embedding the knowledge of declaration sites into symbols that represent values and declarations in the program, making macro expansions hygienic.

We acknowledge this problem, but our means of achieving hygiene do not require changes to the type-checking algorithm. By making use of `reify`, a staging macro, we statically ensure that cross-stage bindings do not occur. Similar approach has been used in MacroML [21], which implements a macro system in a staged language MetaML [22]. Our arrival at this conflux of ideas happened the other way around - we built a staged system with macros.

As a language with syntax, Scala has to work hard to achieve homoiconicity. The incovenience of manipulating abstract syntax trees in their raw form is a well-known problem, and it affects rich languages to a greater extent than it affects minimalistic ones. Traditional solution to this problem is a quasiquoting DSL that lets the programmer encode ASTs in a WYSIWYG manner [26,8,27].

Our answer to this challenge is the same staging macro `reify` that we use to achieve hygiene. Code passed to `reify` becomes an AST one stage later, which provides a quasiquoting facility without the need to introduce a special domain-specific language.

Finally, as a language virtualization platform, Scala macros are conceptually close to Scala-Virtualized [23] which virtualizes base language constructs (e.g. control flow) and even data structures [24]. However, our approach to virtualization is different. Scala macros expose general-purpose Scala trees and types and provide low-level manipulation facilities, whereas Scala-Virtualized is good for embedded DSLs, in particular when the DSL expression trees do not exactly correspond to Scala trees [25].

11 Conclusions

We have presented a minimalistic macro system for Scala, a language with rich syntax and static types. This macro system builds up a metaprogramming facility on a single concept - compile-time AST transformer functions.

Other metaprogramming facilities usually include additional concepts of quasiquoting and hygiene to make themselves suitable for practical use. We have shown, however, that it is possible to implement both on top of our minimalistic core.

Acknowledgements

The authors would like to thank Vladislav Chistyakov, Jan Christopher Vogt, Stefan Zeiger, Adriaan Moors and the entire Scala community for insightful discussions and helpful comments.

References

1. Odersky, M., Spoon L., and Venners B., *Programming in Scala, Second Edition.* Artima Press, 2010.
2. Odersky, M., and Zenger M., *Scalable Component Abstractions.* ACM Sigplan Notices, 2005.
3. Odersky, M., and Moors, A., *Fighting Bit Rot with Types (Experience Report: Scala Collections).* Theoretical Computer Science, 2009.
4. Box, D., and Hejlsberg, A., *LINQ: .NET Language-Integrated Query*, Retrieved from http://msdn.microsoft.com/en-us/library/bb308959.aspx, 2007.
5. Dragos I., *Optimizing Higher-Order Functions in Scala*, Third International Workshop on Implementation Compilation Optimization of ObjectOriented Languages Programs and Systems, 2008.

6. McCool, M. D., Qin, Z., and Popa, T. S., *Shader metaprogramming*, Proceedings of the ACM SIGGRAPHEUROGRAPHICS conference on Graphics hardware, 2002.
7. Sheard, T., and Peyton Jones, S., *Template Meta-programming for Haskell*, Haskell Workshop, 2002.
8. Skalski K., *Syntax-extending and type-reflecting macros in an object-oriented language*, Master Thesis, 2005.
9. Scala Macros, *Use cases*, Retrieved from http://scalamacros.org/usecases.html, 2012.
10. Attardi, G., and Cisternino, A., *Reflection support by means of template metaprogramming*, Time, 2001.
11. Seefried, S., Chakravarty, M., and Keller, G., *Optimising Embedded DSLs using Template Haskell*. Generative Programming and Component Engineering, 2004.
12. Cross, J., and Schmidt, D., *Meta-Programming Techniques for Distributed Real-time and Embedded Systems*, 7th IEEE Workshop on Object-oriented Real-time Dependable Systems, 2002.
13. Steele, G., *Common LISP. The Language. Second Edition*, Digital Press, 1990.
14. *The Revised [6] Report on the Algorithmic Language Scheme*, Journal of Functional Programming, volume 19, issue S1, 2010.
15. Odersky, M., Cremet, V., Rckl, C., and Zenger M., *A Nominal Theory of Objects with Dependent Types*, 17th European Conference on Object-Oriented Programming, 2003.
16. Oliveira, B., Moors, A., and Odersky, M., *Type classes as objects and implicits*, 25th Conference on Object-Oriented Programming, Systems, Languages & Applications, 2010.
17. Odersky, M., *The Scala Language Specification, Version 2.9*, 2011.
18. Schinz, M., *Compiling Scala for the Java Virtual Machine*, PhD thesis, 2005.
19. Kohlbecker, E., Friedman, D., Felleisen, M., and Duba, B., *Hygienic macro expansion*, Symposium on LISP and Functional Programming, 1986.
20. Rahien, A., *DSLs in Boo: Domain-Specific Languages in .NET*, Manning Publications Co., 2010.
21. Ganz, S., Sabry, A., and Taha, W., *Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML*, International Conference on Functional Programming, 2001.
22. Taha, W., and Sheard, T., *MetaML: Multi-Stage Programming with Explicit Annotations*, 1999.
23. Moors, A., Rompf, T., Haller, P., and Odersky, M., *Scala-Virtualized*, Partial Evaluation and Program Manipulation, 2012.
24. Slesarenko A, *Lightweight Polytypic Staging: a new approach to Nested Data Parallelism in Scala*, Scala Days, 2012.
25. Rompf, T., and Odersky, M., *Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs*, 2010.
26. Bawden, A., *Quasiquotation in Lisp*, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and SemanticsBased Program Manipulation, 1999.
27. Mainland, G., *Why it's Nice to be Quoted: Quasiquoting for Haskell*, Applied Sciences, 2007.

A Comparison of Program Transformation Systems

Michael Dever and G.W. Hamilton

Dublin City University
{mdever, hamilton}@computing.dcu.ie

Abstract. Program transformation is a popular technique for attempting to improve the efficiency of a program. At a high level, program transformation is the process of taking an input program and transforming it into an improved version of the original, bearing the same constraints as the original, e.g. termination constraints. In this paper, we focus on three fold/unfold [3] based transformation systems, *positive supercompilation* [26,25,21,2,12] and *distillation* [8,9,10,11] and *HOSC* [19,18]. We focus on derivatives of both that use labelled transition systems [12,11] and we present these systems, their underlying theory, and implementations. Based upon these implementations we will present an analysis of how they compare to each other, and another transformation system, HOSC[19], when applied to a sample of real-world programs.

1 Introduction

Program transformation describes the process of taking an input program and transforming it via various methodologies, discussed below, to a semantically equivalent program [23] that is bounded by the same constraints as the original program. The goal of such a transformation is to enhance and improve the original program, whether the improvements be scalability, efficiency, concurrency or other measures of improvement. This goal exists as programming, in any language, can be an arduous task, with many aspects that have to be taken in to consideration by the developer.

As skill, knowledge and experience can vary greatly from programmer to programmer, program transformation has the potential to prove an immense aid to developers. If a developer can write a version of software, and have it improved for them by another program (the transformer), this will result in better software coming from that developer. There are, however, downsides to applications of program transformation in the field. Program transformers don't always produce intuitive, comprehendible output; if they did, and the results were intuitive, there would not be a need for the transformer.

There exist many different techniques for the application of program transformation techniques to functional programs; Burstall & Darlington's *fold/unfold* algorithms [3], which are the cornerstones of the other transformations detailed here; Pettorossi and Proietti's *transformation rules (fold/unfold based)*

[23], which further the techniques of Burstall & Darlington; Wadler's *deforestation* [31] and its derivatives [5,7], Turchin's *supercompilation* [28], and its derivatives in the form of *positive supercompilation* [26,25,21,2], *two-level supercompilation*[19,18], Hamilton's *distillation* [8,9,10,11] and many others.

While we focus on program transformation applied to functional languages in this paper, it is worth noting that program transformation is applicable to other types of language, such as logic languages [20], and sequential languages [17]. There are a number of reasons why we focus on functional languages, but the most important reasons are that functional languages are easier to analyze, reason about, and to manipulate using program transformation techniques. The lack of side-effects in pure functional languages is a major benefit, as these do not have to be taken into consideration during the transformation process.

A key note to be made about functional languages is that due to their nature, a lot of functions use intermediate data structures to generate results. As an example, the naive definition of *reverse* below relies heavily upon using intermediate lists, in its call to *append*. Another key feature of some functional languages is the ability to use lazy evaluation, where results are evaluated as they are needed. Even within this context, the use of intermediate structures can be a hindrance, as each allocation requires space and time for allocation etc. [31], and the transformations examined here are aimed at reducing/eliminating usage of intermediate data. To show these reductions, we will present our implementations of the transformation systems, and an analysis of how they compare to each other.

$$
\begin{array}{lll}
reverse = \lambda xs.\textbf{case} & xs \textbf{ of} & \\
 & [] & \rightarrow [] \\
 & (x : xs) & \rightarrow append\ (reverse\ xs)\ [x]
\end{array}
$$

$$
\begin{array}{lll}
append = \lambda xs\ ys.\textbf{case} & xs \textbf{ of} & \\
 & [] & \rightarrow ys \\
 & (x : xs) & \rightarrow (x : append\ xs\ ys)
\end{array}
$$

In this paper, we focus on fold/unfold based transformations using *labelled transition systems* [12,11], that depict a programs run-time behavior, and use *weak bisimulation* to prove correctness. The remainder of this paper is structured as follows: in Section 2 we define the higher-order language to be used, and some constraints on it. In Section 3, we define a labelled transition system, and weak bisimilarity. In Section 4 we define both supercompilation and distillation using a labelled transition system, and in Section 5 we present the results of using these to optimize a sample of programs.

2 Language

The simple higher-order language to be used throughout this paper is shown below:

Within this language, a program consists of an expression to be evaluated, e_0, and a set of function definitions, $\Delta = f_1 = e_1 \ldots f_k = e_k$. Constructors must be

$$
\begin{array}{lll}
prog ::= e_0 \textbf{ where } f_1 = e_1 \ldots f_k = e_k & & \text{Program} \\
\\
e \quad ::= x & & \text{Variable} \\
\quad | \ c\ e_1 \ldots e_k & & \text{Constructor} \\
\quad | \ f & & \text{Function} \\
\quad | \ \lambda x.e & & \text{Lambda Abstraction} \\
\quad | \ e_0\ e_1 & & \text{Application} \\
\quad | \ \textbf{case } e_0 \textit{ of } p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k & & \text{Case Expression} \\
\\
p \quad ::= c\ x_1 \ldots x_k & & \text{Pattern}
\end{array}
$$

Fig. 1. Language Definition

of a fixed arity, and within $c\ e_1 \ldots e_k$, k must be equal to constructor c's arity. *Bound* variables are those introduced by λ-abstraction or **case** patterns, and all other variables are *free*. Two expressions, e_1 and e_2, are equivalent, $e_1 \equiv e_2$, if the only difference between the two is the naming of their bound variables. Case expressions may only have non-nested patterns, and if nested patterns are present, they must be transformed into equivalent non-nested versions [1,30].

The language shown uses a standard call-by-name operational semantics, in which there exists an evaluation relation $\Downarrow$ between closed expressions and *values* (expressions in weak head normal form [14]). The one-step reduction relation, $\overset{r}{\rightsquigarrow}$, is shown in Figure 2, and defines three reductions, f, c and β, where f represents an unfolding of function f, c represents a constructor elimination of constructor c and β represents β-substitution.

$$
\frac{(f = e) \in \Delta}{f \overset{f}{\rightsquigarrow} e} \qquad ((\lambda x.e_0)\ e_1) \overset{\beta}{\rightsquigarrow} (e_0\{x \mapsto e_1\}) \qquad \frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(e_0\ e_1) \overset{r}{\rightsquigarrow} (e_0'\ e_1)}
$$

$$
\frac{p_i = c\ x_1 \ldots x_n}{(\textbf{case } (c\ e_1 \ldots e_n) \textbf{ of } p_1 : e_1' \mid \ldots \mid p_k : e_k') \overset{c}{\rightsquigarrow} (e_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\})}
$$

$$
\frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(\textbf{case } e_0 \textbf{ of } p_1 : e_1 \mid \ldots p_k : e_k) \overset{r}{\rightsquigarrow} (\textbf{case } e_0' \textbf{ of } p_1 : e_1 \mid \ldots p_k : e_k)}
$$

Fig. 2. One-Step Reduction Relation

$e \overset{r}{\rightsquigarrow}$ denotes the reduction of an expression, e, by rule r, $e \Uparrow$ denotes e diverging, and $e \Downarrow$ denotes e converging. $e \Downarrow v$ can be used to denote e evaluating to the value v. These notations are defined in below, where $\overset{r}{\rightsquigarrow}^*$ denotes the reflexive transitive closure of $\overset{r}{\rightsquigarrow}$.

$e \overset{r}{\rightsquigarrow}$, iff $\exists e'.e \overset{r}{\rightsquigarrow} e'$ $\quad$ $e \Downarrow$, iff $\exists v.e \Downarrow v$
$e \Downarrow v$, iff $e \overset{r}{\rightsquigarrow}^* v \wedge \neg(v \overset{r}{\rightsquigarrow})$ $\quad$ $e \Uparrow$, iff $\forall e'.e \overset{r}{\rightsquigarrow}^* e' \Rightarrow e' \overset{r}{\rightsquigarrow}$

Definition (Substitution) If e is an expression, and there exists a substitution, $\theta = \{x_1 \to e_1, \ldots x_n \to e_n\}$, then $e\theta = e\{x_1 \to e_1, \ldots x_n \to e_n\}$ denotes the simultaneous substitution of e_i for the variable x_i in e, while ensuring name capture cannot happen. □

Definition (Renaming) If there exists a bijective mapping, σ, such that $\sigma = \{x_1 \to x'_1, \ldots, x_n \to x'_n\}$, and there exists an expression e, then $e\{x_1 \to x'_1, \ldots, x_n \to x'_n\}$ denotes the simultaneous substitution of the variable x_i for x'_i in e. □

Definition (Context) A context, $\mathcal{C}$, is an expression that contains a hole $[]$, where one sub-expression should be, and $\mathcal{C}[e]$ denotes the replacing the hole in $\mathcal{C}$ with the sub-expression e. □

Definition (Evaluation Context, Redex and Observable) Evaluation contexts, $\mathcal{E}$, redexes, $\mathcal{R}$, and observables, $\mathcal{O}$ are as defined below.

$$
\begin{array}{rl}
\mathcal{E} ::= & [] \\
| & \mathcal{E}\ e \\
| & \textbf{case}\ \mathcal{E}\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k \\
\\
\mathcal{R} ::= & f \\
| & (\lambda x.e_0)\ e_1 \\
| & \textbf{case}\ (x\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \ldots \mid p_k \Rightarrow e'_k \\
| & \textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 \Rightarrow e'_1 \mid \ldots \mid p_k \Rightarrow e'_k \\
\\
\mathcal{O} ::= & x\ e_1 \ldots e_n \\
| & c\ e_1 \ldots e_n \\
| & \lambda x.e
\end{array}
$$

□

Definition (Observational Equivalence) Observational equivalence, $\simeq$, equates two expressions if and only if they exhibit the same termination behavior in all closing contexts, i.e. $e_1 \simeq e_2$ iff $\forall \mathcal{C}.\mathcal{C}[e_1] \Downarrow$ iff $\mathcal{C}[e_2] \Downarrow$. □

3 Labelled Transition Systems

As per Gordon [6], Hamilton and Jones [12,11], define and extend a labelled transition system, that depicts immediate observations that can be made on expressions to determine their equivalence. Their extension is to allow free variables in both expressions and actions, and a knock on effect of this is that observational equivalence will now require that both free and bound variables in actions match.

Definition (Labelled Transition System) A *driven* LTS associated with the program e_0 is represented by $t = (\mathcal{E}, e_0, \rightarrow, Act)$ where:

- $\mathcal{E}$ represents the set of states of the LTS. Each state can be either an expression or the *end-of-action* state **0**.
- t contains as root the expression e_0, denoted $root(t)$.
- $\rightarrow \subseteq \mathcal{E}\ X\ Act\ X\ \mathcal{E}$ is a *transition relation* relating pairs of states by actions according to the driving rules.
- If $e \in \mathcal{E}$ and $e \xrightarrow{\alpha} e'$ then $e' \in \mathcal{E}$.
- *Act* is a set of actions, α, each of which can either be *silent* or *non-silent*. Non-silent actions are one of: a variable, a constructor, the i^{th} argument of an application, a λ-abstraction over a variable x, **case** selector, a case branch pattern or a **let** abstraction. Silent actions are one of: τ_f, the unfolding of a function f, τ_c, the elimination of a constructor c or τ_β, β-substitution.

□

λ-abstractions, **case** pattern variables and **let** variables that are within the actions of an LTS, t, are bound, denoted by $bv(t)$, while all other variables are free, denoted by $fv(t)$. **let** transitions do not appear in a driven LTS, and are only introduced later on due to generalization, via **let** transitions. The authors note that the LTS notation allows for identifying program behavior just by looking at the labels on the transitions, and that transitions from constructors or variables lead to the end-of-action state, **0**. In addition to the above notation, Hamilton et. al. provide some additional notation for working with an LTS:

- $e \xrightarrow{\alpha} e'$ represents $(e, \alpha, e') \in \rightarrow$.
- $e \rightarrow (\alpha_1, t_1), ..., (\alpha_n, t_n)$ represents an LTS containing root state e where $t_1, ..., t_n$ are LTSs obtained by following transitions labelled $\alpha_1, ..., \alpha_n$ from e.
- $e \Rightarrow e'$ can be used if and only if there exists a potentially empty set of silent transitions from e to e'.
- In the case of a non-silent action α, $e_1 \stackrel{\alpha}{\Rightarrow} e_2$ can be used if and only if there exists e_1 and e_2 such that $e_1 \Rightarrow e_1' \xrightarrow{\alpha} e_2' \Rightarrow e_2$

Comparisons of program behavior can be completed using by using weak bisimilarity, defined below:

Definition (Weak Simulation) The binary relation $\mathcal{R} \subseteq \mathcal{E}_1 X \mathcal{E}_2$ is a weak simulation of a pure LTS $(\mathcal{E}_1, e_0^1, \rightarrow_1, Act_1)$ by another pure LTS $(\mathcal{E}_2, e_0^2, \rightarrow_2, Act_2)$ if $(e_0^1, e_0^2) \in \mathcal{R}$ and for every pair $(e_1, e_2) \in \mathcal{R}, \alpha \in Act_1, e_1' \in \mathcal{E}_1$ it holds that if $e_1 \stackrel{\alpha}{\Rightarrow} e_1'$ then
$\exists e_2' \in \mathcal{E}_2 . e_2 \stackrel{\alpha}{\Rightarrow} e_2' \wedge (e_1', e_2') \in \mathcal{R}$ □

Definition (Weak Bisimulation) A *weak bisimulation* is a binary relation $\mathcal{R}$ such that itself and its inverse are weak simulations. □

Definition (Weak Bisimilarity) If a weak bisimulation $\mathcal{R}$ exists between two pure LTSs, then there exists a unique maximal one, denoted $\sim$. □

4 Transformation Techniques using Labelled Transformation Systems

Supercompilation, [12], and distillation [11], are both program transformation techniques aimed at reducing the use of intermediate data during the evaluation of programs. The goal of the present paper is to compare distillation with positive supercompilation. For this purpose the paper presents a formulation of positive supercompilation in LTS terms, shown in Figure 4, and compares it with an LTS formulation of distillation, shown in Figure 5. At the core of both of these techniques, is a process known as driving, which is essentially a forced unfolding, applied in a top-down fashion, to construct potentially infinite trees of states and transitions. Within driving, all function applications are removed, and will only be re-introduced due to generalization, $\mathcal{G}$, via **let** transitions at a later point. These driving rules, $\mathcal{D}$, are the transformation rules that define the technique, and are applicable to all terms that satisfy the language definition above.

$$\mathcal{D}[\![e]\!] = \mathcal{D}'[\![e]\!]\ \emptyset$$

$$\mathcal{D}'[\![e = x\ e_1 \ldots e_n]\!]\ \theta$$
$$= \begin{cases} e \rightarrow (\tau_{\downarrow\theta(x)}, \mathcal{D}'[\![\theta(x)\ e_1 \ldots e_n]\!]\ \theta), & \text{if } x \in dom(\theta) \\ e \rightarrow (x, \mathbf{0}), (\#1, \mathcal{D}'[\![e_1]\!]\ \theta), \ldots, (\#n, \mathcal{D}'[\![e_n]\!]\ \theta), & \text{otherwise} \end{cases}$$
$$\mathcal{D}'[\![e = c\ e_1 \ldots e_n]\!]\ \theta = e \rightarrow (c, \mathbf{0}), (\#1, \mathcal{D}'[\![e_1]\!]\ \theta), \ldots, (\#n, \mathcal{D}'[\![e_n]\!]\ \theta)$$
$$\mathcal{D}'[\![e = \lambda x.e]\!]\ \theta = e \rightarrow (\lambda x, \mathcal{D}'[\![e]\!]\ \theta)$$
$$\mathcal{D}'[\![e = E[f]]\!]\ \theta = e \rightarrow (\tau_f, \mathcal{D}'[\![E[e]]\!]\ \theta) \quad \text{where } (f = e) \in \Delta$$
$$\mathcal{D}'[\![e = E[(\lambda x.e_0)\ e_1]]\!]\ \theta = \mathcal{D}'[\![E[e_0]]\!]\ (\theta \cup \{x \mapsto e_1\})$$
$$\mathcal{D}'[\![e = E[\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]]\!]\ \theta$$
$$= \begin{cases} e \rightarrow (\tau_\beta, \mathcal{D}'[\![E[\mathbf{case}\ \theta(x)\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]]\!]\ \theta), & \text{if } x \in dom(\theta) \\ e \rightarrow (\mathbf{case}, \mathcal{D}'[\![x]\!]\ \theta), (p_1, \mathcal{D}'[\![E[e_1]]\!]\ (\theta \cup \{x \mapsto p_1\})), & \\ \qquad \cdots, & \\ \qquad (p_k, \mathcal{D}'[\![E[e_k]]\!]\ (\theta \cup \{x \mapsto p_k\})), & \text{otherwise} \end{cases}$$
$$\mathcal{D}'[\![e = E[\mathbf{case}\ (x\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k]]\!]\ \theta$$
$$= e \rightarrow (\mathbf{case}, \mathcal{D}'[\![x\ e_1 \ldots e_n]\!]\ \theta), (p_1, \mathcal{D}'[\![E[e'_1]]\!]\ \theta), \ldots, (p_k, \mathcal{D}'[\![E[e'_k]]\!]\ \theta)$$
$$\mathcal{D}'[\![e = E[\mathbf{case}\ (c\ e_1 \ldots e_n)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k]]\!]\ \theta$$
$$= e \rightarrow (\tau_c, \mathcal{D}'[\![E[e'_i]]\!]\ (\theta \cup \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\})) \quad \text{where } p_i = c\ x_1 \ldots x_n$$

Fig. 3. Driving Rules

Within both of these transformation systems, driving is performed on an input expression, performing a normal order reduction, in which wherever possible, silent transitions are generated. Whenever an expression without a reduction is encountered, a non-silent transition is generated. If **case** expressions cannot be evaluated, then LTS transitions are generated for their branches with infor-

mation propagated according to each branches pattern. As driving results in a *potentially infinite* labelled transition system, obviously the transformation process cannot stop here, as its results so far may be infinite, and as such, an important issue in both systems is that of termination. Each system approaches this in a similar, but importantly, different manner.

Both make use of both *folding* and *generalization* to guide termination. Generalization is performed when the risk of there being a potentially infinite unfolding has been detected. The distinction between the approach of the two systems is that supercompilation performs both of these on previously encountered *expressions*, while distillation performs these on previously encountered *labelled transition systems*. This difference is quite significant as an LTS obviously contains a lot more information than just a sole expression.

In supercompilation, folding, $\mathcal{F}_s$, is performed upon encountering a renaming of a previously encountered expression, and generalization, $\mathcal{G}_s$, is performed upon encountering an embedding of a previously encountered expression. In distillation, folding, $\mathcal{F}_d$, is performed upon encountering a renaming of a previously encountered LTS, and generalization, $\mathcal{G}_d$, is performed upon encountering an embedding of a previously encountered LTS. In both cases, an embedding is defined by a homeomorphic embedding relation.

In both systems, given an input expression, e, the driving rules above, $\mathcal{D}$ are applied resulting in an LTS, $\mathcal{D}[\![e]\!]$. Next, transformation rules, $\mathcal{T}$, for which distillation is shown in Figure 5 and supercompilation in Figure 4, are applied resulting in an LTS, $\mathcal{T}[\![\mathcal{D}[\![e]\!]]\!]$. Finally, once termination has been guaranteed via generalization, residualization rules $\mathcal{R}$, shown in Figure 6 are applied, resulting in a residualized program, $\mathcal{R}[\![\mathcal{T}[\![\mathcal{D}[\![e]\!]]\!]]\!]$.

$$
\begin{array}{ll}
\mathcal{T}_s[\![e]\!] & = \mathcal{T}'_s[\![e]\!]\ \emptyset \\
\mathcal{T}'_s[\![e \to (\tau_f, e')]\!]\ \rho & = \begin{cases} \mathcal{F}_s[\![e'']\!]\ \rho, & \text{if } \exists e'' \in \rho . e'' \equiv e \\ \mathcal{T}'_s[\![\mathcal{G}_s[\![e'']\!]\ [\![e]\!]\ \sigma]\!]\ \rho\ , & \text{if } \exists e'' \in \rho . e'' \bowtie_s e \\ e \to (\tau_f, \mathcal{T}'_s[\![e']\!]\ (\rho \cup \{e\})\ ), & \text{otherwise} \end{cases} \\
\mathcal{T}'_s[\![e \to (\tau_\beta, e')]\!]\ \rho & = \begin{cases} \mathcal{F}_s[\![e'']\!]\ \rho, & \text{if } \exists e'' \in \rho . e'' \equiv e \\ \mathcal{T}'_s[\![\mathcal{G}_s[\![e'']\!]\ [\![e]\!]\ \sigma]\!]\ \rho\ , & \text{if } \exists e'' \in \rho . e'' \bowtie_s e \\ e \to (\tau_\beta, \mathcal{T}'_s[\![e']\!]\ (\rho \cup \{e\})\ ), & \text{otherwise} \end{cases} \\
\mathcal{T}'_s[\![e\ \to (\alpha_1, e_1), \ldots, (\alpha_n, e_n)]\!]\ \rho & = e\ \to (\alpha_1, \mathcal{T}'_s[\![e_1]\!]\ \rho\ ), \ldots, (\alpha_n, \mathcal{T}'_s[\![e_n]\!]\ \rho\ ) \\
\mathcal{T}'_s[\![e \to (\mathbf{let}, e_0), (x, e_1)]\!]\ \rho & = e \to (\mathbf{let}, \mathcal{T}'_s[\![e_0]\!]\ \rho\ ), (x, \mathcal{T}'_s[\![e_1]\!]\ \rho\ )
\end{array}
$$

Fig. 4. Transformation Rules for Supercompilation

The supercompilation transformation system takes the results of $\mathcal{D}[\![e]\!]$ which is a labelled transition system. As $\mathcal{D}[\![e]\!]$ can be infinite, the transformation rules $\mathcal{T}_s$ only traverse a finite portion lazily from the root. Generalization rules $\mathcal{G}_s$ are applied if the danger of an infinite unfolding (due to recursive function calls) is detected, and a "whistle is blown". When a whistle is blown it indicates the detection of a homeomorphic embedding of a previously encountered *expression*,

and application of these rules results in an LTS with no danger of infinite folding, and a finite set of expressions on any path from it's root. Once the system has been generalized, folding rules, $\mathcal{F}_s$ are applied. Folding takes a generalized LTS and produces a bisimilar LTS, one with a finite number of states that can be residualized into an expression using the residualization rules $\mathcal{R}$.

$$\mathcal{T}_d[\![e]\!] \ = \mathcal{T}'_d[\![e]\!]\ \emptyset\ \emptyset$$

$$\mathcal{T}'_d[\![t = e \to (\tau_f, t')]\!]\ \rho\ \theta \qquad = \begin{cases} \mathcal{F}_s[\![t'']\!]\ \sigma, & \text{if } \exists t'' \in \rho, \sigma.t'' \approx t\sigma \\ \mathcal{T}'_d[\![\mathcal{G}_d[\![t'']\!]\ [\![t]\!]\ \theta\ \sigma]\!]\ \rho\ \phi, & \text{if } \exists t'' \in \rho, \sigma.t'' \bowtie_d t\sigma \\ e \to (\tau_f, \mathcal{T}'_d[\![t']\!]\ (\rho \cup \{t\})\ \theta), & \text{otherwise} \end{cases}$$

$$\mathcal{T}'_d[\![t = e \to (\tau_\beta, t')]\!]\ \rho\ \theta \qquad = \begin{cases} \mathcal{F}_s[\![t'']\!]\ \sigma, & \text{if } \exists t'' \in \rho, \sigma.t'' \approx t\sigma \\ \mathcal{T}'_d[\![\mathcal{G}_d[\![t'']\!]\ [\![t]\!]\ \theta\ \sigma]\!]\ \rho\ \phi, & \text{if } \exists t'' \in \rho, \sigma.t'' \bowtie_d t\sigma \\ e \to (\tau_\beta, \mathcal{T}'_d[\![t']\!]\ (\rho \cup \{t\})\ \theta), & \text{otherwise} \end{cases}$$

$$\mathcal{T}'_d[\![e\ \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)]\!]\ \rho\ \theta = e\ \to (\alpha_1, \mathcal{T}'_d[\![t_1]\!]\ \rho\ \theta), \ldots, (\alpha_n, \mathcal{T}'_d[\![t_n]\!]\ \rho\ \theta)$$

$$\mathcal{T}'_d[\![t = e \to (\mathbf{let}, t_0), (x, t_1)]\!]\ \rho\ \theta$$

$$= \begin{cases} \mathcal{T}'_d[\![t_0\{x \mapsto x'\}]\!]\ \rho\ \theta, & \text{if } \exists (x' \mapsto t_2) \in \theta.t_1 \approx t_2 \\ e \to (\mathbf{let}, \mathcal{T}'_d[\![t_0]\!]\ \rho\ (\theta \cup \{x \mapsto t_1\})), & \\ \quad (x, \mathcal{T}'_d[\![t_1]\!]\ \rho\ \theta), & \text{otherwise} \end{cases}$$

Fig. 5. Transformation Rules for Distillation

The distillation transformation system proceeds in a similar manner, taking the results of $\mathcal{D}[\![e]\!]$ and applying $\mathcal{T}_d$ to this LTS resulting in another LTS with a finite set of states. In distillation however the whistle is blown when an embedding of a previously encountered *LTS* is detected and generalization, $\mathcal{G}_d$, then ensures that a renaming of a previously encountered LTS will be found. Folding, $\mathcal{F}_d$ is then applied to the LTS resulting in one with a finite set of states and then this can be residualized into a program using the residualization rules $\mathcal{R}$. The main difference between the two systems is that in supercompilation the comparisons are performed on *expressions*, and in distillation they are performed on *LTS*'s.

The questions related to correctness and termination, as well as the details of folding and generalization, are out of the scope of the present paper, because they have been earlier addressed in [12,11].

As mentioned previously, both transformation systems aim to remove intermediate data from their inputs, resulting in a more efficient output. This is done through removing silent transitions, defined previously. Briefly, these are either function unfoldings, β-reductions or constructor eliminations, and these are strongly linked to how powerful each system is. In the case of supercompilation, as it looks at expressions, there can only be a constant number of silent transitions between recursive calls of a function, the removal of which leads to a potentially linear increase in efficiency [27]. Distillation however, allows for an increasing number of silent transitions between recursive calls of a function, allowing for a potentially super-linear increase in efficiency [11]. This is more complex than supercompilation, as the LTS's used for comparison can be infi-

$$\mathcal{R}[\![e]\!] \ = \mathcal{R}'[\![e]\!]\ \emptyset$$

$$\begin{array}{ll}
\mathcal{R}'[\![e \to (x, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)]\!]\ \varepsilon & = x\ (\mathcal{R}'[\![t_1]\!]\ \varepsilon) \ldots (\mathcal{R}'[\![t_n]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)]\!]\ \varepsilon & = c\ (\mathcal{R}'[\![t_1]\!]\ \varepsilon) \ldots (\mathcal{R}'[\![t_n]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (\lambda x, t)]\!]\ \varepsilon & = \lambda x.(\mathcal{R}'[\![t]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (\mathbf{case}, t_0)(p_1, t_1), \ldots, (p_n, t_k)]\!]\ \varepsilon & = \mathbf{case}\ (\mathcal{R}'[\![t_0]\!]\varepsilon)\ \mathbf{of}\ p_1 \Rightarrow (\mathcal{R}'[\![t_1]\!]\varepsilon) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p_k \Rightarrow (\mathcal{R}'[\![t_k]\!]\varepsilon) \\
\mathcal{R}'[\![e \to (\mathbf{let}, t_0), (x, t_1)]\!]\ \varepsilon & = (\mathcal{R}'[\![t_0]\!]\ \varepsilon)\{x \mapsto (\mathcal{R}'[\![t_1]\!]\ \varepsilon)\} \\
\mathcal{R}'[\![e \to (\tau_c, t)]\!]\ \varepsilon & = \mathcal{R}'[\![t]\!]\ \varepsilon
\end{array}$$

$$\mathcal{R}'[\![e \to (\tau_f, t)]\!]\ \varepsilon = \begin{cases} f'\ x_1 \ldots x_n, \text{ if } \exists (f'\ x_1 \ldots x_n = e) \in \varepsilon \\ f'\ x_1 \ldots x_n\ \mathbf{where}\ f' = \lambda x_1 \ldots x_n.(\mathcal{R}'[\![t]\!]\ (\varepsilon \cup \{f'\ x_1 \ldots x_n = e\})), \\ \qquad\qquad \text{otherwise } (f' \text{ is fresh}, \{x_1 \ldots x_n\} = fv(t)) \end{cases}$$

$$\mathcal{R}'[\![e \to (\tau_\beta, t)]\!]\ \varepsilon = \begin{cases} f\ x_1 \ldots x_n, \text{ if } \exists (f\ x_1 \ldots x_n = e) \in \varepsilon \\ f\ x_1 \ldots x_n\ \mathbf{where}\ f = \lambda x_1 \ldots x_n.(\mathcal{R}'[\![t]\!]\ (\varepsilon \cup \{f\ x_1 \ldots x_n = e\})), \\ \qquad\qquad \text{otherwise } (f \text{ is fresh}, \{x_1 \ldots x_n\} = fv(t)) \end{cases}$$

Fig. 6. Rules For Residualization

nite, whereas the expression compared in supercompilation is not, however distillation benefits from the fact that as any infinite sequence of transitions must contain either a function unfolding or substitution, once one of these is detected, no further comparison needs to be done.

5 Two-Level Supercompilation

There are many other approaches to our supercompilation technique, such as *supero* [22,21]. However, we expect such supercompilation systems to be similar in nature and success as our system. Another more powerful approach to the removal of intermediate data is that of *two-level supercompilation* [19,18]. The authors of this approach supercompilation as a multi-level program transformation technique, using a lower and an upper level supercompiler to transform an input program. Like distillation, this technique is capable of obtaining a super-linear increase in efficiency, but was originally intended as an analysis tool.

Using a multi-level approach is based upon the concept of *meta-system transition* presented by Turchin in [29]. Briefly, a meta-system S' is a system composed of copies and/or variations of another system S and means of controlling these copies of S. If there exists another meta-system S'' composed of copies of S', then the hierarchy of control in this system is obvious.

In two-level supercompilation, S' is a hierarchical meta-system composed of copies of a 'classic' supercompiler S, that each can control each other. Lower level supercompilers generate improvement lemmas [24] - improved expressions equivalent to an input expression - guided by the expressions labeling the nodes of its partial process tree [19]. These improvement lemmas can then be used to guide the upper level supercompiler in its search for improvement lemmas.

The supercompilers at the different levels of a two-level supercompiler can differ in behavior, allowing for an upper and lower level supercompiler to produce different results. Using its hierarchy of possibly differing supercompilers, it, like distillation is capable of obtaining a super-linear increase in efficiency.

6 Comparison of Techniques

Based upon the fact that distillation is significantly more powerful than supercompilation, we have implemented both techniques for comparison using the popular functional language Haskell [15]. As two-level supercompilation is also more powerful than supercompilation and can obtain results similar to that of distillation, we would have liked to included it in our comparison, however we had difficulties getting the tool functioning, so we have included the output of the HOSC [19] single level supercompiler. To perform these comparisons, we represent parsed Haskell programs into a simple representation of the language similar to that shown in Figure 1. It is worth noting that our transformation tool is still a work in progress, and as such our comparison is on a set of relatively 'simpler' programs, that match up pretty closely to the language definition, with more advanced features of the Haskell language being disallowed.

The benchmarks we used are as follows: *sumsquare* [4], a function that calculates the sum of the squares of two lists; *wordcount*, *linecount* and *charcount* [22] are functions that respectively count the number of words, lines and characters in a file; $factorial$, $treeflip$ and *raytracer* are programs used to benchmark previous supercompilation work [16]; *nrev* represents a naive list reversal program; and a sample of programs from the well known Haskell nofib benchmark suite[13]. We present the results obtained after supercompiling, distilling and applying HOSC to our set of benchmarks, with focus on both execution time and memory usage. The results of the transformation systems are shown as a percentage of the measure used for the unoptimized version, and a lower percentage will obviously indicate an improvement.

6.1 Findings

In Figure 7, we present our findings of the execution time performance of our set of benchmark programs based on seconds taken to complete execution. From left to right, the columns describe the name of the sample program, its unoptimized execution time (in seconds), the time taken by the supercompiled version, the time taken by the output of HOSC, and finally, the time taken by the distilled version.

In Figure 8 we present our findings of the memory performance of our set of benchmark programs based on maximum memory allocated on the heap during execution time. From left to right, the columns describe the name of the sample program, its unoptimized total memory usage (in MB), the memory used by the supercompiled version, the memory used by the output of HOSC, and finally, the memory used by the distilled version.

Name	Unoptimized	Supercompilation	HOSC	Distillation
nrev	62.5	53.3	68.7	0.1
charcount	0.01	0.01	0.01	0.01
exp3_8	45.9	32.4	52.1	-
factorial	2.6	2.5	2.8	-
linecount	28.7	0.01	0.01	0.01
primes	79.2	75.9	104.5	-
raytracer	12.7	10.0	10.4	10.0
rfib	57.7	35.3	37.7	-
sumsquare	81.9	72.7	76.9	-
treeflip	51.2	29.9	32.2	-
wordcount	29.8	0.01	0.01	0.01

Fig. 7. Execution Time

Name	Unoptimized	Supercompilation	HOSC	Distillation
nrev	8	6	11	3
charcount	3	3	3	3
exp3_8	6	4	6	-
factorial	3	3	3	-
linecount	6	1	1	1
primes	2	2	2	-
raytracer	1011	730	732	732
rfib	2073	1061	1047	-
sumsquare	2313	2391	2221	-
treeflip	2176	1083	1069	-
wordcount	6	1	1	1

Fig. 8. Memory Usage

There are some interesting conclusions that can be drawn from both Figure 7 and Figure 8. A quick review of our execution time and memory usage findings show that, while the distillation tool is incomplete at this point it in no case underperforms either of the other two optimization techniques with respect to execution time. It also reveals that our implementation of supercompilation fares better than that of HOSC in terms of execution time. In the following sections we present the following: firstly a comparison of our implementation of supercompilation and HOSC, and secondly a comparison of the supercompilation techniques and our implementation of distillation. It is worth noting at this point that, as mentioned above, our implementation of distillation is a work in progress, and as a result of this there are some programs that we were unable to optimize.

Supercompilation Techniques With respect to execution time, and the test cases involved, we find that our implementation of supercompilation results in

more outputs with improved efficiency than that of HOSC. We also find that our supercompiler results in a reduced execution time, whereas HOSC does not and in some cases results in a drastic increase in execution time.

This reduction in performance is most pronounced in the case of *primes*, where for the same input, the original unoptimized version has an execution time of 79.2 seconds, our super-compiled version takes 75.9 seconds and HOSC takes 104.5 seconds. Compared with the original our implementation results in a 4.16% reduction in execution time, and HOSC results in a 31.94% increase in execution time over the original and a 37.68% increase in execution time over our implementation of supercompilation. In the cases of *charcount*, *linecount* and *wordcount* both implementations resulted in the same decrease of execution time, with *charcount* seeing no increase in efficiency, which was expected and *linecount* and *wordcount* seeing a 99.96% increase in execution time.

With respect to memory usage, and the test cases involved, we find that our implementation of supercompilation often results in more efficient outputs than that of HOSC. In the cases of *nrev*, *exp*3_8 and *raytracer* our supercompiler gives a greater reduction in memory usage than that of HOSC, which in the case of *nrev* actually has a negative impact on memory usage. The original program has a maximum allocation of 8 MB, our supercompiled version uses 6 MB, a reduction of 25%, and the HOSC supercompiled version uses 11 MB, an increase of 37.5%.

However there are also some cases where HOSC seems to obtain a slight decrease in memory usage when compared to both the original and our supercompiler, i.e. $rfib$, and $treeflip$. In the case of *sumsquare* our supercompiled version results in an output that is less efficient than both the original program and the output of HOSC with respect to memory usage. The original program uses 2313 MB, the output of HOSC uses 2221 MB and our supercompiled version uses 2391 MB. Our supercompiled version represents a 3.37% increase in memory usage, and the HOSC supercompiled version represents 3.98% decrease in memory usage over the original and a 7.65% decrease in memory usage over our supercompiled version.

Distillation vs. Supercompilation As our sample set of distillable programs is limited, we have few benchmarks to draw conclusions from, namely *nrev*, *charcount*, *linecount*, *raytracer* and *wordcount*. With respect to execution time for these test cases, *nrev* is probably the most interesting example with the original program taking 62.5 seconds, our supercompiled version taking 53.3 seconds, the HOSC supercompiled version taking 68.7 seconds and our distilled version taking 0.1 seconds, representing a 14.72% increase, a 9.92% decrease and a 99.84% increase in efficiency respectively. An increase of this order was expected for distillation as it is capable of obtaining a super-linear increase in efficiency.

With respect to memory usage, and the same set of benchmarks, *nrev* and *raytracer* are the most interesting examples. In the case of *nrev*, the original program uses 8 MB, our supercompiled version uses 6 MB, the HOSC super-

compiled version uses 11 MB and our distilled version uses 3 MB, representing a 25% reduction, an 11% increase and a 62.5% reduction in memory usage respectively. Again this is to be expected with supercompilation. In the case of *raytracer*, the original program uses 1011 MB, our supercompiled version uses 730 MB, and both the distilled and the HOSC supercompiled versions use 732 MB, representing a 27.79% decrease and a 27.6% reduction respectively. What is interesting about this is that our supercompiled version is more efficient with respect to memory usage than our distilled version, when the opposite would be expected.

7 Future Work

Work is progressing at present on extension of the distillation tool to allow it to handle some of the more powerful features of the Haskell programming language. Once it is capable of handling these more advanced features it will be applied to the benchmarks that weren't possible for this comparison. The results of these optimizations will be published, alongside those of the multi-level supercompiler mentioned previously for comparison. We aim to also support all programs in the Nofib benchmark suite, and some more real-world programs.

8 Acknowldgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre

References

1. Augustsson, L.: Compiling pattern matching. Functional Programming Languages and Computer Architecture (1985)
2. Bolingbroke, M., Jones, S.P.: Supercompilation by evalutation. Proceedings of the 2010 ACM SIGPLAN Haskell Symposium (2010)
3. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. Journal of the Association for Computing Machinery 24(1), 44–67 (January 1977)
4. Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion. from lists to streams to nothing at all. In: ICFP'07 (2007)
5. Gill, A., Launchbury, J., Jones, S.P.: A shortcut to deforestation. FPCA: Proceedings of the conference on Functional programming languages and computer architecture pp. 223–232 (1993)
6. Gordon, A.D.: Bisimilarity as a theory of functional programming. Electronic Notes in Theoretical Computer Science 1, 232 – 252 (1995)
7. Hamilton, G.W.: Higher order deforestation. Fundamenta Informaticae 69(1-2), 39–61 (July 2005)
8. Hamilton, G.W.: Distillation: Extracting the essence of programs. Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation (2007)

9. Hamilton, G.W.: Extracting the essence of distillation. Proceedings of the Seventh International Andrei Ershov Memorial Conference: Perspectives of System Informatics (2009)
10. Hamilton, G.W., Mendel-Gleason, G.: A graph-based definition of distillation. Proceedings of the Second International Workshop on Metacomputation in Russia (2010)
11. Hamilton, G., Jones, N.: Distillation and labelled transition systems. Proceedings of the ACM Workshop on Partial Evaluation and Program Manipulation pp. 15–24 (January 2012)
12. Hamilton, G., Jones, N.: Proving the correctness of unfold/fold program transformations using bisimulation. Lecture Notes in Computer Science 7162, 153–169 (2012)
13. Haskell-Community: Nofib benchmarking suite (2012), `http://darcs.haskell.org/nofib/`
14. Jones, S.P.: The Implementation of Functional Programming Languages. Prentice-Hall (1987)
15. Jones, S.P.: Haskell 98 language and libraries - the revised report. Tech. rep. (2002)
16. Jonsson, P.A., Nordlander, J.: Positive supercompilation for a higher order call-by-value language. SIGPLAN Not. 44(1), 277–288 (Jan 2009)
17. Klimov, A.V.: An approach to supercompilation for object-oriented languages: the java supercompiler case study (2008)
18. Klyuchnikov, I., Romanenko, S.: Towards higher-level supercompilation. In: Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia. pp. 82–101. Ailamazyan University of Pereslavl (2010)
19. Klyuchnikov, I.G.: Towards effective two-level supercompilation (2010)
20. Lloyd, J.W., Shepherdson, J.C.: Partial evaluation in logic programming. J. Log. Program. 11(3-4), 217–242 (1991)
21. Mitchell, N.: Rethinking supercompilation. In: ICFP '10: Proceedings of the 15th ACM SIGPLAN Internation Conference on Functional Programming. pp. 309–320. ACM (September 2010)
22. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In: et al., O.C. (ed.) IFL 2007. LNCS, vol. 5083, pp. 147–164. Springer-Verlag (May 2008)
23. Pettorossi, A., Proietti, M.: Rules and strategies for transforming functional and logic programs. ACM Comput. Surv. 28(2), 360–414 (Jun 1996)
24. Sands, D.: Total correctness by local improvement in the transformation of functional programs. ACM Transactions on Programming Languages and Systems 18, 175–234 (1996)
25. Sørensen, M., Glück, R.: An algorithm of generalization in positive supercompilation. International Logic Programming Symposium pp. 465–479 (1995)
26. Sørensen, M., Glück, R., Jones, N.: A positive supercompiler. Journal of Functional Programming 1(1) (January 1993)
27. Sørensen, M.H.: Turchin's supercompiler revisited - an operational theory of positive information propagation (1996)
28. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (June 1986)
29. Turchin, V.F.: Metacomputation: Metasystem transitions plus supercompilation. In: Selected Papers from the Internaltional Seminar on Partial Evaluation. pp. 481–509. Springer-Verlag, London, UK, UK (1996)
30. Wadler, P.: Efficient compilation of pattern matching. In: Jones, S.P. (ed.) The Implementation of Functional Programming Languages., pp. 78–103. Prentice-Hall (1987)

31. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theoretical Computer Science 73, 231–248 (1990)

Overgraph Representation for Multi-Result Supercompilation*

Sergei A. Grechanik

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

Abstract. In this paper we present a new approach to multi-result supercompilation based on joining together process graphs into a single graph and welding nodes corresponding to equal configurations. We show that this leads to a considerable reduction of nodes being built during supercompilation and enables more efficient configuration graph processing algorithms on the example of an enhanced residualization algorithm.

1 Introduction

Supercompilation [15] is traditionally seen as a program transformation which produces a single program that is equivalent in some sense to the original one. This approach is very understandable: we usually need only one program, actually the most optimal one we can produce. But this approach goes deep down the process of supercompilation. Being a complex transformation, supercompilation consists of smaller steps (like driving, folding and generalization) that can be applied in some order. The principle of single result dictates us to choose a single step each time we have a choice. This means that we ought to make decisions using some a priori heuristics which may lead us to a solution that is far from optimal. Of course there is no purity in the world and supercompilers often implement backtracking through generalization of an upper configuration.

The flaw of this approach becomes more pronounced when we consider a problem that doesn't consist in finding an optimal solution, for example proving of equivalence of two programs. The standard approach is to supercompile both of them and then compare the resultant programs syntactically [6, 10]. But we don't really need an optimal solution to do it, and since *the* optimal solution is hard to find, we would increase chances of success by constructing a set of equivalent programs for each original program and then checking if the sets intersect. Another example – program analysis. In this case a supercompiler can be used as a transformation that simplifies a program into one that is easier to analyze. But "easier to analyze" might not be the same as "more efficient" and it may be more difficult to express using heuristics.

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

Undoubtedly it is possible to stuff the supercompiler with all sorts of heuristics and then tune it to suit the specific task. This approach has the right to live. It must be said that it is also used outside the field of supercompilation, for example, it is common to automatically adjust the set of parameters of optimising compilers to a certain architecture or even to a set of programs [2, 3, 11].

An alternative approach is to dismiss the idea of the single path to the optimal program. Let supercompiler perform all possible steps simultaneously when it has a choice. This idea is known as the multi-result supercompilation [9]. It first appeared not a very long time ago and was intended to be used primarily for program analysis. It should be noted that similar idea had appeared in the field of optimizing compilers [14] where it enabled selecting the best program by evaluating the programs a posteriori using global profitability heuristics; this indicates that the idea of multiple results is advantageous for optimization problems.

The main problem of multi-resultness is lack of efficiency. Each branching point multiplies the number of programs leading to combinatorial explosion. This issue can be resolved by restricting branching using heuristics (which is not desirable but seems unavoidable) or by using some better representation of the set of programs to diminish redundancy. In this paper the latter approach is discussed.

2 The Approach of MRSC

MRSC is a multi-result supercompilation framework written in Scala [8][1]. The goal of MRSC is to provide a set of components to rapidly build various multi-result supercompilers. MRSC consists of a core which implements basic domain-independent operations over process graphs, and several domain-specific (i.e. specific to different languages and tasks) modules which give meaning to these graphs and guide their construction.

MRSC is based on explicit construction of configuration graphs. Being a multi-result supercompiler, it takes an initial configuration and produces a list of corresponding configuration graphs which is then transformed into a list of programs through the process known as residualization. MRSC issues configuration graphs incrementally. In particular, this design choice makes it possible to use MRSC as a traditional single-result supercompiler without efficiency loss by simply taking the first issued graph. On the other hand it determines that MRSC should use depth-first traversal to build and issue every next graph as quickly as possible. Thus, the MRSC supercompilation algorithm can be formulated using a stack of graphs as follows:

1. Put a graph with only one node containing the initial configuration on the stack.
2. While the stack is not empty repeat the following steps.

[1] The source code of MRSC is available at `https://github.com/ilya-klyuchnikov/mrsc`

3. Pop a graph g from the stack.
4. If the graph g is completed, issue it.
5. If the graph is incomplete, transform it according to some domain-specific rules and push the resultant graphs (there may be many of them as the supercompilation is multi-result) onto the stack

It should be noted that although we talk about graphs they are actually represented as trees with back edges (which are represented differently from normal edges).

Each graph has a list of complete nodes and a list of incomplete nodes. The first incomplete node of a graph is called the current node and represents the place where the graph is growing. A graph is completed if it doesn't contain incomplete nodes.

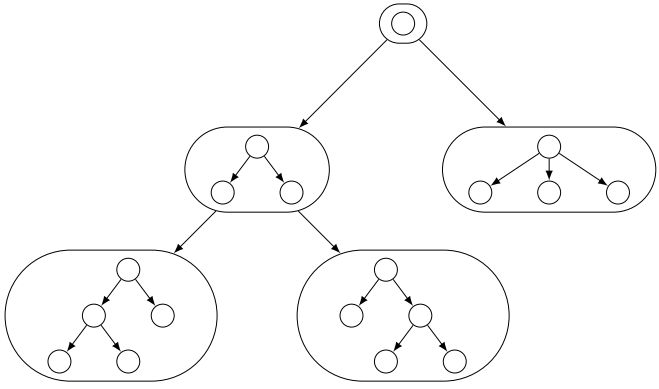


Fig. 1. Tree of graphs

Graph transforming rules are specified with a function that maps a graph into a list of steps to perform on it (by "perform" we don't mean that they actually modify it, we apply steps in a functional way). When a step is performed, the current node usually becomes complete and the next incomplete node becomes current (if there are incomplete nodes left). A step can be one of the following:

1. *CompleteCurrentNode* – just mark the current node complete and move on to the next incomplete node. It is usually used if the the current node contains a fully evaluated expression as a configuration.
2. *Fold*(b) – make a back edge from the current node to b.
3. *AddChildNodes*(cs) – append nodes to the current one.
4. *Rebuild*(c) – replace the configuration of the current node with c. This step does not make the current node complete.
5. *Rollback*(n, c) – perform an upper rebuilding: remove the subgraph that grows from the node n and then replace the configuration of n with c. n becomes current.

If there is no step produced by the rules function, the graph is thrown away as it is incomplete and cannot be completed. It is usually the case when the whistle blows but there is no possibility of generalizing, or the graph doesn't meet some conditions we require (like the safety condition in the case of counter systems).

So what if the rules prescribe several different steps to one incomplete graph? All of them will be performed on it simultaneously, producing several graphs. This leads to the idea of a tree of graphs (Fig. 1). The tree of graphs is a mental construction rather than an actual tree that resides in a memory, but it can be used to give another formulation of MRSC supercompilation algorithm: MRSC just performs the depth-first traversal of this tree filtering out incomplete graphs.

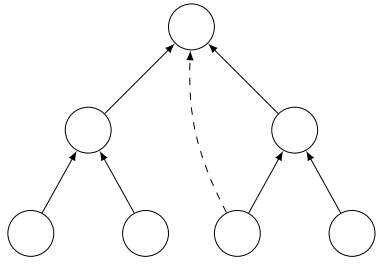


Fig. 2. Spaghetti-stack-based graph representation

When more than one step is applicable the number of graphs multiplies by the number of steps, and it may seem that MRSC doesn't cope with the problem of combinatorial explosion very well. But actually MRSC uses a clever representation to reduce memory usage by exploiting sharing of common subgraphs. This representation is based on spaghetti-stacks [1]. A graph represented this way can be thought of as a configuration graph with all edges reversed (except folding edges which are treated separately) (Fig. 2). This allows initial parts of graphs to be shared. Moreover this representation makes it possible to work with graphs in a functional way efficiently. Note that this optimization doesn't interfere with our view on graphs as separate objects.

3 Can we do better?

Certainly there is a room for improvement. Let us look when MRSC does not work very well. Consider a configuration that being driven produces two child nodes: b and c. Then the node b will become current. Let it have multiple different steps that can be applied to it. We get at least two different graphs with incomplete node c (Fig. 3). That means that the c node will be current at least twice (in different graphs) and thus each subgraph growing from it will be built at least twice and won't be shared, which might be considered as a drawback. This happens because a graph growing from a node is usually determined by the

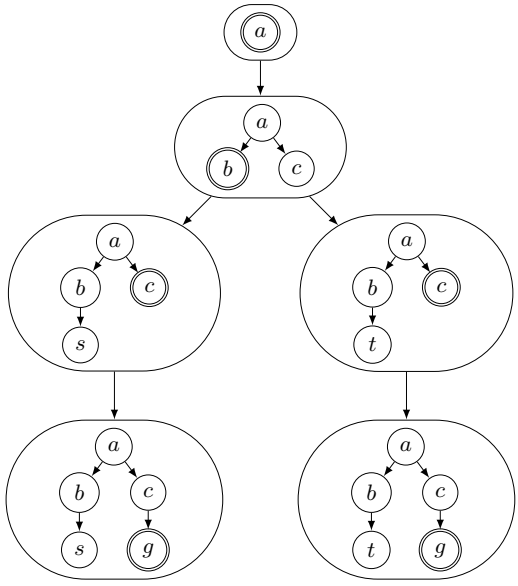


Fig. 3. Two different nodes with the same configuration and identical histories

configuration in this node and the predecessors of the node. But MRSC is too general as it assumes that rules work on whole graphs, not just paths from a root to a current node. So it is possible to write rules which prescribe different steps for the node c in the considered example. It means that we should put more restrictions on rules in order to perform more sophisticated optimizations.

Another drawback of MRSC is a premature loss of information. As graphs are seen as independent from a graph consumer point of view we cannot use more efficient algorithms that can make use of their interconnections. To give an example of such an algorithm let's consider a problem of finding the smallest program among the residual programs. A brute-force solution would be to residualize all completed graphs and then compute the sizes of the programs and find the minimum. A more clever solution would be to pick the smallest residual subprogram for each intermediate node while building the residual program. The algorithm enabling this optimization will be discussed in more detail later in this article.

Now we can see what representation would be more advantageous – let's replace a set of graphs with its description by merging the graphs into one huge graph, which we will call an *overtree* to underline that the graphs are still essentially trees with back edges. It is convenient to combine edges representing a single step into a *hyperedge* with one source and several destinations (Fig. 4, hyperedges are drawn as bundles of edges going from one point). Then it is possible to extract a graph representing a program from an overtree by selecting

one outcoming hyperedge for each node and then removing unreachable nodes. Note that it is convenient to consider terminal nodes (which represent constants, variables, etc.) as having outcoming hyperedges with zero destination nodes.

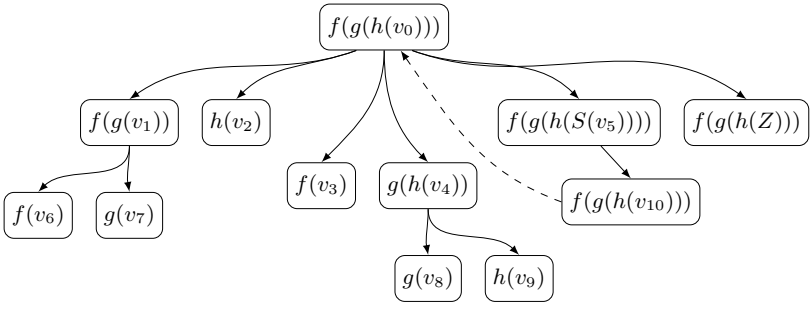


Fig. 4. Overtree representation

This representation, which we will call the overtree representation, doesn't change the graph building algorithm very much, it actually becomes much more similar to the traditional single-result supercompilation. This representation has been implemented in MRSC as an experiment. This experiment unveiled some shortcomings of the proposed representation: turned out there were a lot of equal but unshared subtrees. It is quite clear where they were coming from. Consider some complex program involving multiplication (implemented as a function on Peano numbers). There will be a lot of different ways of supercompiling this program, some of them will have the multiplication function fused with other operations, but some will prefer to generalize it out leading to multiple equal subgraphs corresponding to the multiplication and scattered over the overtree. Obviously we do not want to supercompile the same thing several times, so the next natural step is to get rid of duplicated configurations. We could have done it by introducing cross edges similarly to back edges, but there is a cleaner solution – let's shift from a de facto tree representation to a fully fledged graph (Fig. 5). That is if during supercompilation we encounter a configuration which we have already processed, we do not create a new node for it. Thus each configuration corresponds to no more than one node. This new representation can be called the *overgraph* representation. Note that configurations equal up to renaming should be identified.

Unlike the overtree representation this one seems to be a bit more groundbreaking. Special folding edges are not needed anymore as they can be represented as ordinary edges. However, we cannot safely use traditional binary whistles because possible steps cannot depend on the history of computation (and hence unary whistles can still be used). Why is it so? Because each node may have multiple immediate predecessors and thus multiple histories. Let us de-

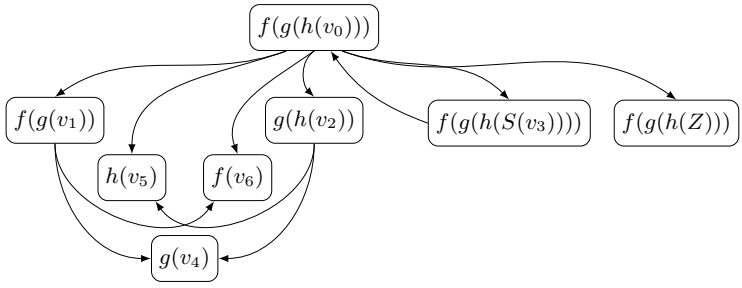


Fig. 5. Overgraph representation

scribe the overgraph representation more formally and consider how these issues can be addressed.

4 Overgraph Representation

In this section a multi-result supercompilation with overgraph representation is discussed. A configuration graph corresponds to (and actually can be represented by) a set of statements of the following form:

$$s \xrightarrow{l} (d_1, \ldots, d_n)$$

where s is a source configuration and d_i are destination configurations. Configurations are in one-to-one correspondence with nodes. The whole statement corresponds to a hyperedge labeled with some information l. The exact meaning of the statements depends on the interpretation of the relation $\rightarrow$ which is different for different applications of supercompilation. For traditional supercompilation of functional programs it can be like this:

$$s \xrightarrow{f} (d_1, \ldots, d_n) \overset{def}{\iff} s \trianglerighteq f(d_1, \ldots, d_n)$$

Note the use of the improvement relation ($\trianglerighteq$) instead of a simple equality ($=$). This is due to the necessity of ensuring correctness of folding [12,13]. Informally $a \trianglerighteq b$ means that a is not only operationally equal to b but also more efficient than b (needs fewer reduction steps to be evaluated in any context). We won't elaborate on the topic of correctness since it is not very troublesome in the case of one-level supercompilation, but importance of the improvement relation consists in asymmetry it brings. When a hyperedge connects only two nodes we cannot generally consider it undirected which could be inferred if the relation were symmetric.

If we do not put any restrictions on the set thus representing a graph, then we get an overgraph. If we want to represent a singular configuration graph then we should state that each configuration can appear to the left of the $\rightarrow$ no more

than once. Several singular graphs can be merged to form an overgraph by simple union.

The supercompilation with overgraphs is proposed to be performed in three steps:

1. Graph construction. In this step edges and nodes are only being added.
2. Graph truncation. In this step edges and nodes are only being removed.
3. Residualization. As it will be seen this step becomes a bit more nontrivial.

For construction and truncation we will use the rules of the form

$$\frac{\text{precondition}}{s \xrightarrow{l} (d_1, \ldots, d_n)} \qquad \frac{\text{precondition}}{\mathit{remove}(\text{node or hyperedge})}$$

to add a new hyperedge (together with necessary nodes) and to remove a hyperedge or a node (together with all incident hyperedges) correspondingly.

4.1 Graph Construction

This step seems to be rather straightforward: start with a graph containing only a root node (with the configuration being supercompiled) and then apply the rules until the saturation is reached, i.e. there is no applicable rule that can add a new node or edge. There are two problems: the rules formulation and the order of their application. The rules should ensure termination and provide enough diversity but not too much. The traditional way of ensuring termination is to use a whistle. But whistles have been used not only for the termination problem but also for picking the configurations that are to be generalized. Since we use the multi-result supercompilation the generalization can be decoupled from the whistle and thus a much simpler whistle can be used. For example, it is possible to simply limit the depth of nodes, although this solution does not scale very well.

Concerning the problem of diversity, there are usually a lot of possible generalizations of a configuration, and if we take all of them, even the overgraph representation won't help us fight combinatorial explosion. If we pick few of them, we risk missing some good opportunities. Obviously heuristics are unavoidable. Limiting the number of unbound variables appeared to be a good one for preventing overgeneralization. Good heuristics for multi-result supercompilation are yet to be found and researched.

A problem of rules application order may arise when there are rules with non-monotonic preconditions, i.e. a rule precondition can change its value from `true` to `false` when certain nodes or edges are *added* to the graph. For example consider the following rule:

$$\frac{\neg\exists m \colon m \in V \wedge \mathit{whistle}(n, m)}{n \to \mathit{drive}(n)}$$

where V is the set of nodes of the graph. It prescribes a drive step for a node n if there is no node m in the graph which makes a binary whistle blow. If there is another rule that adds such a node m then these two rules won't commute.

Since we agreed not to remove edges and nodes on the graph construction step, there won't be any problem if all the preconditions are monotonic. For example it is possible to use a whistle this way: let's allow driving if there *is* another node that does *not* make the binary whistle blow.

4.2 Graph Truncation

This step is dual to the previous. Its purpose is to reduce the number of nodes by removing useless ones. Useless nodes are those which are unreachable from the root or have no outcoming edges (remember that we have agreed to consider terminal nodes as having an outcoming edge without destination nodes). When a node is deleted, all hyperedges incident with it must be deleted as well, leading to new useless nodes. That's why this procedure should be repeated until there are no useless nodes left. It can be also described with the following rules:

$$\frac{\neg\exists l, d_1, \ldots, d_n : s \xrightarrow{l} (d_1, \ldots, d_n)}{remove(s)}$$

$$\frac{\neg\exists p : p \text{ is a path from the root to } s}{remove(s)}$$

In this step it is also possible to apply a whistle. On the one hand it may seem a bit too late: the graph has already been built by now, so we don't need to ensure termination on this step. Moreover, we can't return consumed CPU time by removing parts of the graph (we can return some memory though). But on the other hand experiments show that most time is being consumed later on the residualization step, so reducing the number of nodes and edges of the graph is a good thing to do. However it is still impossible to use a binary whistle in a traditional way for the same reason: the traditional usage may lead to noncommutativity of rules. At this stage to ensure commutativity preconditions should be monotonically decreasing, and actually it is possible to use rules from the previous step by negating their preconditions, i.e. "if the whistle doesn't blow, add a node" becomes "if the whistle blows, remove a node". The only advantage of using whistles in this step seems that now it is possible to look at the graph as a whole and extract some useful information to adjust the whistle.

Note also that although commutativity of rules makes a supercompiler a bit cleaner and more predictable, this property is not obligatory and can be discarded in practice.

5 Residualization

The residualization step is special because it transforms a set of graphs represented as one graph into an actual set of graphs (or programs which are a special case of graphs), and here the danger of explosion threatens again.

Firstly we should agree upon what graphs we will consider residual. In this section we will study the case of trees with back edges (but without cross edges),

i.e. we won't identify equivalent subprograms in different contexts. This choice was initially determined by the structure of the language for which the overtree representation had been initially implemented (it will be dicussed in the next section). It may look a bit inconsistent: why not go further and represent programs as graphs (especially recalling that it is a traditional program representation)? We will return to this question later in Section 6.2.

$$
\begin{aligned}
&\mathcal{R}[\![n, h]\!] = \{n\} && \text{if } n \text{ is terminal}\\
&\mathcal{R}[\![n, h]\!] = \{\mathrm{Call}(n)\} && \text{if } n \in h\\
&\mathcal{R}[\![n, h]\!] = \{\mathrm{Def}(n)[\![f(r_1, \ldots, r_k)]\!] \mid n \xrightarrow{f} (d_1, \ldots, d_k),\\
&\qquad\qquad r_i \in \mathcal{R}[\![d_i, h \cup \{n\}]\!]\} && \text{otherwise}
\end{aligned}
$$

Fig. 6. Naive residualization algorithm

Consider a naive residualization algorithm (Fig. 6). It takes a node and a history and returns a set of residual programs. It is usually applied to the root node and the empty history: $\mathcal{R}[\![root, \emptyset]\!]$. The algorithm recursively traverses the graph memorizing visited nodes in the history h. If it encounters a node that is already in the history, it creates a function call which is designated as $\mathrm{Call}(n)$. If a node is not in the history and has successors, a function definition should be created with the construction $\mathrm{Def}(n)[\![body]\!]$, so as it can be called with a $\mathrm{Call}(n)$ construction from within the *body*. The implementation of Call and Def depends on the output language, for example the Def construction can be implemented with letrec expressions:

$$\mathrm{Def}(n)[\![body]\!] = \textbf{letrec } n = body \textbf{ in } n$$

Usually it is a bit more complex because of unbound variables. Note also that practical implementations of the algorithm should create a new function definition only if there is a corresponding function call in the body, we just create a function definition for each node for simplicity.

When applied, this residualization algorithm will visit each node the number of times equal to the number of computation paths from the root to it. So the problem reappeared: we need to compute (almost) the same thing many times.

5.1 Enhanced Residualization Algorithm

The solution to the problem is quite obvious – cache residual programs for intermediate nodes. It cannot be applied directly though because the naive residualization algorithm takes a history of computation besides a node. However, residualization doesn't need full information contained in a history, i.e. the residual program for a node may be the same for different histories. So if we can do with less information, the algorithm will be feasible to memoize.

To do this we need analyze the structure of a computation history for a given node. Let's first give a couple of definitions.

Definition *A node m is a successor of a node n, $n \to^* m$, if there is a directed path from n to m (possibly with zero length). A set of successors will be denoted as $succs(n)$.*

Definition *A node m is a predecessor of a node n if n is a successor of m. A set of predecessors will be denoted as $preds(n)$.*

Definition *A node n dominates a node m, n dom m, if every path from the root to m contains n.*

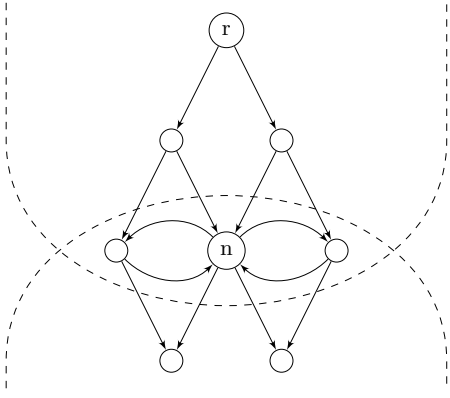


Fig. 7. Nodes whose presence in a history can influence residualization process are in the intersection of the set of all predecessors and the set of all successors

Given a node n, the nodes that are not its predecessors are not interesting as they won't be in a history. The nodes that are not successors are not interesting either because they don't influence the residualization process (they can be in a history but they won't be encountered again). Thus we care only about nodes which are successors and predecessors of n at the same time (i.e. those which are in the same strongly connected component with n, Fig. 7), so we just need to remove nodes that are not successors from history when calling the residualization function:

$$\mathcal{R}[\![n, h]\!] = \big\{ \mathrm{Def}(n)[\![f(r_1, \ldots, r_k)]\!] \mid n \xrightarrow{f} (d_1, \ldots, d_k), \\ r_i \in \mathcal{R}[\![d_i, (h \cup \{n\}) \cap succs(d_i)]\!] \big\}$$

This small modification to the last equation of the residualization algorithm is sufficient to make it ready for memoization. An interesting side-effect of the

memoization is that we get residual programs for intermediate nodes for free. These programs can then be used to enable two-level supercompilation (although the topic of two-level supercompilation in context of overgraph representation is yet to be researched as few results have been obtained so far). Note though that the residual programs for intermediate nodes are likely to be defined in terms of other nodes, being used as if they were built-in functions.

Let's look at the structure of a history in more detail. It can contain only nodes from $S(n) = preds(n) \cap succs(n)$. The elements of this set fall into the following groups:

- Dominators of n except n. These elements are always in a history.
- Dominatees of n except n. If there is such a node in a history then the history must also contain the node n and this falls into the second equation of the algorithm.
- Neither dominators nor dominatees of n. These nodes are responsible for the diversity of possible histories.
- n itself.

We believe that the information about the history structure can be used somehow to develop more sophisticated residualization algorithms.

6 Implementation and Experimental Results

The described overgraph representation has been implemented as a part of MRSC. The implementation is experimental and not language-independent yet. The language being used is higher-order although it is meant to be used primarily as first-order.

$e ::= v$		variable
$\mid$	$\lambda v.e$	λ-abstraction
$\mid$	$e_1\ e_2$	application
$\mid$	**fix** e	fixed point
$\mid$	$c\ e_1 \ldots e_n$	constructor
$\mid$	**case** e_0 **of** $\{c_1\ v_1 \ldots v_{k_1} \to e_1; \ \ldots\}$	case-expression

The explicit fixed point operator is meant to be the only source of nontermination. There are no let-expressions, so there is no possibility of expressing sharing. This was done for the sake of simplicity.

Configurations are just expressions. They may contain unbound variables which are named in some canonical way (e.g. numbered from left to right), so as to make expressions, equal up to a renaming, syntactically equal. Configurations are self-contained, i.e. we don't need some external set of function definitions to understand their meaning (this is necessary for higher-level supercompilation [4]).

The following rules are used for graph construction:

$$depth(n) = \min\{|p| \mid p \text{ is a path from the root to } n\}$$

$$\frac{depth(n) \leq MaxDepth \qquad (f, d) = drive(n)}{n \xrightarrow{f} d}$$

$$\frac{depth(n) \leq MaxDepth \qquad (f, g, h) \in rebuildings(n) \qquad |\mathrm{FV}(g)| \leq MaxFree}{n \xrightarrow{f} (g, h)}$$

The *drive* function performs a drive step and returns a pair (f, d) where d is a reduced expression, or a tuple of expressions if it is a case analysis step, and f is a function such that $f(d) = n$. The function $rebuildings(n)$ returns a set of triples (f_v, g, h) where f_v is a substitution function such that for all expressions x and y

$$f_v(x, y) = x[v := y]$$

and expressions g and h are such that

$$f_v(g, h) = g[v := h] = n$$

where v appears in g exactly once. In theory, the latter restriction can be lifted leading to the ability of introducing sharing (note that we don't need let-expressions as the configuration is being immediately split).

The constants *MaxDepth* and *MaxFree* limit the depth of nodes and the maximum number of free variables in configurations respectively.

The functions Call and Def for this language look like this:

$$\mathrm{Call}(n) = v_n\ \overrightarrow{x}$$
$$\mathrm{Def}(n)[\![b]\!] = \mathbf{fix}\ (\lambda v_n.\lambda\overrightarrow{x}.\ b)$$

where the variable v_n has a unique name corresponding to the configuration n and $\overrightarrow{x}$ is a vector of n's free variables.

6.1 Results

The graph representations and the residualization algorithms have been assessed on the following programs:

$$\begin{aligned}
\mathrm{add} &= \mathbf{fix}\ \lambda fxy.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to S\ (f\ x\ y);\ Z \to y;\}\\
\mathrm{mul} &= \mathbf{fix}\ \lambda fxy.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to \mathrm{add}\ y\ (f\ x\ y);\ Z \to Z;\}\\
\mathrm{fict} &= \mathbf{fix}\ \lambda fxy.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to f\ x\ (S\ y);\ Z \to Z;\}\\
\mathrm{idle} &= \mathbf{fix}\ \lambda fx.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to f\ (f\ x);\ Z \to Z;\}\\
\mathrm{evenBad} &= \mathbf{fix}\ \lambda fx.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to \mathbf{case}\ (f\ x)\ \mathbf{of}\ \{T \to F;\ F \to T;\};\\
&\qquad\qquad Z \to T;\}\\
\mathrm{nrev} &= \mathbf{fix}\ \lambda fx.\mathbf{case}\ x\ \mathbf{of}\ \{S\ x \to \mathrm{add}\ (f\ x)\ (S\ Z);\ Z \to Z;\}
\end{aligned}$$

At this stage of research some numerical characteristics of the described ideas were measured rather than their ability to solve problems. It can be seen that the overgraph representation leads to much fewer nodes even in comparison with the overtree representation which enables full sharing of initial subtrees (Fig. 8).

	overtree	overgraph
add	7	6
mul	379	77
fict	132	30
idle	5237	139
evenBad	27307	242
nrev	46320	277

Fig. 8. Comparison of graph representations: the number of nodes created by the supercompiler, $MaxDepth = 10$

At the same time the results of the caching residualization algorithm look much more modest (Fig. 9). Note that in this case the *MaxDepth* constant was chosen to be 6, so the residualization could terminate in a reasonable amount of time. The number of nodes left after truncation are shown to compare with the number of node visits each algorithm makes. Although the caching algorithm performs much better than the naive one, the growth of the node visits is too rapid.

	nodes after truncation	nodes visited		residuals
		naive	caching	
add	6	9	8	1
mul	19	81	53	4
fict	13	52	48	4
idle	33	2413	682	112
evenBad	76	33223	2751	229
nrev	30	4269	402	19

Fig. 9. Comparison of residualization algorithms, $MaxDepth = 6$

6.2 Application to Counter Systems

There was a hypothesis that the overgraph representation would be advantageous for other domains. It was checked on the domain of counter transition systems. This choice was quite obvious as a considerable work had been done to implement and evaluate a supercompiler for counter systems within MRSC [5], so we had to simply replace the core of this supercompiler and compare it with the original one.

Let's briefly describe the domain. Counter systems are similar to programs but much simpler. A system can be in some state represented by a tuple of integers. The system can nondeterministically move from one state to another according to certain rules. Configurations represent sets of states and are actually tuples of integers and wildcards, wildcard meaning "any number". So a configuration can be driven by applying rules, or generalized by replacing a number with the wildcard. The task is to find a minimal correctness proof for some protocol modelled by a counter system. For each protocol certain states are unsafe, so a protocol is correct if the corresponding counter system can't turn out in an unsafe state. A proof is actually a completed graph (i.e. each node has an outcoming edge) without unsafe configurations. The whistle used in MRSC for this task was a unary one, so it was used with the overgraph-based core without any problem.

The results of applying the overgraph representation with the residualization algorithm described above were disappointing. The supercompiler with overgraph core not only spent much more time than the original one, but also failed to find the minimal proofs. It is easy to see why this happened – our residualization algorithm was designed to find trees with back edges, not graphs, and thus equivalent proof parts were duplicated. On the other hand, the original implementation of supercompiler had the ability to introduce cross edges which made it almost as powerful as the overgraph supercompiler.

	original	overgraph truncation
Synapse	12	12
MSI	10	10
MOSI	36	34
MESI	46	18
MOESI	156	57
Illinois	58	19
Berkley	50	33
Firefly	18	15
Futurebus	476106	1715
Xerox	94	43
Java	109410	12165
ReaderWriter	2540	154
DataRace	21	12

Fig. 10. Number of nodes visited by the supercompilers for each protocol

So the residualization algorithm must have been replaced with some different algorithm. We have tried a depth-first search with pruning of too large graphs which is exactly what MRSC was doing. So the only important difference left was that our implementation built an overgraph explicitly. Actually there is an advantage of explicit overgraph building: an overgraph can be truncated and thus

we can avoid obviously useless branches. It was quite difficult to compare the implementations fairly because of multiple subtle differences affecting the results. The comparison of the number of nodes visited during the supercompilation by the original supercompiler and the supercompiler with overgraph truncation enabled is shown on Figure 10. As can be seen, truncation turned out to be quite advantageous on complex protocols.

This unsuccessful application of the presented earlier residualization algorithm to another domain doesn't mean that it is useless. It was just applied to the kind of tasks it hadn't been designed for, namely to the task of optimizing for size (in this case a proof size). It is very hard to divide a task of graph minimization into subtasks of subgraph minimization because these subgraphs may have intersections, i.e. the minimal size of the whole doesn't equal to the sum of the minimal sizes of the parts. However if we want the same node to be residualized differently in different contexts, our residualization algorithm may be used.

7 Conclusion

We have presented a more efficient representation of configuration graphs for multi-result supercompilation and shown that this representation enables a more efficient residualization algorithm.

The idea of representing the space of configurations as a graph rather than a tree is quite obvious. The process of supercompilation can be viewed as some kind of graph search similar to finding paths in mazes, and it is natural to use the graph structure instead of unrolling it into a tree.

The overgraph representation also gives rise to a parallel to the field of optimizing compilers which manifests itself in the similarity of a configuration graph and a control flow graph. It is not yet obvious if this parallel is fruitful.

One of the closest work to this one seems to be the work on equality saturation [14]. One of the most important difference is that we need to support folding and thus we work with some directed relations rather than simple equalities to ensure correctness.

We believe that the new approach leads to many directions of reasearch and further improvement.

The residualization algorithm is still a bottleneck. There are many possible solutions. It may be useful to do without residualization if the task doesn't actually consist in producing a program, e.g. there should be some algorithm for proving programs equality which works directly on configuration graphs rather than on sets of residual programs. Another way is to reduce the number of nodes using some heuristics, especially interesting are methods that make possible to tune how long the supercompilation will run.

As it has been seen, there should be different residualization algorithms for different tasks. In the case of optimizing for size its goal is to extract a minimal supgraph. Apparently there may be some residualization algorithm of this sort

which would take advantage of the overgraph representation but it hasn't been found yet. Besides, it may be advantageous to represent programs as graphs.

It also seems interesting to apply the new approach to different domains. We also plan to add support for higher-level supercompilation [7] which may benefit from sharing information among multiple lower-level supercompilation instances.

Acknowledgements

The author would like to express his gratitude to Sergei Romanenko, Andrei Klimov and Ilya Klyuchnikov for their comments, fruitful discussions and encouragement.

References

1. D. G. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Commun. ACM*, 16:591–603, October 1973.
2. K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 34(7):1–9, July 1999.
3. G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin. MILEPOST GCC: machine learning based research compiler. In *GCC Summit*, Ottawa, Canada, 2008. MILEPOST project (http://www.milepost.eu).
4. G. W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70. ACM Press New York, NY, USA, 2007.
5. A. V. Klimov, I. G. Klyuchnikov, and S. A. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation. Preprint 19, Keldysh Institute of Applied Mathematics, 2012.
6. I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.
7. I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.
8. I. G. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute of Applied Mathematics, 2011.
9. I. G. Klyuchnikov and S. A. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In E. Clarke, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011*, volume 7162 of *Lecture Notes in Computer Science*, pages 210–226. Springer, 2012.
10. A. Lisitsa and M. Webster. Supercompilation for equivalence testing in metamorphic computer viruses detection. In *Proceedings of the First International Workshop on Metacomputation in Russia*, 2008.

11. Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO*, pages 319–332. IEEE Computer Society, 2006.
12. D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theor. Comput. Sci.*, 167(1-2):193–233, 1996.
13. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.
14. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.
15. V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

A Hierarchy of Program Transformers

G. W. Hamilton

School of Computing
Dublin City University
Dublin 9
Ireland
e-mail: *hamilton@computing.dcu.ie*

Abstract. In this paper, we describe a hierarchy of program transformers in which the transformer at each level of the hierarchy builds on top of the transformers at lower levels. The program transformer at the bottom of the hierarchy corresponds to positive supercompilation, and that at the next level corresponds to the first published definition of distillation [4]. We then show how the more recently published definition of distillation [5] can be described using this hierarchy. We see that this moves up through the levels of the transformation hierarchy until no further improvements can be made. The resulting definition of distillation uses only finite data structures, as opposed to the definition in [5], and we therefore argue that it is easier to understand and to implement.

1 Introduction

It is well known that programs written using functional programming languages often make use of intermediate data structures and this can be inefficient. Several program transformation techniques have been proposed to eliminate some of these intermediate data structures; for example *partial evaluation* [7], *deforestation* [17] and *supercompilation* [15]. *Positive supercompilation* [14] is a variant of Turchin's supercompilation that was introduced in an attempt to study and explain the essentials of Turchin's supercompiler. Although strictly more powerful than both partial evaluation and deforestation, Sørensen has shown that positive supercompilation (and hence also partial evaluation and deforestation) can only produce a linear speedup in programs [13].

The *distillation* algorithm was originally motivated by the need for automatic techniques for obtaining superlinear speedups in programs. The original definition of distillation [4] was very similar in its formulation to positive supercompilation; the main difference being that in positive supercompilation, generalization and folding are performed with respect to *expressions*, while in distillation, they are performed with respect to *graphs*. The graphs which were used in distillation for this purpose were in fact those produced by positive supercompilation, so we can see that this definition of distillation was built on top of positive supercompilation. This suggests the existence of a hierarchy of program transformers, where the transformer at each level is built on top of those at lower levels, and higher

level transformers are more powerful. In this paper, we define such a hierarchy inductively, with positive supercompilation as the base case at the bottom level, and each new level defined in terms of the previous ones. The original definition of distillation is therefore at the second level in this hierarchy, while the more recently published definition of distillation [5] does not actually belong to any single level of the hierarchy, but in fact moves up through the levels until no further improvements can be made. We also define this more recent version of distillation using our program transformer hierarchy.

The remainder of this paper is structured as follows. In Section 2, we define the syntax and semantics of the higher-order functional language on which the described transformations are performed. In Section 3, we define labelled transition systems, which are used to represent the results of transformations. In Section 4, we define the program transformer hierarchy, where the transformer at the bottom level corresponds to positive supercompilation, and each successive transformer is defined in terms of the previous ones. In Section 5, we describe the more recent definition of distillation using the program transformer hierarchy, and show how it moves up through the levels of this hierarchy until no further improvements can be made. Section 6 concludes and considers related work.

2 Language

In this section, we describe the higher-order functional language that will be used throughout this paper. It uses call-by-name evaluation.

Definition 1 (Language Syntax). The syntax of this language is as shown in Fig. 1.

$e ::= x$	Variable
$\mid c\ e_1 \ldots e_k$	Constructor Application
$\mid \lambda x.e$	λ-Abstraction
$\mid f$	Function Call
$\mid e_0\ e_1$	Application
$\mid \textbf{case}\ e_0\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$	Case Expression
$\mid \textbf{let}\ x = e_0\ \textbf{in}\ e_1$	Let Expression
$\mid e_0\ \textbf{where}\ f_1 = e_1 \ldots f_n = e_n$	Local Function Definitions
$p ::= c\ x_1 \ldots x_k$	Pattern

Fig. 1. Language Grammar

A program in the language is an expression which can be a variable, constructor application, λ-abstraction, function call, application, **case**, **let** or **where**. Variables introduced by λ-abstraction, **let** or **case** patterns are *bound*; all other

variables are *free*. We write $e_1 \equiv e_2$ if e_1 and e_2 differ only in the names of bound variables.

It is assumed that the input program contains no **let** expressions; these are only introduced during transformation. Each constructor has a fixed arity; for example *Nil* has arity 0 and *Cons* has arity 2. In an expression $c\ e_1 \ldots e_n$, n must equal the arity of c. The patterns in **case** expressions may not be nested. No variable may appear more than once within a pattern. We assume that the patterns in a **case** expression are non-overlapping and exhaustive. It is also assumed that erroneous terms such as $(c\ e_1 \ldots e_n)\ e$ where c is of arity n and **case** $(\lambda x.e)$ **of** $p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k$ cannot occur.

Example 1. An example program in our language which calculates the n^{th} fibonacci number is shown in Fig. 2.

```
fib n
where
fib  = λn. case n of
              Z    ⇒ S Z
            | S n' ⇒ case n' of
                       Z    ⇒ S Z
                     | S n'' ⇒ add (fib n'') (fib n')
add = λx.λy. case x of
               Z    ⇒ y
             | S x' ⇒ S (add x' y)
```

Fig. 2. Example Program

Definition 2 (Substitution). $\theta = \{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ denotes a *substitution*. If e is an expression, then $e\theta = e\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}$ is the result of simultaneously substituting the expressions $e_1, \ldots, e_n$ for the corresponding variables $x_1, \ldots, x_n$, respectively, in the expression e while ensuring that bound variables are renamed appropriately to avoid name capture.

Definition 3 (Renaming). $\sigma = \{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$, where σ is a bijective mapping, denotes a *renaming*. If e is an expression, then $e\{x_1 \mapsto x'_1, \ldots, x_n \mapsto x'_n\}$ is the result of simultaneously replacing the variables $x_1, \ldots, x_n$ with the corresponding variables $x'_1, \ldots, x'_n$, respectively, in the expression e.

Definition 4 (Shallow Reduction Context). A shallow reduction context $\mathcal{R}$ is an expression containing a single hole $\bullet$ in the place of the redex, which can have one of the two following possible forms:

$$\mathcal{R} ::= \bullet\ e \mid (\textbf{case}\ \bullet\ \textbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k)$$

Definition 5 (Evaluation Context). An evaluation context $\mathcal{E}$ is represented as a sequence of shallow reduction contexts (known as a *zipper* [6]), representing

the nesting of these contexts from innermost to outermost within which the expression redex is contained. An evaluation context can therefore have one of the two following possible forms:

$$\mathcal{E} ::= \langle\rangle \mid \langle \mathcal{R} : \mathcal{E} \rangle$$

Definition 6 (Insertion into Evaluation Context). The insertion of an expression e into an evaluation context κ, denoted by $\kappa \bullet e$, is defined as follows:

$$\begin{array}{ll}
\langle\rangle \bullet e & = e \\
\langle (\bullet\ e') : \kappa \rangle \bullet e & = \kappa \bullet (e\ e') \\
\langle (\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) : \kappa \rangle \bullet e & \\
\qquad & = \kappa \bullet (\mathbf{case}\ e\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k)
\end{array}$$

Free variables within the expression e may become bound within $\kappa \bullet e$; if $\kappa \bullet e$ is closed then we call κ a *closing context* for e.

Definition 7 (Unfolding). The unfolding of a function in the redex of expression e with function environment Δ is defined as shown in Fig. 3.

$$\mathcal{U}[\![e]\!]\ \Delta = \mathcal{U}'[\![e]\!]\ \langle\rangle\ \varnothing\ \Delta$$

$$\begin{array}{ll}
\mathcal{U}'[\![x]\!]\ \kappa\ \rho\ \Delta & = \begin{cases} \mathcal{U}'[\![\rho(x)]\!]\ \kappa\ \rho\ \Delta, & \text{if } x \in dom(\rho) \\ \kappa \bullet x, & \text{otherwise} \end{cases} \\
\mathcal{U}'[\![c\ e_1 \ldots e_k]\!]\ \kappa\ \rho\ \Delta & = \kappa \bullet (c\ e_1 \ldots e_k) \\
\mathcal{U}'[\![\lambda x.e]\!]\ \kappa\ \rho\ \Delta & = \kappa \bullet (\lambda x.e) \\
\mathcal{U}'[\![f]\!]\ \kappa\ \rho\ \Delta & = \kappa \bullet e \text{ where } (f = e) \in \Delta \\
\mathcal{U}'[\![e_0\ e_1]\!]\ \kappa\ \rho\ \Delta & = \mathcal{U}'[\![e_0]\!]\ \langle (\bullet\ e_1) : \kappa \rangle\ \rho\ \Delta \\
\mathcal{U}'[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \Delta & \\
\qquad & = \mathcal{U}'[\![e_0]\!]\ \langle (\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa \rangle\ \rho\ \Delta \\
\mathcal{U}'[\![\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \Delta & = \mathbf{let}\ x = e_0\ \mathbf{in}\ \mathcal{U}'[\![e_1]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \Delta \\
\mathcal{U}'[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \Delta & \\
\qquad & = \mathcal{U}'[\![e_0]\!]\ \kappa\ \rho\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\}) \\
\qquad & \quad \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n
\end{array}$$

Fig. 3. Function Unfolding

Within these rules, the context around the redex is built up within κ, the values of **let** variables are stored in ρ and the set of function definitions are stored in Δ. If the redex is a variable which has a value within ρ, then that value is substituted into the redex position. If the redex can itself be divided into an inner redex and shallow reduction context, then the shallow reduction context is added to the overall context and the inner redex is further unfolded. If the innermost redex is a function then it is replaced by its definition within Δ; otherwise this innermost redex is simply inserted back into its context.

The call-by-name operational semantics of our language is standard: we define an evaluation relation $\Downarrow$ between closed expressions and *values*, where values

are expressions in *weak head normal form* (i.e. constructor applications or λ-abstractions). We define a one-step reduction relation $\overset{r}{\rightsquigarrow}$ inductively as shown in Fig. 4, where the reduction r can be f (unfolding of function f), c (elimination of constructor c) or β (β-substitution).

$$((\lambda x.e_0)\ e_1) \overset{\beta}{\rightsquigarrow} (e_0\{x \mapsto e_1\}) \qquad (\textbf{let}\ x = e_0\ \textbf{in}\ e_1) \overset{\beta}{\rightsquigarrow} (e_1\{x \mapsto e_0\})$$

$$\frac{f = e}{f \overset{f}{\rightsquigarrow} e} \qquad \frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(e_0\ e_1) \overset{r}{\rightsquigarrow} (e_0'\ e_1)}$$

$$\frac{p_i = c\ x_1 \ldots x_n}{(\textbf{case}\ (c\ e_1 \ldots e_n)\ \textbf{of}\ p_1 : e_1' | \ldots | p_k : e_k') \overset{c}{\rightsquigarrow} (e_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\})}$$

$$\frac{e_0 \overset{r}{\rightsquigarrow} e_0'}{(\textbf{case}\ e_0\ \textbf{of}\ p_1 : e_1 | \ldots p_k : e_k) \overset{r}{\rightsquigarrow} (\textbf{case}\ e_0'\ \textbf{of}\ p_1 : e_1 | \ldots p_k : e_k)}$$

Fig. 4. One-Step Reduction Relation

We use the notation $e \overset{r}{\rightsquigarrow}$ if the expression e reduces, $e \Uparrow$ if e diverges, $e \Downarrow$ if e converges and $e \Downarrow v$ if e evaluates to the value v. These are defined as follows, where $\overset{r}{\rightsquigarrow}^*$ denotes the reflexive transitive closure of $\overset{r}{\rightsquigarrow}$:

$$e \overset{r}{\rightsquigarrow}, \text{ iff } \exists e'.e \overset{r}{\rightsquigarrow} e' \qquad e \Downarrow, \text{ iff } \exists v.e \Downarrow v$$
$$e \Downarrow v, \text{ iff } e \overset{r}{\rightsquigarrow}^* v \wedge \neg(v \overset{r}{\rightsquigarrow}) \qquad e \Uparrow, \text{ iff } \forall e'.e \overset{r}{\rightsquigarrow}^* e' \Rightarrow e' \overset{r}{\rightsquigarrow}$$

We assume that all expressions are typable under system F, and that types are *strictly positive*. This ensures that all infinite sequences of reductions must include the unfolding of a function.

Definition 8 (Observational Equivalence). Observational equivalence, denoted by $\simeq$, equates two expressions if and only if they exhibit the same termination behaviour in all closing contexts i.e. $e_1 \simeq e_2$ iff $\forall \kappa \bullet (\kappa \bullet e_1 \Downarrow \text{ iff } \kappa \bullet e_2 \Downarrow)$.

3 Labelled Transition Systems

In this section, we define the labelled transition systems (LTSs) which are used to represent the results of our transformations.

Definition 9 (Labelled Transition System). The LTS associated with program e is given by $t = (\mathcal{E}, e, \rightarrow, Act)$ where:

- $\mathcal{E}$ is the set of *states* of the LTS each of which is either an expression or the end-of-action state **0**.
- t always contains as root the expression e, denoted by $root(t) = e$.

- $\rightarrow \subseteq \mathcal{E} \times Act \times \mathcal{E}$ is a *transition relation* that relates pairs of states by actions according to Fig. 5. We write $e \rightarrow (\alpha_1, t_1), \ldots, (\alpha_n, t_n)$ for a LTS with root state e where $t_1 \ldots t_n$ are the LTSs obtained by following the transitions labelled $\alpha_1 \ldots \alpha_n$ respectively from e.
- If $e \in \mathcal{E}$ and $(e, \alpha, e') \in \rightarrow$ then $e' \in \mathcal{E}$.
- *Act* is a set of actions α that can be *silent* or *non-silent*. A non-silent action may be: x, a variable; c, a constructor; @, the function in an application; $\#i$, the i^{th} argument in an application; λx, an abstraction over variable x; **case**, a case selector; p, a case branch pattern; or **let**, an abstraction. A silent action may be: τ_f, unfolding of the function f; τ_c, elimination of the constructor c; or τ_β, β-substitution.

$$
\begin{array}{ll}
\mathcal{L}[\![x]\!]\ \rho\ \Delta & = x \rightarrow (x, \mathbf{0}) \\
\mathcal{L}[\![c\ e_1 \ldots e_n]\!]\ \rho\ \Delta & = (c\ e_1 \ldots e_n) \rightarrow (c, \mathbf{0}), (\#1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), \ldots, (\#n, \mathcal{L}[\![e_n]\!]\ \rho\ \Delta) \\
\mathcal{L}[\![\lambda x.e]\!]\ \rho\ \Delta & = (\lambda x.e) \rightarrow (\lambda x, \mathcal{L}[\![e]\!]\ \rho\ \Delta) \\
\mathcal{L}[\![f]\!]\ \rho\ \Delta & = \begin{cases} f \rightarrow (\tau_f, \mathbf{0}), & \text{if } f \in \rho \\ f \rightarrow (\tau_f, \mathcal{L}[\![e]\!]\ (\rho \cup \{f\})\ \Delta), & \text{otherwise where } (f = e) \in \Delta \end{cases} \\
\mathcal{L}[\![e_0\ e_1]\!]\ \rho\ \Delta & = (e_0\ e_1) \rightarrow (@, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta), (\#1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta) \\
\mathcal{L}[\![e = (\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k)]\!]\ \rho\ \Delta & \\
\quad = e \rightarrow (\mathbf{case}, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta), (p_1, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), \ldots, (p_k, \mathcal{L}[\![e_k]\!]\ \rho\ \Delta) & \\
\mathcal{L}[\![e = (\mathbf{let}\ x = e_0\ \mathbf{in}\ e_1)]\!]\ \rho\ \Delta & \\
\quad = e \rightarrow (\mathbf{let}, \mathcal{L}[\![e_1]\!]\ \rho\ \Delta), (x, \mathcal{L}[\![e_0]\!]\ \rho\ \Delta) & \\
\mathcal{L}[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \rho\ \Delta & \\
\quad = \mathcal{L}[\![e_0]\!]\ \rho\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\}) &
\end{array}
$$

Fig. 5. LTS Representation of a Program

Within the rules $\mathcal{L}$ shown in Fig. 5 for converting a program to a corresponding LTS, the parameter ρ is the set of previously encountered function calls and the parameter Δ is the set of function definitions. If a function call is re-encountered, no further transitions are added to the constructed LTS. Thus, the constructed LTS will always be a finite representation of the program.

Example 2. The LTS representation of the program in Fig. 2 is shown in Fig. 6.

Within the actions of a LTS, λ-abstracted variables, **case** pattern variables and **let** variables are *bound*; all other variables are *free*. We use $fv(t)$ and $bv(t)$ to denote the free and bound variables respectively of LTS t.

Definition 10 (Extraction of Residual Program from LTS). A residual program can be constructed from a LTS using the rules $\mathcal{R}$ as shown in Fig. 7. Within these rules, the parameter ε contains the set of new function calls that have been created, and associates them with the expressions they replaced. On encountering a renaming of a previously replaced expression, it is also replaced by the corresponding renaming of the associated function call.

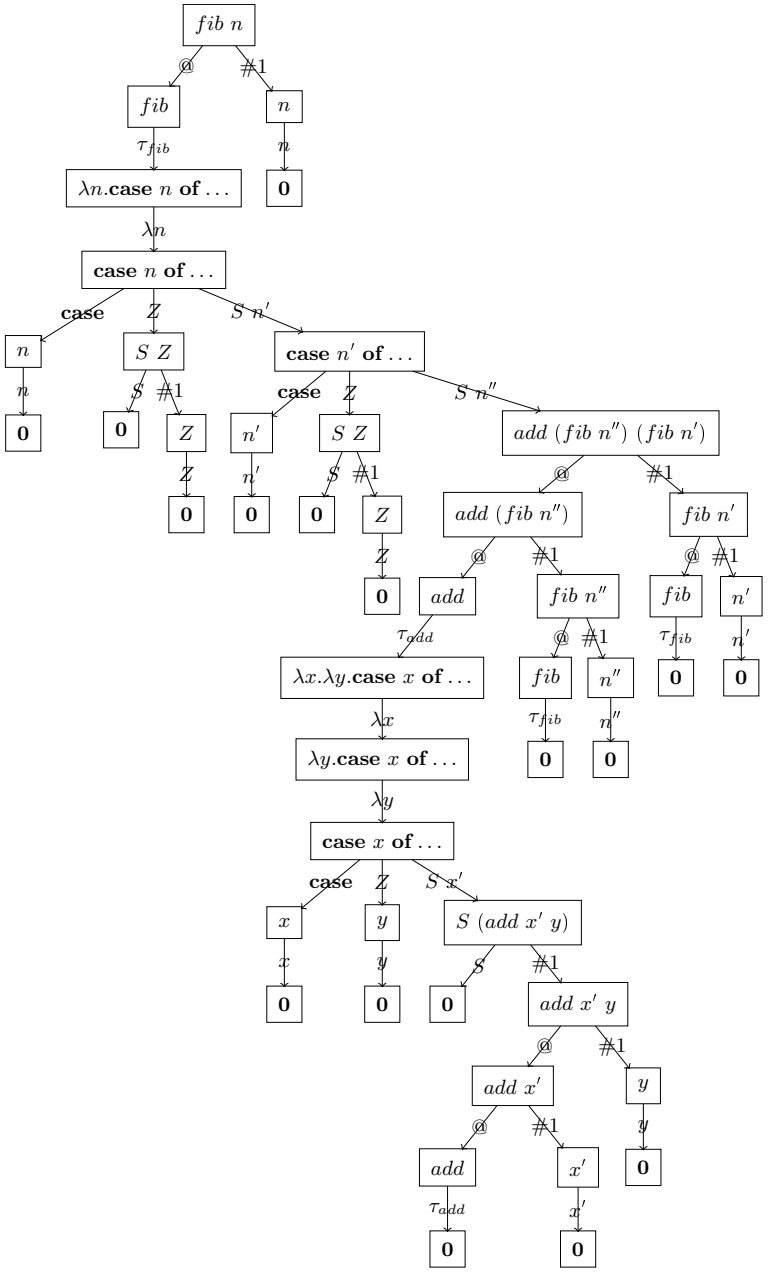


Fig. 6. LTS Corresponding to *fib n*

$$\mathcal{R}[\![t]\!] = \mathcal{R}'[\![t]\!]\ \varnothing$$

$$\begin{array}{l}
\mathcal{R}'[\![e \to (x, \mathbf{0})]\!]\ \varepsilon\ = x \\
\mathcal{R}'[\![e \to (c, \mathbf{0}), (\#1, t_1), \ldots, (\#n, t_n)]\!]\ \varepsilon \\
\qquad = c\ (\mathcal{R}'[\![t_1]\!]\ \varepsilon) \ldots (\mathcal{R}'[\![t_n]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (\lambda x, t)]\!]\ \varepsilon = \lambda x.(\mathcal{R}'[\![t]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (@, t_0), (\#1, t_1)]\!]\ \varepsilon \\
\qquad = (\mathcal{R}'[\![t_0]\!]\ \varepsilon)\ (\mathcal{R}'[\![t_1]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (\mathbf{case}, t_0)(p_1, t_1), \ldots, (p_n, t_k)]\!]\ \varepsilon \\
\qquad = \mathbf{case}\ (\mathcal{R}'[\![t_0]\!]\ \varepsilon)\ \mathbf{of}\ p_1 \Rightarrow (\mathcal{R}'[\![t_1]\!]\ \varepsilon)\ |\cdots|\ p_k \Rightarrow (\mathcal{R}'[\![t_k]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (\mathbf{let}, t_0), (x_1, t_1), \ldots, (x_n, t_n)]\!]\ \varepsilon \\
\qquad = \mathbf{let}\ x_1 = (\mathcal{R}'[\![t_1]\!]\ \varepsilon)\ \mathbf{in} \ldots \mathbf{let}\ x_n = (\mathcal{R}'[\![t_n]\!]\ \varepsilon)\ \mathbf{in}\ (\mathcal{R}'[\![t_0]\!]\ \varepsilon) \\
\mathcal{R}'[\![e \to (\tau_f, t)]\!]\ \varepsilon\ = \begin{cases} e'\theta, \text{ if } \exists (e' = e'') \in \varepsilon \bullet e \equiv e''\theta \\ f'\ x_1 \ldots x_n\ \mathbf{where}\ f' = \lambda x_1 \ldots x_n.(\mathcal{R}'[\![t]\!]\ (\varepsilon \cup \{f'\ x_1 \ldots x_n = e\})), \\ \qquad \text{otherwise } (f' \text{ is fresh}, \{x_1 \ldots x_n\} = fv(t)) \end{cases} \\
\mathcal{R}'[\![e \to (\tau_\beta, t)]\!]\ \varepsilon\ = \mathcal{R}'[\![t]\!]\ \varepsilon \\
\mathcal{R}'[\![e \to (\tau_c, t)]\!]\ \varepsilon\ = \mathcal{R}'[\![t]\!]\ \varepsilon
\end{array}$$

Fig. 7. Rules For Residualization

Example 3. The residual program constructed from the LTS in Fig. 6 is esentially that shown in Fig. 2 (modulo renaming of functions).

4 A Hierarchy of Program Transformers

In this section, we define a hierarchy of program transformers in which the transformer at each level of the hierarchy makes use of those at lower levels. Each transformer takes as its input the original program and produces as its output a labelled transition system, from which a new (hopefully improved) program can be residualized. In all the transformers, LTSs corresponding to previously encountered terms are compared to the LTS for the current term. If a *renaming* of a previously encountered LTS is detected, then *folding* is performed. If an *embedding* of a previously encountered LTS is detected, then *generalization* is performed. The use of LTSs rather than expressions when checking for renaming or embedding allows us to abstract away from the specific function names which are used within expressions and to focus on their underlying recursive structure.

Definition 11 (LTS Renaming). LTS t_1 is a *renaming* of LTS t_2 iff there is a renaming σ such that $t_1 \approx_\sigma^\varnothing t_2$, where the reflexive, transitive and symmetric relation $\approx_\sigma^\rho$ is defined as follows:

$$\begin{array}{l}
(x \to (x, \mathbf{0})) \approx_\sigma^\rho (x' \to (x', \mathbf{0})), \quad \text{if } x\sigma = x' \\
(e \to (\tau_f, t)) \approx_\sigma^\rho (e' \to (\tau_{f'}, t')), \text{ if } ((f, f') \in \rho) \vee (t \approx_\sigma^{\rho \cup \{(f,f')\}} t') \\
(e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) \approx_\sigma^\rho (e' \to (\alpha'_1, t'_1), \ldots, (\alpha'_n, t'_n)), \\
\qquad\qquad \text{if } \forall i \in \{1 \ldots n\} \bullet (\exists \sigma' \bullet (\alpha_i\ \sigma' = \alpha'_i \wedge t_i \approx_{\sigma \cup \sigma'}^\rho t'_i))
\end{array}$$

The rules for renaming are applied in top-down order, where the final rule is a catch-all. Two LTSs are renamings of each other if the same transitions are possible from each corresponding state (modulo variable renaming according to the renaming σ). The definition also handles transitions that introduce bound variables (λ, **case** and **let**); in these cases the corresponding bound variables are added to the renaming σ. The parameter ρ is used to keep track of the corresponding function names which have been matched with each other.

Definition 12 (LTS Embedding). LTS t_1 is *embedded* within LTS t_2 iff there is a renaming σ such that $t_1 \lesssim_\sigma^\emptyset t_2$, where the reflexive, transitive and anti-symmetric relation $\lesssim_\sigma^\rho$ is defined as follows:

$$
\begin{array}{ll}
t \lesssim_\sigma^\rho t', & \text{if } (t \lhd_\sigma^\rho t') \vee (t \bowtie_\sigma^\rho t') \\
t \lhd_\sigma^\rho (e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)), & \text{if } \exists i \in \{1 \ldots n\} \bullet t \lesssim_\sigma^\rho t_i \\
(x \to (x, \mathbf{0})) \bowtie_\sigma^\rho (x' \to (x', \mathbf{0})), & \text{if } x\sigma = x' \\
(e \to (\tau_f, t)) \bowtie_\sigma^\rho (e' \to (\tau_{f'}, t')), & \text{if } ((f, f') \in \rho) \vee (t \lesssim_\sigma^{\rho \cup \{(f,f')\}} t') \\
(e \to (\alpha_1, t_1), \ldots, (\alpha_n, t_n)) \bowtie_\sigma^\rho (e' \to (\alpha_1', t_1'), \ldots, (\alpha_n', t_n')), & \\
\qquad \text{if } \forall i \in \{1 \ldots n\} \bullet (\exists \sigma' \bullet (\alpha_i\ \sigma' = \alpha_i' \wedge t_i \lesssim_{\sigma \cup \sigma'}^\rho t_i')) &
\end{array}
$$

The rules for embedding are applied in top-down order, where the final rule is a catch-all. One LTS is embedded within another by this relation if either *diving* (denoted by $\lhd_\sigma^\rho$) or *coupling* (denoted by $\bowtie_\sigma^\rho$) can be performed. In the rules for diving, a transition can be followed from the current state in the embedding LTS that is not followed from the current state in the embedded one. In the rules for coupling, the same transitions are possible from each of the current states. Matching transitions may contain different free variables; in this case the transition labels should respect the renaming σ. Matching transitions may also introduce bound variables (λ, **case** and **let**); in these cases the corresponding bound variables are added to the renaming σ. The parameter ρ is used to keep track of the corresponding function names which have been coupled with each other.

Definition 13 (Generalization of LTSs). The function $\mathcal{G}[\![t]\!][\![t']\!]\ \theta$ that generalizes LTS t with respect to LTS t' is defined in Fig. 8, where θ is the set of previous generalizations which can be reused. The result of this function is the generalization of the LTS t, in which some sub-components have been extracted from t using **let**s.

Within the rules $\mathcal{G}'$, γ is the set of bound variables within the LTS being generalized and ρ is used to keep track of the corresponding function names which have been matched with each other. The rules are applied in top-down order. If two corresponding states have the same transitions, these transitions remain in the resulting generalized LTS, and the corresponding targets of these transitions are then generalized. Matching transitions may introduce bound variables (λ, **case** and **let**); in these cases the bound variables are added to the set of bound variables γ. Unmatched LTS components are extracted into a substitution and replaced by variable applications. The arguments of the variable applications

$$
\begin{array}{l}
\mathcal{G}[\![t]\!][\![t']\!]\ \theta = \mathcal{A}[\![t^g]\!]\ \theta' \\
\text{where} \\
(t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \varnothing\ \varnothing
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{G}'[\![e \to (x,\mathbf{0})]\!][\![e' \to (x',\mathbf{0})]\!]\ \theta\ \gamma\ \rho & = ((e \to (x,\mathbf{0})), \varnothing) \\
\mathcal{G}'[\![e \to (c,\mathbf{0}), (\#1,t_1), \ldots, (\#n,t_n)]\!][\![e' \to (c,\mathbf{0}), (\#1,t'_1), \ldots, (\#n,t'_n)]\!]\ \theta\ \gamma\ \rho & \\
\qquad = ((e \to (c,\mathbf{0}), (\#1,t^g_1), \ldots, (\#n,t^g_n)), \bigcup_{i=1}^{n} \theta_i) & \\
\qquad \text{where} & \\
\qquad \forall i \in \{1 \ldots n\} \bullet (t^g_i, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t'_i]\!]\ \theta\ \gamma\ \rho & \\
\mathcal{G}'[\![e \to (\lambda x,t)]\!][\![e' \to (\lambda x',t')]\!]\ \theta\ \gamma\ \rho & = ((e \to (\lambda x, t^g)), \theta') \\
\qquad \text{where} & \\
\qquad (t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ (\gamma \cup \{x\})\ \rho & \\
\mathcal{G}'[\![e \to (@,t_0), (\#1,t_1)]\!][\![e' \to (@,t'_0), (\#1,t'_1)]\!]\ \theta\ \gamma\ \rho & \\
\qquad = ((e \to (@,t^g_0), (\#1,t^g_1)), \theta_0 \cup \theta_1) & \\
\qquad \text{where} & \\
\qquad \forall i \in \{0,1\} \bullet (t^g_i, \theta_i) = \mathcal{G}[\![t_i]\!][\![t'_i]\!]\ \theta\ \gamma\ \rho & \\
\mathcal{G}'[\![e \to (\mathbf{case},t_0), (p_1,t_1), \ldots, (p_n,t_n)]\!][\![e' \to (\mathbf{case},t'_0), (p'_1,t'_1), \ldots, (p'_n,t'_n)]\!]\ \theta\ \gamma\ \rho & \\
\qquad = ((e \to (\mathbf{case},t^g_0), (p_1,t^g_1), \ldots, (p_n,t^g_n)), \bigcup_{i=0}^{n} \theta_i) & \\
\qquad \text{where} & \\
\qquad \forall i \in \{1 \ldots n\} \bullet (\exists \sigma \bullet p_i \equiv p'_i\sigma) & \\
\qquad (t^g_0, \theta_0) = \mathcal{G}'[\![t_0]\!][\![t'_0]\!]\ \theta\ \gamma\ \rho & \\
\qquad \forall i \in \{1 \ldots n\} \bullet (t^g_i, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t'_i]\!]\ \theta\ (\gamma \cup fv(p_i))\ \rho & \\
\mathcal{G}'[\![e \to (\mathbf{let},t_0), (x_1,t_1), \ldots, (x_n,t_n)]\!][\![e' \to (\mathbf{let},t'_0), (x'_1,t'_1), \ldots, (x'_n,t'_n)]\!]\ \theta\ \gamma\ \rho & \\
\qquad = ((e \to (\mathbf{let},t^g_0), (x_1,t^g_1), \ldots, (x_n,t^g_n)), \bigcup_{i=0}^{n} \theta_i) & \\
\qquad \text{where} & \\
\qquad (t^g_0, \theta_0) = \mathcal{G}'[\![t_0]\!][\![t'_0]\!]\ \theta\ \gamma\ \rho & \\
\qquad \forall i \in \{1 \ldots n\} \bullet (t^g_i, \theta_i) = \mathcal{G}'[\![t_i]\!][\![t'_i]\!]\ \theta\ \gamma\ \rho & \\
\mathcal{G}'[\![e \to (\tau_f,t)]\!][\![e' \to (\tau_{f'},t')]\!]\ \theta\ \gamma\ \rho & = \begin{cases} ((e \to (\tau_f,t)), \varnothing), & \text{if } (f,f') \in \rho \\ ((e \to (\tau_f,t^g)), \theta'), & \text{otherwise} \\ \quad \text{where} & \\ \quad (t^g, \theta') = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ (\rho \cup \{(f,f')\}) \end{cases} \\
\mathcal{G}'[\![e \to (\tau_\beta,t)]\!][\![e' \to (\tau_\beta,t')]\!]\ \theta\ \gamma\ \rho & = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho \\
\mathcal{G}'[\![e \to (\tau_c,t)]\!][\![e' \to (\tau_c,t')]\!]\ \theta\ \gamma\ \rho & = \mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho \\
\mathcal{G}'[\![t]\!][\![t']\!]\ \theta\ \gamma\ \rho & = \begin{cases} (\mathcal{B}[\![x \to (x,\mathbf{0})]\!]\ \gamma', \varnothing), & \text{if } \exists (x,t_1) \in \theta \bullet t_1 \approx^{\varnothing}_{\varnothing} t_2 \\ (\mathcal{B}[\![x \to (x,\mathbf{0})]\!]\ \gamma', \{x \mapsto t_2\}), & \text{otherwise } (x \text{ is fresh}) \end{cases} \\
\qquad \text{where} & \\
\qquad \gamma' = fv(t) \cap \gamma & \\
\qquad t_2 = \mathcal{C}[\![t]\!]\ \gamma'
\end{array}
$$

$$
\mathcal{A}[\![t]\!]\ \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\} = root(t) \to (\mathbf{let}, t), (x_1,t_1), \ldots, (x_n,t_n)
$$

$$
\mathcal{B}[\![t]\!]\ \{x_1 \ldots x_n\} = root(t) \to (@, (\ldots root(t) \to (@,t), (\#1, x_1 \to (x_1,\mathbf{0})) \ldots)), (\#1, x_n \to (x_n,\mathbf{0}))
$$

$$
\mathcal{C}[\![t]\!]\ \{x_1 \ldots x_n\} = root(t) \to (\lambda x_1, \ldots root(t) \to (\lambda x_n, t) \ldots)
$$

Fig. 8. Rules for Generalization

introduced are the free variables of the LTS component which are also contained in the set of overall bound variables γ; this ensures that bound variables are not extracted outside their binders. If an extracted LTS component is contained in the set of previous generalizations θ, then the variable name from this pre-

vious generalization is reused. Otherwise, a new variable is introduced which is different and distinct from all other program variables.

4.1 Level 0 Transformer

We now define the level 0 program transformer within our hierarchy, which corresponds closely to the positive supercompilation algorithm. The transformer takes as its input the original program and produces as its output a labelled transition system, from which a new (hopefully improved) program can be residualized. The level 0 transformer effectively performs a normal-order reduction of the input program. The (LTS representation of) previously encountered terms are 'memoized'. If the (LTS representation of the) current term is a renaming of a memoized one, then folding is performed, and the transformation is complete.

$$
\begin{array}{ll}
\mathcal{T}_0[\![x]\!]\ \kappa\ \rho\ \theta\ \Delta & = \mathcal{T}_0'[\![x \rightarrow (x, \mathbf{0})]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}_0[\![e = c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta & \\
\qquad = e \rightarrow (c, \mathbf{0}), (\#1, \mathcal{T}_0[\![e_1]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{T}_0[\![e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) & \\
\mathcal{T}_0[\![e = c\ e_1 \ldots e_n]\!]\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') : \kappa'\rangle)\ \rho\ \theta\ \Delta & \\
\qquad = (\kappa \bullet e) \rightarrow (\tau_c, \mathcal{T}_0[\![e_i'\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) & \\
\qquad \text{where } p_i = c\ x_1 \ldots x_n & \\
\mathcal{T}_0[\![e = \lambda x.e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta & \\
\qquad = e \rightarrow (\lambda x, \mathcal{T}_0[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) & \\
\mathcal{T}_0[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta & \\
\qquad = (\kappa \bullet e) \rightarrow (\tau_\beta, \mathcal{T}_0[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) & \\
\mathcal{T}_0[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta & = \begin{cases} (\kappa \bullet f) \rightarrow (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \approx_\sigma^\varnothing t \\ \mathcal{T}_0[\![\mathcal{R}[\![\mathcal{G}[\![t]\!][\![t']\!]\ \theta]\!]]\!]\ \langle\rangle\ \rho\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie_\sigma^\varnothing t \\ (\kappa \bullet f) \rightarrow (\tau_f, \mathcal{T}_0[\![\mathcal{U}[\![\kappa \bullet f]\!]\ \Delta]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \Delta), & \text{otherwise} \end{cases} \\
\qquad \text{where } t = \mathcal{L}[\![\kappa \bullet f]\!]\ \varnothing\ \Delta & \\
\mathcal{T}_0[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta = \mathcal{T}_0[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta & \\
\mathcal{T}_0[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \theta\ \Delta & \\
\qquad = \mathcal{T}_0[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta & \\
\mathcal{T}_0[\![e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta & \\
\qquad = (\kappa \bullet e) \rightarrow (\mathbf{let}, \mathcal{T}_0[\![e_1]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{T}_0[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) & \\
\mathcal{T}_0[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \theta\ \Delta & \\
\qquad = \mathcal{T}_0[\![e_0]\!]\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\}) &
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{T}_0'[\![t]\!]\ \langle\rangle\ \rho\ \theta\ \Delta & = t \\
\mathcal{T}_0'[\![t]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta & \\
\qquad = \mathcal{T}_0'[\![(t\ e) \rightarrow (@, t), (\#1, \mathcal{T}_0[\![e]\!]\ \langle\rangle\ \rho\ \theta\ \Delta)]\!]\ \kappa\ \rho\ \theta\ \Delta & \\
\mathcal{T}_0'[\![x \rightarrow (x, \mathbf{0})]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') : \kappa\rangle\ \rho\ \theta\ \Delta & \\
\qquad = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') \rightarrow & \\
\qquad (\mathbf{case}, t), (p_1, \mathcal{T}_0[\![(\kappa \bullet e_1')\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_0[\![(\kappa \bullet e_k')\{x \mapsto p_k\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) & \\
\mathcal{T}_0'[\![t]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') : \kappa\rangle\ \rho\ \theta\ \Delta & \\
\qquad = (\mathbf{case}\ (root(t))\ \mathbf{of}\ p_1 \Rightarrow e_1' \mid \cdots \mid p_k \Rightarrow e_k') \rightarrow & \\
\qquad\qquad (\mathbf{case}, t), (p_1, \mathcal{T}_0[\![e_1']\!]\ \kappa\ \rho\ \theta\ ), \ldots, (p_k, \mathcal{T}_0[\![e_k']\!]\ \kappa\ \rho\ \theta\ \Delta) &
\end{array}
$$

Fig. 9. Level 0 Transformation Rules

If the (LTS representation of the) current term is an embedding of a memoized one, then generalization is performed, and the resulting generalized term is further transformed. Generalization ensures that a renaming of a previously encountered term is always eventually encountered, and that the transformation therefore terminates. The rules for level 0 transformation are as shown in Fig. 9.

The rules $\mathcal{T}_0$ are defined on an expression and its surrounding context, denoted by κ. The parameter ρ contains the LTS representations of memoized terms; for our language it is only necessary to add LTSs to ρ for terms in which the redex is a function, as any infinite sequence of reductions must include such terms. The parameter θ contains terms which have been extracted using a **let** expression as a result of generalization. If an identical term is subsequently extracted as a result of further generalization, then the extraction is removed and the same variable is used as for the previous extraction. The parameter Δ contains the set of function definitions.

The rules $\mathcal{T}_0'$ are defined on an LTS and its surrounding context, also denoted by κ. These rules are applied when the normal-order reduction of the input program becomes 'stuck' as a result of encountering a variable in the redex position. In this case, the context surrounding the redex is further transformed. If the context surrounding a variable redex is a **case**, then information is propagated to each branch of the **case** to indicate that this variable has the value of the corresponding branch pattern.

Example 4. The result of transforming the *fib* program in Fig. 2 using the level 0 transformer is shown in Fig. 10 (due to space constraints, we present the results of transformation in this and further examples as residualized programs rather than LTSs). As we can see, this is no real improvement over the original program.

$$
\begin{array}{l}
f\ n \\
\textbf{where} \\
f = \lambda n.\,\textbf{case}\ n\ \textbf{of} \\
\qquad\qquad Z \quad \Rightarrow S\ Z \\
\qquad\quad \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of} \\
\qquad\qquad\qquad\qquad Z \quad \Rightarrow S\ Z \\
\qquad\qquad\qquad \mid S\ n'' \Rightarrow \textbf{case}\ (f\ n'')\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Z \quad \Rightarrow f\ (S\ n'') \\
\qquad\qquad\qquad\qquad\qquad \mid S\ x \Rightarrow S\ (g\ x\ n'') \\
g = \lambda x.\lambda n.\,\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\quad Z \quad \Rightarrow f\ (S\ n) \\
\qquad\qquad \mid S\ x' \Rightarrow S\ (g\ x'\ n)
\end{array}
$$

Fig. 10. Result of Level 0 Transformation

4.2 Level $n+1$ Transformer

We now define the transformers at all levels above 0 within our hierarchy. Each of these transformers makes use of the transformers below it in the hierarchy. The rules for a level $n+1$ transformer are actually very similar to those for the level 0 transformer; the level $n+1$ transformer also takes as its input the original program, performs normal-order reduction on it, and produces as its output a labelled transition system. Where the level $n+1$ transformer differs from that at level 0 is that the LTSs, which are memoized for the purposes of comparison when determining whether to fold or generalize, are those resulting from the level n transformation of previously encountered terms.

After the LTS resulting from level n transformation has been memoized, it is residualized, unfolded and further transformed. If a renaming of a memoized LTS is encountered, then folding is performed. If an embedding of a memoized LTS is encountered, then generalization is performed; this generalization will have the effect of adding an extra layer of **let**s around the LTS. Thus, each successive

$$
\begin{array}{l}
\mathcal{T}_{n+1}[\![x]\!]\ \kappa\ \rho\ \theta\ \Delta = \mathcal{T}'_{n+1}[\![x \to (x, \mathbf{0})]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e = c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \to (c, \mathbf{0}), (\#1, \mathcal{T}_{n+1}[\![e_1]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{T}_{n+1}[\![e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e = c\ e_1 \ldots e_n]\!]\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\tau_c, \mathcal{T}_{n+1}[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\qquad\quad \text{where } p_i = c\ x_1 \ldots x_n \\
\mathcal{T}_{n+1}[\![e = \lambda x.e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \to (\lambda x, \mathcal{T}_{n+1}[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\tau_\beta, \mathcal{T}_{n+1}[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta = \begin{cases} (\kappa \bullet f) \to (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \approx^{\mathscr{D}}_{\sigma} t \\ \mathcal{T}_{n+1}[\![\mathcal{R}[\![\mathcal{G}[\![t]\!][\![t']\!]\ \theta]\!]]\!]\ \langle\rangle\ \rho\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie^{\mathscr{D}}_{\sigma} t \\ (\kappa \bullet f) \to (\tau_f, \mathcal{T}_{n+1}[\![\mathcal{U}[\![\mathcal{R}[\![t]\!]]\!]\ \varnothing]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \varnothing), & \text{otherwise} \end{cases} \\
\qquad\quad \text{where } t = \mathcal{T}_n[\![f]\!]\ \kappa\ \varnothing\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \cdots \mid p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{T}_{n+1}[\![e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \to (\mathbf{let}, \mathcal{T}_{n+1}[\![e_1\{x \mapsto e_0\}]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{T}_{n+1}[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}_{n+1}[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}_{n+1}[\![e_0]\!]\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\})
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}'_{n+1}[\![t]\!]\ \langle\rangle\ \rho\ \theta\ \Delta = t \\
\mathcal{T}'_{n+1}[\![t]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{T}'_{n+1}[\![(t\ e) \to (@, t), (\#1, \mathcal{T}_{n+1}[\![e]\!]\ \langle\rangle\ \rho\ \theta\ )]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{T}'_{n+1}[\![x \to (x, \mathbf{n+1})]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \to \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{T}_{n+1}[\![(\kappa \bullet e'_1)\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_{n+1}[\![(\kappa \bullet e'_k)\{x \mapsto p_k\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{T}'_{n+1}[\![t]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ root(t)\ \mathbf{of}\ p_1 \Rightarrow e'_1 \mid \cdots \mid p_k \Rightarrow e'_k) \to \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{T}_{n+1}[\![e'_1]\!]\ \kappa\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{T}_{n+1}[\![e'_k]\!]\ \kappa\ \rho\ \theta\ \Delta)
\end{array}
$$

Fig. 11. Level $n+1$ Transformation Rules

level in the transformer hierarchy will have the effect of adding an extra layer of **let**s around the LTS representation of the current term.

As over-generalization may occur at each level in the transformer hierarchy, the extracted terms at each level are substituted back in at the next successive level. The **let**s themselves are retained as the extracted terms could be re-encountered in further generalization, and the **let** variables reused; if they are not reused, then the **let** can be removed in a post-processing phase. Note that at each successive level in the transformer hierarchy, the size of the components which are extracted by generalization must get smaller as they are sub-components of the result of transformation at the previous level.

The level $n + 1$ transformation rules are defined as shown in Fig. 11. The parameters used within these rules are the same as those for the level 0 transformer, except that the memoization environment ρ contains the LTSs resulting from the level n transformation of previously encountered terms.

Example 5. If the result of the level 0 transformation of the *fib* program shown in Fig. 10 is further transformed within a level 1 transformer, then the level 0 result as shown in Fig. 12 is encountered.

$$
\begin{array}{l}
f\ n \\
\textbf{where} \\
f = \lambda n.\,\textbf{case}\ n\ \textbf{of} \\
\qquad\qquad Z \quad \Rightarrow S\ S\ Z \\
\qquad\quad \mid S\ n' \Rightarrow \textbf{case}\ n'\ \textbf{of} \\
\qquad\qquad\qquad\qquad Z \quad \Rightarrow S\ S\ S\ Z \\
\qquad\qquad\qquad \mid S\ n'' \Rightarrow \textbf{case}\ (f\ n'')\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad Z \quad \Rightarrow f\ (S\ n'') \\
\qquad\qquad\qquad\qquad\qquad \mid S\ x \Rightarrow S\ (g\ x\ n'') \\
g = \lambda x.\lambda n.\,\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\quad Z \quad \Rightarrow f\ (S\ n) \\
\qquad\qquad \mid S\ x' \Rightarrow S\ (g\ x'\ n)
\end{array}
$$

Fig. 12. Further Result of Level 0 Transformation

Generalization is therefore performed with respect to the previous level 0 result in Fig. 10 to obtain the level 1 result shown in Fig. 13. We can see that an extra layer of **let**s has been added around this program. If this level 1 program is further transformed within a level 2 transformer, then the level 1 result as shown in Fig. 14 is encountered.

Generalization is therefore performed with respect to the previous level 1 result in Fig. 13 to obtain the level 2 result shown in Fig. 15. Again, we can see that an extra layer of **let**s has been added around this program. If this level 2 program is further transformed within a level 3 transformer, then the level 2 result as shown in Fig. 16 is encountered. Generalization is therefore performed with respect to the previous level 2 result in Fig. 15 to obtain the

```
let x1 = S Z
in let x2 = S S Z
   in f n
      where
      f = λn. case n of
                  Z    ⇒ S x1
                | S n' ⇒ case n' of
                             Z    ⇒ S x2
                           | S n'' ⇒ case (f n'') of
                                         Z   ⇒ f (S n'')
                                       | S x ⇒ S (g x n'')
      g = λx.λn. case x of
                     Z    ⇒ f (S n)
                   | S x' ⇒ S (g x' n)
```

Fig. 13. Result of Level 1 Transformation

```
let x3 = S S S Z
in let x4 = S S S S S Z
   in f n
      where
      f = λn. case n of
                  Z    ⇒ S S x3
                | S n' ⇒ case n' of
                             Z    ⇒ S S S x4
                           | S n'' ⇒ case (f n'') of
                                         Z   ⇒ f (S n'')
                                       | S x ⇒ S (g x n'')
      g = λx.λn. case x of
                     Z    ⇒ f (S n)
                   | S x' ⇒ S (g x' n)
```

Fig. 14. Further Result of Level 1 Transformation

level 3 result shown in Fig. 17. In this case, we can see that an extra layer of **let**s has not been added around the program; this is because the components which would have been extracted had been extracted previously, so the variables from these previous extractions were reused. If this level 3 program is further transformed within a level 4 transformer, then the level 3 result as shown in Fig. 18 is encountered.

We can now see that the level 3 result in Fig. 18 is a renaming of the level 3 result in Fig. 17. Folding is therefore performed to obtain the result in Fig. 19 (for the sake of brevity, this program has been compressed, but the actual result can be obtained from this by performing a couple of function unfoldings).

We can see that the original program which contained double recursion has been transformed into one with single recursion. Note that the result we have

```
let x5 = λx.S x
in let x6 = λx.S S x
   in let x3 = S x2
      in let x4 = S S x3
         in f n
            where
            f = λn. case n of
                      Z    ⇒ S (x5 x3)
                    | S n' ⇒ case n' of
                               Z    ⇒ S (x6 x4)
                             | S n'' ⇒ case (f n'') of
                                         Z    ⇒ f (S n'')
                                       | S x ⇒ S (g x n'')
            g = λx.λn. case x of
                         Z    ⇒ f (S n)
                       | S x' ⇒ S (g x' n)
```

Fig. 15. Result of Level 2 Transformation

```
let x9 = λx.S S S x
in let x10 = λx.S S S S S x
   in let x7 = S S S x4
      in let x8 = S S S S S x7
         in f n
            where
            f = λn. case n of
                      Z    ⇒ S (x9 x7)
                    | S n' ⇒ case n' of
                               Z    ⇒ S (x10 x8)
                             | S n'' ⇒ case (f n'') of
                                         Z    ⇒ f (S n'')
                                       | S x ⇒ S (g x n'')
            g = λx.λn. case x of
                         Z    ⇒ f (S n)
                       | S x' ⇒ S (g x' n)
```

Fig. 16. Further Result of Level 2 Transformation

obtained is not the linear version of the *fib* function which we might have expected. This is because the addition operation within the original program is not a constant-time operation; it is a linear-time operation defined on Peano numbers. If we had used a constant-time addition operator, then we would have obtained the linear version of the *fib* function using transformers at level 1 and upwards. This requires building the addition operator into our language, and defining specific transformation rules for such built-in operators which transform their arguments in sequence.

```
let x9 = λx.x5 (x6 x)
in let x10 = λx.x6 (x5 (x6 x))
   in let x7 = x5 (x6 x4)
      in let x8 = x6 (x5 (x6 x4))
         in f n
            where
            f = λn. case n of
                      Z    ⇒ S (x9 x7)
                    | S n' ⇒ case n' of
                               Z    ⇒ S (x10 x8)
                             | S n'' ⇒ case (f n'') of
                                         Z    ⇒ f (S n'')
                                       | S x ⇒ S (g x n'')
            g = λx.λn. case x of
                         Z    ⇒ f (S n)
                       | S x' ⇒ S (g x' n)
```

Fig. 17. Result of Level 3 Transformation

```
let x13 = λx.x9 (x10 x)
in let x14 = λx.x10 (x9 (x10 x))
   in let x11 = x9 (x10 x8)
      in let x12 = x10 (x9 (x10 x8))
         in f n
            where
            f = λn. case n of
                      Z    ⇒ S (x13 x11)
                    | S n' ⇒ case n' of
                               Z    ⇒ S (x14 x12)
                             | S n'' ⇒ case (f n'') of
                                         Z    ⇒ f (S n'')
                                       | S x ⇒ S (g x n'')
            g = λx.λn. case x of
                         Z    ⇒ f (S n)
                       | S x' ⇒ S (g x' n)
```

Fig. 18. Further Result of Level 3 Transformation

5 Distillation

In this section, we show how distillation can be described within our hierarchy of program transformers. One difficulty with having such a hierarchy of transformers is knowing which level within the hierarchy is sufficient to obtain the desired results. For example, for many of the examples in the literature of programs which are improved by positive supercompilation (level 0 in our hierarchy), no further improvements are obtained at higher levels in the hierarchy. However, only linear improvements in efficiency are possible at this level [13]. Higher levels

```
case n of
    Z    ⇒ S Z
  | S n' ⇒ case n' of
               Z    ⇒ S Z
             | S n'' ⇒ f n'' (λx.S x) (λx.S S x) Z Z
where
f = λn.λg.λh.λx.λy. case n of
                        Z    ⇒ S (g x)
                      | S n' ⇒ case n' of
                                   Z    ⇒ S(h y)
                                 | S n'' ⇒ f n'' (λx.g (h x)) (λx.h (g (h x))) (g x) (h y)
```

Fig. 19. Overall Result of Level 4 Transformation

in the hierarchy are capable of obtaining super-linear improvements in efficiency, but are overkill in many cases.

We therefore give a formulation of distillation which initially performs at level 0 in our hierarchy, and only moves to higher levels when necessary. Moving to a higher level in the hierarchy is only necessary if generalization has to be performed at the current level. Thus, when generalization is performed, the result of the generalization is memoized and is used for comparisons at the next level when checking for renamings or embeddings.

The rules for distillation are shown in Fig. 20. These rules are very similar to those for the transformers within the hierarchy; they take the original program as input, perform normal-order reduction, and produce a labelled transition system as output. The rules differ from those for the transformers within the hierarchy in that when generalization has to be performed, the LTS resulting from generalization at the current level is memoized, residualized, unfolded and then transformed at the next level up in the transformer hierarchy. Thus, each generalization will have the effect of moving up to the next level in the transformer hierarchy in addition to adding an extra layer of **let**s around the LTS representation of the current term.

We have already seen that at each successive level in the transformer hierarchy, the size of the components which are extracted by generalization must get smaller as they are sub-components of the result of transformation at the previous level. A point must therefore always be reached eventually at which extracted components re-occur, and the same generalization variables will be reused without introducing new **let**s. At this point, no further generalization will be done and the distiller will not move any further up the transformer hierarchy and must terminate at the current level.

Example 6. The result of transforming the *fib* program in Fig. 2 using distillation is the same as that of the level 4 transformer within our transformation hierarchy shown in Fig. 19.

$$
\begin{array}{l}
\mathcal{D}_n[\![x]\!]\ \kappa\ \rho\ \theta\ \Delta = \mathcal{D}'_n[\![x \rightarrow (x, \mathbf{0})]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{D}_n[\![e = c\ e_1 \ldots e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \rightarrow (c, \mathbf{0}), (\#1, \mathcal{D}_n[\![e_1]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (\#n, \mathcal{D}_n[\![e_n]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n[\![e = c\ e_1 \ldots e_n]\!]\ (\kappa = \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1\ |\cdots|\ p_k \Rightarrow e'_k) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \rightarrow (\tau_c, \mathcal{D}_n[\![e'_i\{x_1 \mapsto e_1, \ldots, x_n \mapsto e_n\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\qquad\quad \text{where } p_i = c\ x_1 \ldots x_n \\
\mathcal{D}_n[\![e = \lambda x.e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta \\
\qquad = e \rightarrow (\lambda x, \mathcal{D}_n[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n[\![e = \lambda x.e_0]\!]\ (\kappa = \langle(\bullet\ e_1) : \kappa'\rangle)\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \rightarrow (\tau_\beta, \mathcal{D}_n[\![e_0\{x \mapsto e_1\}]\!]\ \kappa'\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n[\![f]\!]\ \kappa\ \rho\ \theta\ \Delta = \left\{ \begin{array}{ll}
(\kappa \bullet f) \rightarrow (\tau_f, \mathbf{0}), & \text{if } \exists t' \in \rho, \sigma \bullet t' \approx^{\varnothing}_{\sigma} t \\
\mathcal{D}_{n+1}[\![\mathcal{U}[\![\mathcal{R}[\![t^g]\!]]\!]\ \varnothing]\!]\ \langle\rangle\ \{t^g\}\ \theta\ \varnothing, & \text{if } \exists t' \in \rho, \sigma \bullet t' \bowtie^{\varnothing}_{\sigma} t \\
\quad \text{where } t^g = \mathcal{G}[\![t]\!][\![t']\!]\ \theta & \\
(\kappa \bullet f) \rightarrow (\tau_f, \mathcal{D}_n[\![\mathcal{U}[\![\mathcal{R}[\![t]\!]]\!]\ \varnothing]\!]\ \langle\rangle\ (\rho \cup \{t\})\ \theta\ \varnothing), & \text{otherwise}
\end{array} \right. \\
\qquad\quad \text{where } t = \mathcal{T}_n[\![f]\!]\ \kappa\ \varnothing\ \theta\ \Delta \\
\mathcal{D}_n[\![e_0\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{D}_n[\![e_0]\!]\ \langle(\bullet\ e_1) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{D}_n[\![\mathbf{case}\ e_0\ \mathbf{of}\ p_1 \Rightarrow e_1\ |\cdots|\ p_k \Rightarrow e_k]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{D}_n[\![e_0]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e_1\ |\cdots|\ p_k \Rightarrow e_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\mathcal{D}_n[\![e = \mathbf{let}\ x = e_0\ \mathbf{in}\ e_1]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = (\kappa \bullet e) \rightarrow (\mathbf{let}, \mathcal{D}_n[\![e_1\{x \mapsto e_0\}]\!]\ \kappa\ (\rho \cup \{x \mapsto e_0\})\ \theta\ \Delta), (x, \mathcal{D}_n[\![e_0]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}_n[\![e_0\ \mathbf{where}\ f_1 = e_1 \ldots f_n = e_n]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{D}_n[\![e_0]\!]\ \kappa\ \rho\ \theta\ (\Delta \cup \{f_1 = e_1, \ldots, f_n = e_n\}) \\
\\
\mathcal{D}'_n[\![t]\!]\ \langle\rangle\ \rho\ \theta\ \Delta = t \\
\mathcal{D}'_n[\![t]\!]\ \langle(\bullet\ e) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = \mathcal{D}'_n[\![(t\ e) \rightarrow (@, t), (\#1, \mathcal{D}_n[\![e]\!]\ \langle\rangle\ \rho\ \theta\ )]\!]\ \kappa\ \rho\ \theta\ \Delta \\
\mathcal{D}'_n[\![x \rightarrow (x, \mathbf{n+1})]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1\ |\cdots|\ p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ x\ \mathbf{of}\ p_1 \Rightarrow e'_1\ |\cdots|\ p_k \Rightarrow e'_k) \rightarrow \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{D}_n[\![(\kappa \bullet e'_1)\{x \mapsto p_1\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{D}_n[\![(\kappa \bullet e'_k)\{x \mapsto p_k\}]\!]\ \langle\rangle\ \rho\ \theta\ \Delta) \\
\mathcal{D}'_n[\![t]\!]\ \langle(\mathbf{case}\ \bullet\ \mathbf{of}\ p_1 \Rightarrow e'_1\ |\cdots|\ p_k \Rightarrow e'_k) : \kappa\rangle\ \rho\ \theta\ \Delta \\
\qquad = (\mathbf{case}\ root(t)\ \mathbf{of}\ p_1 \Rightarrow e'_1\ |\cdots|\ p_k \Rightarrow e'_k) \rightarrow \\
\qquad\quad (\mathbf{case}, t), (p_1, \mathcal{D}_n[\![e'_1]\!]\ \kappa\ \rho\ \theta\ \Delta), \ldots, (p_k, \mathcal{D}_n[\![e'_k]\!]\ \kappa\ \rho\ \theta\ \Delta)
\end{array}
$$

Fig. 20. Distillation Transformation Rules

6 Conclusion and Related Work

We have defined a hierarchy of transformers in which the transformer at each level of the hierarchy makes use of the transformers at lower levels. At the bottom of the hierarchy is the level 0 transformer, which corresponds to positive supercompilation [14], and is capable of achieving only linear improvements in efficiency. The level 1 transformer corresponds to the first published definition of distillation [4], and is capable of achieving super-linear improvements in efficiency. Further improvements are possible at higher levels in the hierarchy, but the difficulty is in knowing the appropriate level at which to operate for an arbitrary input program. We have also shown how the more recently published definition of distillation [5] moves up through the levels of the transformation hierarchy until no further improvements can be made.

Previous works [9,2,1,18,13] have noted that the unfold/fold transformation methodology is incomplete; some programs cannot be synthesized from each other. This is because the transformation methodologies under consideration correspond to level 0 in our hierarchy; higher levels are required to achieve the desired results.

There have been several attempts to work on a meta-level above supercompilation, the first one by Turchin himself using *walk grammars* [16]. In this approach, traces through residual graphs are represented by regular grammars that are subsequently analysed and simplified. This approach is also capable of achieving superlinear speedups, but no automatic procedure is defined for it; the outlined heuristics and strategies may not terminate.

The most recent work on building a meta-level above supercompilation is by Klyuchnikov and Romanenko [8]. They construct a hierarchy of supercompilers in which lower level supercompilers are used to prove lemmas about term equivalences, and higher level supercompilers utilise these lemmas by rewriting according to the term equivalences (similar to the "second order replacement method" defined by Kott [10]). This approach is also capable of achieving superlinear speedups, but again no automatic procedure is defined for it; the need to find and apply appropriate lemmas introduces infinite branching into the search space, and various heuristics have to be used to used to try to limit this search.

Logic program transformation is closely related, and the equivalence of partial deduction and driving has been argued by Glück and Sørensen [3]. Superlinear speedups can be achieved in logic program transformation by *goal replacement* [11,12]: replacing one logical clause with another to facilitate folding. Techniques similar to the notion of "higher level supercompilation" [8] have been used to prove correctness of goal replacement, but have similar problems regarding the search for appropriate lemmas.

Acknowledgements

The work presented here owes a lot to the input of Neil Jones, who provided many useful insights and ideas in our collaboration during the sabbatical of the author at the Department of Computer Science, University of Copenhagen (DIKU). This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

1. Amtoft, T.: Sharing of Computations. Ph.D. thesis, DAIMI, Aarhus University (1993)
2. Andersen, L., Gomard, C.: Speedup Analysis in Partial Evaluation: Preliminary Results. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). pp. 1–7 (1992)
3. Glück, R., Jørgensen, J.: Generating Transformers for Deforestation and Supercompilation. In: Proceedings of the Static Analysis Symposium. Lecture Notes in Computer Science, vol. 864, pp. 432–448. Springer-Verlag (1994)

4. Hamilton, G.W.: Distillation: Extracting the Essence of Programs. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 61–70 (2007)
5. Hamilton, G.W., Jones, N.D.: Distillation with Labelled Transition Systems. In: Proceedings of the SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. pp. 15–24 (2012)
6. Huet, G.: The Zipper. Journal of Functional Programming 7(5), 549–554 (1997)
7. Jones, N., Gomard, C., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall (1993)
8. Klyuchnikov, I., Romanenko, S.: Towards Higher-Level Supercompilation. In: Proceedings of the Second International Workshop on Metacomputation in Russia (META) (2010)
9. Kott, L.: A System for Proving Equivalences of Recursive Programs. In: Proceedings of the Fifth Conference on Automated Deduction (CADE). pp. 63–69 (1980)
10. Kott, L.: Unfold/Fold Transformations. In: Nivat, M., Reynolds, J. (eds.) Algebraic Methods in Semantics, chap. 12, pp. 412–433. CUP (1985)
11. Petterossi, A., Proietti, M.: A Theory of Totally Correct Logic Pro- gram Transformations. In: Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM). pp. 159–168 (2004)
12. Roychoudhury, A., Kumar, K., Ramakrishnan, C., Ramakrishnan, I.: An Unfold/Fold Transformation Framework for Definite Logic Programs. ACM Transactions on Programming Language Systems 26(3), 464–509 (2004)
13. Sørensen, M.H.: Turchin's Supercompiler Revisited. Master's thesis, Department of Computer Science, University of Copenhagen (1994), DIKU-rapport 94/17
14. Sørensen, M.H., Glück, R., Jones, N.: A Positive Supercompiler. Journal of Functional Programming 6(6), 811–838 (1996)
15. Turchin, V.: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 90–121 (Jul 1986)
16. Turchin, V.: Program Transformation With Metasystem Transitions. Journal of Functional Programming 3(3), 283–313 (1993)
17. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Lecture Notes in Computer Science 300, 344–358 (1988)
18. Zhu, H.: How Powerful are Folding/Unfolding Transformations? Journal of Functional Programming 4(1), 89–112 (1994)

Obfuscation by Partial Evaluation of Distorted Interpreters
(Invited Talk)

Neil D. Jones

Computer Science Department
University of Copenhagen
2100 Copenhagen, Denmark
e-mail: neil@diku.dk

Abstract. How to construct a general *program obfuscator*? We present a novel approach to automatically generating obfuscated code P' from any program P whose source code is given. Start with a (program-executing) interpreter `interp` for the language in which P is written. Then "distort" `interp` so it is still correct, but its specialization P' w.r.t. P is transformed code that is equivalent to the original program, but harder to understand or analyze. Potency of the obfuscator is proved with respect to a general model of the attacker, modeled as an approximate (abstract) interpreter. A systematic approach to distortion is to make program P obscure by transforming it to P' on which (abstract) interpretation is *incomplete*. Interpreter distortion can be done by making residual in the specialization process sufficiently many interpreter operations to defeat an attacker in extracting sensible information from transformed code. Our method is applied to: code flattening, data-type obfuscation, and opaque predicate insertion. The technique is language independent and can be exploited for designing obfuscating compilers.

Keywords: Obfuscation, semantics, partial evaluation, program transformation, program interpretation, abstract interpretation.

The talk is based on joint work with Roberto Giacobazzi and Isabella Mastroeni [1].

References

1. Roberto Giacobazzi, Neil D. Jones, and Isabella Mastroeni. Obfuscation by partial evaluation of distorted interpreters. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, pages 63–72, New York, NY, USA, 2012. ACM.

Superlinear Speedup by Program Transformation
(Extended Abstract)

Neil D. Jones

Computer Science Department
University of Copenhagen
2100 Copenhagen, Denmark
e-mail: neil@diku.dk

G.. W. Hamilton

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

There seems to be, at least in practice, a fundamental conflict within program transformations. One way: *hand transformations* can yield dramatic speedups, but seem to require human insight. They are thus are only suited to small programs and have not been successfully automated. On the other hand, there exist a number of well-known *automatic program transformations*; but these have been proven to give at most linear speedups.

This work in progress addresses this apparent conflict, and concerns the principles and practice of superlinear program speedup. A disclaimer: we work in a simple sequential program context: no caches, parallelism, etc.

Many interesting program transformations (by Burstall-Darlington, Bird, Pettorossi, and many others) have been published that give superlinear program speedups on some program examples. However, these techniques all seem to require a "Eureka step" where the transformer understands some essential property relevant to the problem being solved (e.g., associativity, commutativity, occurrence of repeated subproblems, etc.). Such transformations have proven to be very difficult to automate.

On the other hand a number of fully automatic transformers exist, including: classical compiler optimisations, deforestation, partial evaluation and supercompilation. Mostly these give only linear speedups. (There are, however, two Refal-based exceptions: the supercompiler work by Turchin, and Nemytykhs supercompiler SCP4.)

The limitation to linear time improvement has been proven in some cases, e.g., by Jones and colleagues for partial evaluation (using call-by-value), and by Sørensen for positive supercompilation (using call-by-name).

An instance: a goal for several years was automatically to achieve the speedup of the Knuth-Morris-Pratt string pattern matching algorithm. The KMP speedup is still linear though, although the constant speedup coefficient can be proportional to the length of the pattern being searched for.

What principles can lead to superlinear speedup? Some examples that suggest principles to be discovered and automated:

1. In functional programs:
   - finding shared subcomputations (e.g., the Fibonacci example)

   - finding unneeded computations (e.g., most of the computation done by "naive reverse")
2. In imperative programs:
   - finding unneeded computations (e.g., major speedups can result from generalising the usual compiler "dead code" analysis also to span over program loops)
   - finding shared subcomputations (e.g., the factorial sum example)
   - code motion to move an entire nested loop outside an enclosing loop
   - strength reduction
   - common subexression elimination across loop boundaries, e.g., extending "value numbering"

In 2007 Hamilton showed that the "distillation" transformation (a further development of positive supercompilation) can sometimes yield superlinear speedups. Distillation has automatically transformed the quadratic-time "naive reverse" program, and the exponential-time "Fibonacci" program, each into a linear-time equivalent program that uses accumulating parameters.

On the other hand, there are subtleties, e.g., distillation works with a higher-order call-by-name source language. Further, distillation is a very complex algorithm, involving positive information propagation, homeomorphic embedding, generalisation by tree matching, and folding. A lot of the complexity in the algorithm arises from the use of potentially infinite data structures and the need to process these in a finite way. It is not yet clear which programs can be sped up so dramatically, and when and why this speedup occurs. It is as yet also unclear whether the approach can be scaled up to use in practical, industrial-strength contexts, as can classical compiler optimisations.

The aim of this work in progress is to discover an essential "inner core" to distillation. Our approach is to study a simpler language, seeking programs that still allow superlinear speedup. Surprisingly, it turns out that asymptotic speedups can be obtained even for first-order tail recursive call-by-value programs (in other words, imperative flowchart programs). An example discovered just recently concerned computing $f(n) = 1!+2!+...+n!$. Distillation transforms the natural quadratic time factorial sum program into a linear time equivalent.

Even though distillation achieves many of these effects automatically, the principles above seem to be buried in the complexities of the distillation algorithm and the subtleties of its input language.

One goal of our current work is to extract the essential transformations involved. Ideally, one could extend classical compiler optimisations (normally only yielding small linear speedups) to obtain a well-understood and automated "turbo" version that achieves substantially greater speedups, and is efficient enough for daily use.

References

1. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

2. Søren Debois. Imperative program optimization by partial evaluation. In Heintze and Sestoft [5], pages 113–122.
3. Geoffrey William Hamilton and Neil D. Jones. Distillation with labelled transition systems. In *PEPM (ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation)*, pages 15–24. ACM, 2012.
4. Geoffrey William Hamilton and Neil D. Jones. Proving the correctness of unfold/fold program transformations using bisimulation. In *Proceedings of the 8th Andrei Ershov Informatics Conference*, volume 7162 of *Lecture Notes in Computer Science*, pages 150–166. Springer, 2012.
5. Nevin Heintze and Peter Sestoft, editors. *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2004, Verona, Italy, August 24-25, 2004*. ACM, 2004.
6. Neil D. Jones. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 52:307–339, 2004.
7. V. F. Turchin. Supercompilation: Techniques and results. In *Perspectives of System Informatics*, volume 1181 of *LNCS*. Springer, 1996.

Why Multi-Result Supercompilation Matters: Case Study of Reachability Problems for Transition Systems

Andrei V. Klimov*

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
klimov@keldysh.ru

Abstract. We sum up some current results of the theoretical study of the reasons of successful application of supercompilers to verification of monotonic counter systems representing the models of practical protocols and other parallel systems. Three supercompilation-like algorithms for forward analysis of counter systems and a procedure interactively invoking a supercompiler to solve the coverability problem are presented. It always terminates for monotonic counter systems. The algorithms are considered as an instance of multi-result supercompilation (proposed by I. Klyuchnikov and S. Romanenko) for special-purpose analysis of counter systems.

We speculate on the insufficiency of single-result supercompilation for solving the reachability and coverability problems and the necessity of multi-result supercompilation. Related work by Gilles Geeraerts et al. on the algorithmic schema referred to as 'Expand, Enlarge and Check' (EEC) is discussed. We regard the work on EEC as a theory of domain-specific multi-result supercompilation for well-structured transition systems. The main purpose of the theory is to prune combinatorial search for suitable residual graphs as much as possible. Based of the EEC theory and our results, several levels of the restrictions of the combinatorial search dependent on the properties of a subject transition system are revealed: some minimal restrictions when solving reachability for arbitrary transition systems; more efficient search (according to the EEC schema) in the case of well-structured transitions systems; iterative calls of a single-result supercompiler with a varying parameter of generalization for monotonic counter systems. On the other hand, no reasonable practical class of transition systems, for which the reachability or coverability is always solvable by a version of single-result supercompilation is known yet.

Keywords: multi-result supercompilation, verification, reachability, coverability, well-structured transition systems, counter systems.

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

1 Introduction

Supercompilation [27] is a forward analysis and transformation method. *Forward* means that the traces of program execution are elaborated starting from a set of initial states to a certain depth. Respectively, *backward* analysis is based on elaboration of traces backwards from a set of final states. Many other forward and backward analysis techniques have been developed for specific classes of algorithm systems, and their comparison with supercompilation and cross-fertilization is a productive research topic.[1]

Traditionally supercompilation has been developed for programs, while in this paper we deal with *transition systems* and solving the reachability problems (reachability and coverability) for them.

Programs (algorithms) and transition systems differ in some respects:

- (Essential) The result of evaluation of a program is its *final state*. (In more detail: the sense of a program is a mapping from a set of initial states to a set of final states). The outcome of a transition system comprises *all* states reachable from a set of initial states and even more up to the set of all traces. Different outcomes are considered depending on a problem under consideration. For the reachability problems the set of all reachable states is used.
- (Inessential) Programs are usually deterministic, while transition systems nondeterministic. Nondeterministic systems are somewhat technically simpler for supercompilation. (There is no need of propagating negative conditions to else branches.)

The tasks of verification of programs or transition systems allows for easy and natural comparison of methods: if some method proves more properties than another one, the former is more "powerful" than the latter. I. Klyuchnikov and S. Romanenko came to the idea of multi-result supercompilation (MRSC) [15] while dealing with proving program equivalence by supercompilation: two programs are equivalent when the respective residual terms coincide. A multi-result supercompiler, which returns a set of residual programs instead of just one in case of single-results supercompilation, allows for proving the equivalence of more programs: two programs are equivalent when the respective sets of residual programs intersect.

Solving the reachability problem (whether a given set of states is reachable from a given set of initial states by a program or a transition system) turned out to be even more sensible to the use of multi-result supercompilation instead of single-result one.

The goal of this paper is to demonstrate and argue that, and under what conditions, multi-result supercompilation is capable of solving the reachability and coverability problems for transition systems, while single-result supercompilers solve the problem only occasionally.

[1] "Backward supercompilation" is also a promising topic, as well as "multi-directed" analysis from arbitrary intermediate program points in both directions. V. Turchin initiated this research in [25,28], but much work is still required.

History and Related Work. This paper continues research [12,13,14] into why and how *supercompilers* are capable of solving the *reachability* and *coverability problems* for *counter systems* where the set of target states is upward-closed and the set of initial states has a certain form. The idea of verification by supercompilation stems from the pioneering work by V. Turchin [26]. The fact that a supercompiler can solve these problems for a lot of practically interesting counter systems has been experimentally discovered by A. Nemytykh and A. Lisitsa [19,21] with a Refal supercompiler SCP4 [24] and then the result has been reproduced with the Java supercompiler JScp [10,11] and other supercompilers.

In paper [14] two classic versions of the supercompilation algorithm are formulated for counter systems, using the notation close to the works on Petri nets and transition systems. Another paper [13] contains a simplification of one of them (namely, with the so-called lower-node generalization) by clearing out the details that have been found formally unnecessary for solving the coverability problem. The algorithms are reproduced here.

It has been found that the coverability problem for monotonic counter systems is solvable by iterative application of a supercompiler varying an integer parameter $l = 0, 1, 2, \ldots$, that controls termination of driving and generalization of integers: small non-negative integers less than l are not allowed to be generalized to a variable.[2] Papers [13,14] contain a proof that there exists such value l that the supercompiler solves the problem. (Remarkably, the proof is based on the existence of a non-computable upper estimate of l.)

In [14] (and here in Algorithms 2 and 3) this rule is used together with the traditional whistle[3] based on a well-quasi-order: the whistle is not allowed to blow if two configurations under test differ only in integers less than l. In [13] (and here in Algorithm 1) this rule is used in pure form as an imperative: every integer greater or equal to l is immediately generalized.

The iterative procedure of application a supercompiler demonstrates usefulness of *gradual specialization*: starting with a trivial result of supercompilation gradually produce more and more specialized versions until one is obtained that satisfies the needs. This differs from the traditional design of supercompilers, which try to produce the most specific residual code at once.

These algorithms may be considered as an instance of *multi-result supercompilation* [15]: to solve a problem one varies some of the degrees of freedom inherent in supercompilation, obtains several residual programs and chooses a better one w.r.t. his specific goals. They have shown the usefulness of multi-result supercompilation for proving the equivalence of expressions and in two-level supercompilation.

In our work, a problem unsolvable by single-result supercompilation is solvable by enumerating a potentially infinite set of supercompilation results parameterized by an integer.

[2] In terms of Petri net theory this phrase sounds as follows: integer values less than l are not allowed to be *accelerated*, i.e., be replaced with a special value ω.

[3] In supercompilation jargon, a *whistle* is a termination rule.

The decidability of the coverability problem for well-structured transition systems, which counter systems belong to, is well-known [1]. The iterative procedure of application of the supercompilers presented below is close to the algorithmic schema referred to as 'Expand, Enlarge and Check' (EEC) [5,6] for solving the coverability problem of *well-structured transition systems* (WSTS). We discuss their relation is Section 4.

Outline. The paper is organized as follows.

In Section 2 we recall some known notions from the theory of transition systems as well as give specific notions from supercompilation theory used in the algorithms presented in Section 3.3. The algorithms are reproduced from [13,14] to illustrate the idea of multi-result supercompilation applied to solving the coverability problem. (If you are familiar with these papers, you may go immediately to Section 4 and return back when needed.)

Section 4 contains a new contribution and answers the question in the title: it takes a more general look at the methods of solving the reachability problems for transition systems; discusses the related work by G. Geeraerts et al., which we regard as a theory of multi-result supercompilation for well-structured transition systems (WSTS); and reveal an hierarchy of properties of transition systems, which allows for pruning extra enumeration of residual graphs and making multi-result supercompilation more efficient. Monotonic counter systems turned out to be the most efficient case in this hierarchy.

In Section 5 we discuss related work and in Section 6 conclude.

2 Basic Notions

2.1 Transition Systems

We use the common notions of *transition system*, *monotonic transition system*, *well-structured transition system*, *counter system* and related ones.

A *transition system* $\mathcal{S}$ is a tuple $\langle S, \Rightarrow \rangle$ such that S is a possibly infinite set of states, $\Rightarrow \subseteq S \times S$ a transition relation.

A *transition function* $\mathsf{Post}(\mathcal{S}, s)$ is used to denote the set $\{s' \mid s \Rightarrow s'\}$ of one-step successors of s.[4]

$\mathsf{Post}^*(\mathcal{S}, s)$ denotes the set $\{s' \mid s \stackrel{*}{\Rightarrow} s'\}$ of successors of s.

$\mathsf{Reach}(\mathcal{S}, I)$ denotes the set $\bigcup_{s \in I} \mathsf{Post}^*(\mathcal{S}, s)$ of states *reachable* from a set of states I.

We say a transition system $\langle S, \Rightarrow \rangle$ is (*quasi-, partially*) *ordered* if some (quasi-,[5] partial[6]) order $\preccurlyeq$ is defined on its set of states S.

For a quasi-ordered set X, $\downarrow X$ denotes $\{x \mid \exists y \in X \colon x \preccurlyeq y\}$, the downward closure of X. $\uparrow X$ denotes $\{x \mid \exists y \in X \colon x \succcurlyeq y\}$, the upward closure of X.

[4] For effectiveness, we assume the set of one-step successors is finite.

[5] A *quasi-order* (*preorder*) is a reflexive and transitive relation.

[6] A *partial order* is an antisymmetric quasi-order.

The *covering set* of a quasi-ordered transition system $\mathcal{S}$ w.r.t. an initial set I, noted $\mathsf{Cover}(\mathcal{S}, I)$, is the set $\downarrow\mathsf{Reach}(\mathcal{S}, I)$, the downward closure of the set of states reachable from I.

The *coverability problem* for a quasi-ordered transition system $\mathcal{S}$, an initial set of states I and an upward-closed target set of states U asks a question whether U is reachable from I: $\exists s \in I, s' \in U : s \stackrel{*}{\Rightarrow} s'$.[7]

A quasi-order $\preccurlyeq$ is a *well-quasi-order* iff for every infinite sequence $\{x_i\}$ there are two positions $i < j$ such that $x_i \preccurlyeq x_j$.

A transition system $\langle S, \Rightarrow\rangle$ equipped with a quasi-order $\preccurlyeq \subseteq S \times S$ is said to be *monotonic* if for every $s_1, s_2, s_3 \in S$ such that $s_1 \Rightarrow s_2$ and $s_1 \preccurlyeq s_3$ there exists $s_4 \in S$ such that $s_3 \stackrel{*}{\Rightarrow} s_4$ and $s_2 \preccurlyeq s_4$.

A transition system is called *well-structured (WSTS)* if it is equipped with a well-quasi-order $\preccurlyeq \subseteq S \times S$ and is monotonic w.r.t. this order.

A *k-dimensional counter system* $\mathcal{S}$ is a transition system $\langle S, \Rightarrow\rangle$ with states $S = \mathbb{N}^k$, k-tuples of non-negative integers. It is equipped with the component-wise partial order $\preccurlyeq$ on k-tuples of integers:

$$
\begin{aligned}
s_1 \preccurlyeq s_2 \quad &\text{iff} \quad \forall i \in [1, k] : \; s_1(i) \leq s_2(i), \\
s_1 \prec s_2 \quad &\text{iff} \quad s_1 \preccurlyeq s_2 \wedge s_1 \neq s_2.
\end{aligned}
$$

Proposition 1. *The component-wise order $\preccurlyeq$ of k-tuples of non-negative integers is a well-quasi order. A counter system equipped with this order is a well-structured transitions system.*

2.2 Configurations

In supercompilation the term *configuration* denotes a representation of a set of states, while in Petri net and transition system theories the same term stands for a ground state. In this paper the supercompilation terminology is used. Our term *configuration* is equivalent to *ω-configuration* and *ω-marking* in Petri net theory.

The general rule of construction of the notion of a configuration in a supercompiler from that of the program state in an interpreter is as follows: add configuration variables to the data domain and allow these to occur anywhere where a ground value can occur. A configuration represents the set of states that can be obtained by replacing configuration variables with all possible values. Thus the notion of a configuration implies a set represented by some constructive means rather than an arbitrary set.

A state of a counter system is a k-tuple of integers. According to the above rule, a configuration should be a tuple of integers and configuration variables. For the purpose of this paper we use a single symbol ω for all occurrences of variables and consider each occurrence of ω a distinct configuration variable.

[7] In other words, the coverability problem asks a question whether such a state r is reachable from I that $\downarrow\{r\} \cap U \neq \emptyset$, where there is no requirement that the target set U is upward-closed.

Thus, in supercompilation of k-dimensional counter systems *configurations* are k-tuples over $\mathbb{N}\cup\{\omega\}$, and we have the set of all configurations $\mathcal{C} = (\mathbb{N}\cup\{\omega\})^k$.

A configuration $c \in \mathcal{C}$ represents a set of states noted $[\![c]\!]$:

$$[\![c]\!] = \{\langle x_1, \ldots, x_k\rangle \mid x_i \in \mathbb{N} \text{ if } c(i) = \omega,\ x_i = c(i) \text{ otherwise},\ 1 \le i \le k\}.$$

These notations agree with that used in Petri net and counter system theories. Notice that by using one symbol ω we cannot capture information about equal unknown values represented by repeated occurrences of a variable. However, when supercompiling counter systems, repeated variables do not occur in practice, and such simplified representation satisfies our needs.

We also use an extension of $[\![\cdot]\!]$ to sets of configurations to denote all states represented by the configurations from a set C: $[\![C]\!] = \bigcup_{c\in C}[\![c]\!]$.

Definition 1 (Coverability set). *A* coverability set *is a finite set of configurations C that represents the covering set in the following way: $\downarrow[\![C]\!] = \mathsf{Cover}(\mathcal{S}, I)$.*

Notice that if we could find a coverability set, we could solve the coverability problem by checking its intersection with the target set U.

2.3 Residual Graph, Tree and Set

Definition 2 (Residual graph and residual set). *Given a transition system $\mathcal{S} = \langle S, \Rightarrow\rangle$ along with an initial set $I \subseteq S$ and a set $\mathcal{C}$ of configurations,* a residual graph *is a tuple $\mathcal{T} = \langle N, B, n_0, C\rangle$, where N is a set of nodes, $B \subseteq N\times N$ a set of edges, $n_0 \in N$ a root node, $C\colon N \to \mathcal{C}$ a labeling function of the nodes by configurations, and*

1. *$[\![I]\!] \subseteq [\![C(n_0)]\!]$, and for every state $s \in S$ reachable from I there exists a node $n \in N$ such that $s \in [\![C(n)]\!]$, and*
2. *for every node $n \in N$ and states s, s' such that $s \in [\![C(n)]\!]$ and $s \Rightarrow s'$ there exists an edge $\langle n, n'\rangle \in B$ such that $s' \in [\![C(n')]\!]$.*

We call the set $\{C(n) \mid n \in N\}$ of all configurations in the graph a residual set.

Notice that a residual set is a representation of an over-approximation of the set of reachable states: $\downarrow[\![\{C(n) \mid n \in N\}]\!] \supseteq \mathsf{Reach}(\mathcal{S}, I)$.

The term *residual* is borrowed from the metacomputation terminology, where the output of a supercompiler is referred to as a *residual graph* and a *residual program*. The literature on transition systems lacks a term for what we call *residual set*. They use only the term *coverability set*, which means a specific case of a residual set, where it is a precise representation of the covering set $\mathsf{Cover}(\mathcal{S}, I) = \downarrow\mathsf{Reach}(\mathcal{S}, I)$.

The value of these notions for our purpose is as follows. To solve the coverability problem it is sufficient to find a coverability set among the residual sets: then we check whether all configurations in the coverability set are disjoint with the target set or not. Unfortunately, computing a coverability set is undecidable

for counter systems of our interest. Fortunately, this is not necessary. It is sufficient to build a sequence of residual sets that contains a coverability set. We may not know which one among the residual sets is a coverability set (this is incomputable), it is sufficient to know it exists in the sequence. This is the main idea of our algorithm and the 'Expand, Enlarge and Check' (EEC) algorithmic schema of [6].

Notice that such procedure of solving the coverability problem does not use the edges of the residual graph, and we can keep in B only those edges that are needed for the work of our versions of supercompilation algorithms. Hence the definition of a residual tree:

Definition 3 (Residual tree). *A* residual tree *is a spanning tree of a residual graph. The root of the tree is the root node n_0 of the graph.*

2.4 Operations on Configurations

To define a supercompiler we need the transition function Post on states to be extended to the corresponding function Drive on configurations. It is referred to as (one-step) *driving* in supercompilation and must meet the following properties (where $s \in S$, $c \in \mathcal{C}$):

1. $\mathsf{Drive}(\mathcal{S}, s) = \mathsf{Post}(\mathcal{S}, s)$ — a configuration with ground values represents a singleton and its successor configurations are respective singletons;
2. $[\![\mathsf{Drive}(\mathcal{S}, c)]\!] \supseteq \bigcup\{\mathsf{Post}(\mathcal{S}, s) \mid s \in [\![c]\!]\}$ — the configurations returned by Drive over-approximate the set of one-step successors. This is the *soundness* property of driving. The over-approximation suits well for applications to program optimization, but for verification the result of Drive must be more precise. Hence the next property:
3. $[\![\mathsf{Drive}(\mathcal{S}, c)]\!] \subseteq \downarrow\bigcup\{\mathsf{Post}(\mathcal{S}, s) \mid s \in [\![c]\!]\}$ — for solving the coverability problem it is sufficient to require that configurations returned by Drive are subsets of the downward closure of the set of the successors.

For the practical counter systems we experimented with, the transition function Post is defined in form of a finite set of partial functions taking the coordinates v_i of the current state to the coordinates v'_i of the next state:

$$v'_i = \textbf{ if } G_i(v_1, \ldots, v_k) \textbf{ then } E_i(v_1, \ldots, v_k), i \in [1, k],$$

where the 'guards' G_i are conjunctions of elementary predicates $v_j \geq a$ and $v_j = a$, and the arithmetic expressions E_i consist of operations $x + y$, $x + a$ and $x - a$, where x and y are variables or expressions, $a \in \mathbb{N}$ a constant.

The same partial functions define the transition function Drive on configurations, the operations on ground data being generalized to the extended domain $\mathbb{N} \cup \{\omega\}$: $\forall a \in \mathbb{N}$: $a < \omega$ and $\omega + a = \omega - a = \omega + \omega = \omega$.

2.5 Restricted Ordering of Configurations of Counter Systems

To control termination of supercompilers we use a restricted partial order on integers $\preccurlyeq_l$ parameterized by $l \in \mathbb{N}$. For $a, b \in \mathbb{N} \cup \{\omega\}$, we have:

$$a \preccurlyeq_l b \quad \text{iff} \quad l \leq a \leq b < \omega.$$

This partial order makes two integers incompatible when one of them is less than l. The order is a well-quasi-order.

Then the partial order on states and configurations of counter systems is the respective component-wise comparison: for $c_1, c_2 \in \mathcal{C} = (\mathbb{N} \cup \{\omega\})^k$,

$$c_1 \preccurlyeq_l c_2 \quad \text{iff} \quad \forall i \in [1, k]\colon\ c_1(i) \preccurlyeq_l c_2(i).$$

This order is also a well-quasi-order. It may be regarded as a specific case of the homeomorphic embedding of terms used in supercompilation to force termination. As we will see in the supercompilation algorithms below, when two configurations c_1 and c_2 are met on a path such that $c_1 \prec_l c_2$, 'a whistle blows' and generalization of c_1 and c_2 is performed. Increasing parameter l prohibits generalization of small non-negative integers and makes 'whistle' to 'blow' later. When $l = 0$, the order is the standard component-wise well-quasi-order on tuples of integers. When $l = 1$, value 0 does not compare with other positive integers and generalization of 0 is prohibited. And so on.

2.6 Generalization

When the set of states represented by a configuration c is a subset of the set represented by a configuration g, $[\![c]\!] \subseteq [\![g]\!]$, we say the configuration g is *more general* than the configuration c, or g is a *generalization* of c.[8] Let $\sqsubseteq$ denote a *generalization relation* $\sqsubseteq \in \mathcal{C} \times \mathcal{C}$: $c \sqsubseteq g$ iff $[\![c]\!] \subseteq [\![g]\!]$, $c \sqsubset g$ iff $[\![c]\!] \subsetneq [\![g]\!]$.

For termination of traditional supercompilers generalization must meet the requirement of the finiteness of the number of possible generalization for each configuration:

$$\forall c \in \mathcal{C}\colon \{g \in \mathcal{C} \mid c \sqsubseteq g\} \text{ is finite.}$$

In Section 4 we speculate on a possibility to lift this restriction for multi-result supercompilation.

We use a function $\mathsf{Generalize}\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ to find a configuration g that is more general than two given ones. Usually $\mathsf{Generalize}(c_1, c_2)$ returns the least general configuration, however this is not formally required for the soundness and termination of supercompilation, as well as for the results of this paper, although it is usually desirable for obtaining 'better' residual programs.

In the case of counter systems and the set of configurations defined above, function $\mathsf{Generalize}(c_1, c_2)$ sets to ω those coordinates where configurations c_1 and c_2 differ:

$$\mathsf{Generalize}(c_1, c_2) = c \text{ s.t. } \forall i \in [1, k]\colon\ c(i) = c_1(i) \text{ if } c_1(i) = c_2(i),\ \omega \text{ otherwise.}$$

[8] In Petri net and counter system theories generalization is referred to as *acceleration*.

Notice that the function Generalize has no parameter l. However in the supercompilation algorithms below it is called in cases where $c_1 \prec_l c_2$. Hence generalization is not performed when one of the integers is less than l.

In Algorithm 1 a generalization function $\mathsf{Generalize}_l(c)$ with an integer parameter l and one configuration argument c is used. It sets to ω those coordinates that are greater or equal than l:

$$\mathsf{Generalize}_l(c) = g \text{ s.t. } \forall i \in [1,k]\colon\ g(i) = c(i) \text{ if } c(i) < l,\ \omega \text{ otherwise.}$$

Because of the importance of the parameter l, we write it in a special position: as the index of the function names that take it as an argument.

3 Supercompilation-Based Algorithms for Solving Reachabiliy Problems

In this section we present an algorithm to solve the reachability problem for transition systems. It is applicable to an arbitrary transition system provided the basic functions Drive, $\mathsf{Generalize}_l$ and Generalize are properly defined, but has been proven to terminate only for well-structured transition systems when solving the coverability problem, provided the basic functions satisfy certain properties. In the previous section we gave an example of their definition for counter systems. The theory by G. Geeraerts et al. [5,6] explains how to define them for well-structured transition systems.

The algorithm consists of the main function $\mathsf{Reachable}(\mathcal{S}, I, U)$ (Algorithm 0 Approximate) that contains the top level loop, which invokes functions $\mathsf{Over}_l(\mathcal{S}, I)$ and $\mathsf{Under}_l(\mathcal{S}, I)$ to iteratively compute more and more precise over- and under-approximations of the set of reachable states, while increasing parameter l.

We give three versions of the definition of the approximation functions. The simplest one (Algorithm 1) fits the EEC schema. Another two are based on domain-specific versions for transition systems of classical supercompilation algorithms with lower-node and upper-node generalization (Algorithm 2 ScpL and Algorithm 3 ScpU respectively).

3.1 Main Function

Main function $\mathsf{Reachable}(\mathcal{S}, I, U)$ takes a transition system $\mathcal{S}$, an initial configuration I and a target set U and iteratively applies functions $\mathsf{Under}_l(\mathcal{S}, I)$ and $\mathsf{Over}_l(\mathcal{S}, I)$ with successive values of the parameter of generalization $l = 0, 1, 2, \ldots$ When it terminates, it returns an answer whether the set U is reachable or not. The algorithm is rather natural.[9] Reachability is determined when some under-approximation returned by $\mathsf{Under}_l(\mathcal{S}, I)$ intersects with U. Unreachability is determined when some over-approximation returned by $\mathsf{Over}_l(\mathcal{S}, I)$ is disjoint with U.

[9] The algorithm is close to but even simpler than analogous Algorithm 4.1 in [5, page 122] and Algorithm 5.2 in [5, page 141].

Algorithm 0: $\mathsf{Reachable}(\mathcal{S}, I, U)$: Solving the coverability problem for a monotonic counter system

Data: $\mathcal{S}$ a monotonic counter system
Data: I an initial configuration (representing a set of initial states)
Data: U an upward-closed target set
Result: Is U reachable from I?

```
Reachable(S, I, U)
    for l = 0, 1, 2, ... do
        if [[Under_l(S, I)]] ∩ U ≠ ∅ then
            return 'Reachable'
        if [[Over_l(S, I)]] ∩ U = ∅ then
            return 'Unreachable'
```

We regard this algorithm as a domain-specific multi-result supercompilation when the approximation functions are implemented by means of single-result supercompilation.

3.2 Simplest Approximation Function

Function $\mathsf{Approximate}_l(over, \mathcal{S}, I)$ is invoked by functions $\mathsf{Over}_l(\mathcal{S}, I)$ and $\mathsf{Under}_l(\mathcal{S}, I)$ with a Boolean parameter *over* that tells either an over- or under-approximation is to be computed. The approximation depends on parameter l.

The set of residual configurations that is a partially evaluated approximation is collected in variable R starting from the initial configuration I. Untreated configurations are kept in a set T ('to treat'). The algorithm terminates when $T = \emptyset$.

At each step an arbitrary untreated configuration c is picked up from T and *driving step* is performed: the successors of c are evaluated by function Drive and each new configuration c' is processed as follows:

1. The current configuration c' is checked whether it is covered or not by one of the configurations collected in R: $\nexists \bar{c} \in R \colon \bar{c} \sqsupseteq c'$. If covered (such $\bar{c}$ exists) there is no need to proceed c' further, as its descendants are covered by the descendants of the existing configuration $\bar{c}$.
2. In the case of under-approximation, it is checked whether the current configuration c' is not to be generalized by comparing it with $g = \mathsf{Generalize}_l(c')$. If it is to be generalized, the configuration c' is not proceeded further. In such a way, configurations reachable from the initial configuration I without generalization are collected in R. They form an under-approximation.
3. The (possibly) generalized current configuration g is added to the sets R and T. Additionally, the configurations from R and T covered by the new one are deleted from the sets.

Algorithm 1: $\mathsf{Approximate}_l(over, \mathcal{S}, I)$, $\mathsf{Over}_l(\mathcal{S}, I)$: $\mathsf{Under}_l(\mathcal{S}, I)$, Building over- and under-approximations of a set of reachable states of a transition system

Data: $\mathcal{S}$ a transition system
Data: I an initial configuration
Data: l an integer parameter of generalization
Data: $over$ = **true** an over-approximation, **false** an under-approximation
Result: R an under- or over-approximation of a set of reachable states

$\mathsf{Approximate}_l(over, \mathcal{S}, I)$
  $R \leftarrow \{I\}$
  $T \leftarrow \{I\}$
  while $T \neq \emptyset$ **do**
    select some configuration $c \in T$
    $T \leftarrow T \setminus \{c\}$
    foreach $c' \in \mathsf{Drive}(\mathcal{S}, c)$ **do**
      $g \leftarrow \mathsf{Generalize}_l(c')$
      if $\nexists \bar{c} \in R\colon \bar{c} \sqsupseteq c' \wedge (over \vee g = c')$ **then**
        $R \leftarrow R \cup \{g\} \setminus \{\bar{c} \in R \mid \bar{c} \sqsubset g\}$
        $T \leftarrow T \cup \{g\} \setminus \{\bar{c} \in T \mid \bar{c} \sqsubset g\}$
  return R

$\mathsf{Over}_l(\mathcal{S}, I) = \mathsf{Approximate}_l(\mathbf{true}, \mathcal{S}, I)$
$\mathsf{Under}_l(\mathcal{S}, I) = \mathsf{Approximate}_l(\mathbf{false}, \mathcal{S}, I)$

The algorithm $\mathsf{Reachable}(\mathcal{S}, I, U)$ with this definition of approximation functions and the Drive and $\mathsf{Generalize}_l$ functions from the previous section does not always terminate for an arbitrary counter system $\mathcal{S}$ but do terminate for a monotonic counter system and an upward-closed target set U.

3.3 Supercompilation of Transition Systems

In this section we define two classic supercompilation algorithms for well-quasi-ordered transition systems.

Supercompilation is usually understood as an equivalence transformation of programs, transition systems, etc., from a *source* one to a *residual* one. However, for the purpose of this paper the supercompilation algorithms presented here returns a part of information sufficient to extract the residual set of configurations rather than a full representation of a residual transition system.

Two algorithms, Algorithm 2 ScpL and Algorithm 3 ScpU, have very much in common. They take a transition system $\mathcal{S}$, a quasi-order $\prec$ on the set of configurations and an initial configuration I, and return a residual tree, which represents an over-approximation of the set of states reachable from the initial configuration I. The order $\prec$ is a parameter that controls the execution of the algorithms, their termination and influences resulting residual trees. If the order

Algorithm 2: ScpL: Supercompilation of a quasi-ordered transition system with lower-node generalization.

Data: $\mathcal{S}$ a transition system
Data: I an initial configuration
Data: $\prec$ a quasi-order on configurations, a binary whistle
Result: $\mathcal{T}$ a residual tree

$\mathsf{ScpL}(\mathcal{S}, I, \prec)$
  $\mathcal{T} \leftarrow \langle N, B, n_0, C\rangle$ where $N = \{n_0\}$, $B = \emptyset$, $C(n_0) = I$, n_0 a new node
  $T \leftarrow \{n_0\}$
  while $T \neq \emptyset$ **do**
    select some node $n \in T$
    $T \leftarrow T \setminus \{n\}$
    if $\exists \bar{n} \in N\colon\ C(\bar{n}) \sqsupseteq C(n)$ **then** — terminate the current path (1)
      do nothing
    else if $\exists \bar{n}\colon\ B^+(\bar{n}, n) \wedge C(\bar{n}) \prec C(n) \wedge C(\bar{n}) \not\sqsupseteq C(n)$ **then**
      — generalize on whistle (2)
      $\bar{n} \leftarrow$ some node such that $B^+(\bar{n}, n) \wedge C(\bar{n}) \prec C(n)$
      $C(n) \leftarrow \mathsf{Generalize}(C(\bar{n}), C(n))$
      mark n as generalized
      $T \leftarrow T \cup \{n\}$
    else — unfold (drive) otherwise (3)
      foreach $c \in \mathsf{Drive}(\mathcal{S}, C(n))$ **do**
        $n' \leftarrow$ a new node
        $C(n') \leftarrow c$
        $N \leftarrow N \cup \{n'\}$
        $B \leftarrow B \cup \{\langle n, n'\rangle\}$
        $T \leftarrow T \cup \{n'\}$
  return $\mathcal{T}$

$\mathsf{Over}_l(\mathcal{S}, I) = \mathbf{let}\ \langle N, B, n_0, C\rangle = \mathsf{ScpL}(\mathcal{S}, I, \prec_l)\ \mathbf{in}\ \{C(n) \mid n \in N\}$
$\mathsf{Under}_l(\mathcal{S}, I) = \mathbf{let}\ \langle N, B, n_0, C\rangle = \mathsf{ScpL}(\mathcal{S}, I, \prec_l)\ \mathbf{in}\ \{C(n) \mid n \in N \wedge$
$\forall \bar{n}$ s.t. $B^*(\bar{n}, n)\colon\ \bar{n}$ is not marked as generalized$\}$

is a well-quasi-order the algorithms terminates for sure. Otherwise, in general, the algorithms sometimes terminate and sometimes do not.

The residual trees are gradually constructed from the root node n_0.

The nodes are labeled with configurations by a labeling function $C : N \to \mathcal{C}$, initially $C(n_0) = I$.

Untreated leaves are kept in a set T ('to treat') in Algorithm 2 and in a stack T in Algorithm 3. The first algorithm is non-deterministic and takes leaves from set T in arbitrary order. The second algorithm is deterministic and takes leaves from stack T in the FIFO order. The algorithms terminate when $T = \emptyset$ and $T = \epsilon$ (the empty sequence) respectively.

At each step, one of three branches is executed, marked in comments as (1), (2) and (3). Branches (1) and (3) are almost identical in the two algorithms.

Algorithm 3: ScpU: Supercompilation of a quasi-ordered transition system with upper-node generalization.

Data: $\mathcal{S}$ a transition system
Data: I an initial configuration
Data: $\prec$ a quasi-order on configurations, a binary whistle
Result: $\mathcal{T}$ a residual tree

```
ScpU(S, I, ≺)
    T ← ⟨N, B, n0, C⟩ where N = [n0], B = ∅, C(n0) = I, n0 a new node
    T ← [n0]
    while T ≠ ϵ do
        n ← Last(T)
        T ← T \ {n}
        if ∃n̄ ∈ N: C(n̄) ⊒ C(n) then          — terminate the current path (1)
            do nothing
        else if ∃n̄: B⁺(n̄, n) ∧ C(n̄) ≺ C(n) then  — generalize on whistle (2)
            n̄ ← the highest node such that B⁺(n̄, n) ∧ C(n̄) ≺ C(n)
            C(n̄) ← Generalize(C(n̄), C(n))
            mark n̄ as generalized
            T ← RemoveSubtreeExceptRoot(n̄, T)
            T ← T \ {n | B⁺(n̄, n)}              — drop nodes lower than n̄
            T ← Append(T, n̄)
        else                                    — unfold (drive) otherwise (3)
            foreach c ∈ Drive(S, C(n)) do
                n′ ← a new node
                C(n′) ← c
                N ← N ∪ {n′}
                B ← B ∪ {⟨n, n′⟩}
                T ← Append(T, n′)
    return T

Over_l(S, I)  = let ⟨N, B, n0, C⟩ = ScpU(S, I, ≺_l) in {C(n) | n ∈ N}
Under_l(S, I) = let ⟨N, B, n0, C⟩ = ScpU(S, I, ≺_l) in {C(n) | n ∈ N ∧
                    ∀n̄ s.t. B*(n̄, n): n̄ is not marked as generalized}
```

- Branches (1): if a configuration $C(\bar{n})$ more general than the current one $C(n)$ exists in the already constructed tree, the current path is terminated and nothing is done.
- Branches (3): if the conditions on branches (1) and (2) do not hold, a *driving step* is performed: the successors of the current configuration $C(n)$ are evaluated by the function Drive; for each new configuration c a new node n' is created; edges from the current node n to the new ones are added to the tree and the new nodes are added to set (or respectively, stack) T of untreated nodes.
- Branches (2) check whether on the path to the current node n (call it *lower*) there exists a node $\bar{n}$ (call it *upper*) with the configuration $C(\bar{n})$ which is

less than the current one $C(n)$, generalize the two configurations and assign the generalized configuration to the lower node in Algorithm 2 ScpL and to the upper node in Algorithm 3 ScpU. In the latter case the nodes below $\bar{n}$ are deleted from the residual tree and from stack T of untreated nodes. The nodes where generalization has been performed are marked as 'generalized'. These marks are used in the Algorithm 0.

Over- and under-approximations $\mathsf{Over}_l(\mathcal{S}, I)$ and $\mathsf{Under}_l(\mathcal{S}, I)$ are extracted from the residual tree. The over-approximation is the set of all configurations in the residual tree. The under-approximation is comprised of the configurations developed from the initial configuration without generalization (where all nodes on the path are not marked as 'generalized').

The two supercompilation algorithms always terminate for a quasi-ordered transition systems with a set of configuration such that every configuration has finitely many generalizations. The algorithm $\mathsf{Reachable}(\mathcal{S}, I, U)$ with these definitions of approximation functions and the proper definition of the Drive and $\mathsf{Generalize}_l$ functions, does not terminate for an arbitrary transition system, even for an well-structured transition systems. It terminates for so called *degenerated* transition systems (see discussion in Section 4), which strongly monotonic counter systems belong too, as well as for monotonic counter systems, which may be not degenerated, but guarantee the termination as well.

4 Why Multi-Result Supercompilation Matters?

In this section we recap the ideas, on which solving the reachability problems (reachability and coverability) is based, and argue that multi-result supercompilation is powerful enough to solve the reachability problems for transition systems under certain conditions. The main problem of multi-result supercompilation is the blow-up of the exhaustive search of a suitable residual graph. An efficient multi-result supercompiler should prune as much extra search branches as possible. Possibilities to do so depend on the properties of transition system under analysis.

Papers [6,7] and the PhD thesis [5] by G. Geeraerts demonstrate that (and how) a kind of multi-result supercompilation solves the coverability problem for well-structured transition systems (WSTS) with some combinatorial search. Also they showed that several classes of monotonic counter systems belong to a "degenerated" case, which allows for limited enumeration of residual graphs. In [13,14] we strengthened this result by showing that this is true for *all* monotonic counter systems, provided the counter system and the driving function are good enough ($[\![\mathsf{Drive}(\mathcal{S}, c)]\!] \subseteq \downarrow\mathsf{Post}(\mathcal{S}, c)$).

Let us discuss when the reachability problem for transition systems can be solved in principle, when and how multi-result supercompilation can find a solution, how the combinatorial search can be pruned based on the properties of a transition system, a set of configurations and driving and generalization functions.

But first, notice an important difference between single- and multi-result supercompilation. One of the main properties of a single-result supercompiler is its termination. The respective property of multi-result supercompilation is the fairness of enumeration of residual graphs. The notion of fairness depends on the problem under consideration: a fair multi-result supercompiler should enumerate either all residual graphs, or at least a subset of them sufficient to solve the problem.

4.1 Reachability Problem for Transition Systems

To implement a supercompiler for a transition system $\mathcal{S}$ one needs a set of configurations $\mathcal{C}$, a driving function Drive, and a generalization function $\mathsf{Generalize}$ that enumerates generalizations of a given configuration. Solving the reachability problem by supercompilation (as well as any forward analysis) is based on two ideas:

1. The supercompiler returns a residual set of configurations $R \subset \mathcal{C}$ which is a fixed point of the driving function:

$$[\![I]\!] \subseteq [\![R]\!] \wedge [\![\mathsf{Drive}(\mathcal{S}, R)]\!] \subseteq [\![R]\!].$$

2. To prove that a target set U is unreachable from I we check:

$$[\![R]\!] \cap U = \emptyset.$$

This is a necessary condition: if such R does not exist, no supercompiler can solve the reachability problem even if U is unreachable from I indeed.

This observation allows us to see the most essential limitation of the supercompilation method for solving the reachability problem: if the set of configurations $\mathcal{C}$ includes the representations of all such fixed points then we may expect of multi-result supercompilation to solve the problem when such a solution exists; if not we cannot expect. Designing such a complete set of configurations faces a conflict with the requirement that any configuration should have finitely many generalizations, which is used in traditional single-result supercompilers to guarantee termination. It is also used in the supercompilation-like algorithms presented above since they iteratively call the single-result supercompilers. In multi-result supercompilation, where the finiteness of the whole process is not expected (as it enumerates infinitely many residual graphs), this restriction of the finiteness of the number of generalizations could be lifted. However, this topic is not studied yet.

Thus, multi-result supercompilation could, in principle, solve the reachability problem when a solution is representable in form of a fixed point of Drive. The main problem is the exponential blow-up of the search space.

4.2 Pruning Enumeration of Residual Graphs

Do *all* residual sets are actually needed?

The first step of pruning the search space is based on the monotonicity of Drive (as a set function) and is applicable to any forward analysis solving the reachability problem: It is sufficient for a multi-result supercompiler to return not all fixed points R such that $R \supseteq I$, rather for every such fixed point R it should return at least one fixed point $R' \subseteq R$ such that $R' \supseteq I$.

This allows for a multi-result supercompiler to consider at each step of driving only the most specific generalizations among suitable ones. Unfortunately, there are many of them in general case, and fortunately, there is just one in our algorithms for monotonic counter systems.

This idea is utilized in the 'Expand, Enlarge and Check' (EEC) method by G. Geeraerts et al. The idea suffices to formulate the algorithmic schema and to prove that it terminates and solves the reachability problem when driving is perfect and the solution is representable in $\mathcal{C}$. (It is representable indeed for well-structured transition system.)

4.3 Ordered and Well-Structured Transition Systems

Now let us turn to transition systems with ordered sets of states (at least quasi-ordered) and to the coverability problem, which implies the target set of states is upward-closed. Two properties of transition systems may be formulated in terms of the ordering:

1. the order may be a well-quasi-order;
2. the transition system may be monotonic w.r.t. this order.

How does these properties influence the problem of the completeness of the set of configurations and the requirements of driving?

The well-quasi ordering of the set of states allows for a finite representation of all downward-closed sets. This is based on that any upward-closed subset of a well-quasi-ordered set has a finite number of *generators*, its minimal elements that uniquely determine the subset. A downward-closed set is the complement of some upward-closed set, and hence the generators of the complement determines it as well. However, such a "negative" representation may be inconvenient in practice (to implement driving), and G. Geeraerts et al. required that there exists a set of constructive objects called *limits* such that any downward closed set is representable by a finite number of limits. In terms of supercompilation, the limits are non-ground configurations.

The monotonicity of the transition system allows for using only downward-closed sets as configurations. That is, for solving the coverability problem, the Drive function may generalize configurations downwards at each step: $[\![\mathsf{Drive}(\mathcal{S}, c)]\!] \subseteq {\downarrow}\mathsf{Post}(\mathcal{S}, \{c\})$, rather than be perfect: $[\![\mathsf{Drive}(\mathcal{S}, c)]\!] = \mathsf{Post}(\mathcal{S}, \{c\})$.

These are the main ideas the EEC algorithmic schema for solving the coverability problem for WSTS is based upon, except the last one described in the next subsection.

4.4 Coverability Problem for WSTS: EEC Schema of Algorithms

The 'Expand, Enlarge and Check' algorithmic schema (EEC) suggests to split enumeration of residual graphs in two levels. Our algorithm with the Approximate function fits well the EEC schema and is actually its simplest instance. Refer to it as an example.

To define an EEC algorithm one selects an expanding sequence of finite sets of configurations such that $\mathcal{C}_l \subseteq \mathcal{C}_{l+1}$ and $\mathcal{C} = \bigcup_l \mathcal{C}_l$. For example, for counter systems Algorithm Approximate uses $\mathcal{C}_l = \{1, \ldots, l, \omega\}^k$, sets of configurations with coordinates not greater than l or equal to ω .

An EEC algorithm consists of a top level loop and a multi-result supercompilation-like algorithm invoked in each iteration with the requirement that only configurations from $\mathcal{C}_l$ are used in construction of the set of residual graphs. Since $\mathcal{C}_l$ is finite, the set of residual graphs is finite as well, hence each iteration surely terminates.

Thus, the lower level of enumeration of residual graphs is performed in each iteration, and the upper level of enumeration is organized by the top level loop.

Notice that the other two Algorithms ScpL and ScpU do not fit the EEC schema exactly, since the sets of configurations are not fixed in advance. But they also forcedly restrict the sets of possible residual graphs explored in each iteration by making the set of graphs finite for a given transition system and a given initial configuration with the use of the well-quasi-order $\preccurlyeq_l$ parameterized by integer l.

It is an open problem for future work to device direct multi-result supercompilation algorithms that efficiently enumerate residual graphs in one process rather than in sequential calls of a single-result supercompiler. In [12] an example of such a monolithic algorithm obtained by manually fusing the top level loop with Algorithm Approximate is presented.

4.5 Coverability Problem for Monotonic Counter Systems

We saw that in the general case after each driving step the exploration of all most specific suitable generalizations is required in order not to loose residual graphs. However, there may be such a case that the most specific generalization (represented as a finite set of allowed configurations) is just one. G. Geeraerts [5,6] calls this case *degenerated*. This depends on the specifics of a transition system and the sets of configurations $\mathcal{C}_l$. In [5,6] such sets of configurations are called *perfect* and it is proved that for strongly monotonic counter systems sets of configurations $\mathcal{C}_l = \{1, \ldots, l, \omega\}^k$ are perfect.

Our proofs of termination of the above algorithms [13,14] shows that the requirement of the strong monotonicity can be weaken to the general monotonicity.

Thus, in the degenerated case of monotonic counter systems many residual graphs are produced due to the top level loop only, while in the loop body the use of a single-result supercompiler is sufficient.

5 Related Work

Supercompilation. This research originated from the experimental work by A. Nemytykh and A. Lisitsa on verification of cache-coherence protocols and other models by means of the Refal supercompiler SCP4 [18,19,20,21]. It came as a surprise that all of the considered correct models had been successfully verified rather than some of the models had been verified while others had not, as is a common situation with supercompiler applications. It was also unclear whether the evaluation of the heuristic parameter to control generalization of integers discovered by A. Nemytykh could be automated. Since then the theoretical explanation of these facts was an open problem.

In invited talk [22] A. Lisitsa and A. Nemytykh reported that supercompilation with upper-node generalization and without the restriction of generalization (i.e., with $l = 0$) was capable of solving the coverability problem for ordinary Petri nets, based on the model of supercompilation presented in their paper [21].

In this paper the problem has been solved for a larger class, and the sense of the generalization parameter has been uncovered. However the problem to formally characterize some class of non-monotonic counter systems verifiable by the same algorithm, which the other part of the successful examples belongs to, remains open.

We regard the iterative invocation of single-result supercompilation with a varying parameter as a domain-specific instance of multi-result supercompilation suggested by I. Klyuchnikov and S. Romanenko [15]. As they argue and as this paper demonstrates, multi-result supercompilation is capable of significantly extending the class of program properties provable by supercompilation.

Partial Deduction. Similar work to establish a link between algorithms in Petri net theory and program specialization has been done in [8,16,17]. Especially close is the work [17] where a simplified version of partial deduction is put into one-to-one correspondence with the Karp&Miller algorithm [9] to compute a coverability tree of a Petri net. Here a Petri net is implemented as a (non-deterministic) logic program and partial deduction is applied to produce a specialized program from which a coverability set can be directly obtained.

(Online) partial deduction and supercompilation has many things in common. The method of [17] can be transferred from partial deduction to supercompilation, and our work is a step forward in the same direction after [17].

Petri Nets and Transition Systems. Transition systems and their subclasses—Petri nets and counter systems—have been under intensive study during last decades: [1,2,4,5,6,7,9], just to name a few. Supercompilation resembles forward analysis algorithms proposed in the literature.

A recent achievement is an algorithmic schema referred to as 'Expand, Enlarge and Check' (EEC). In paper [6] and in the PhD thesis by G. Geeraerts [5] a proof is given that any algorithm that fits EEC terminates on a well-structured

transition systems (WSTS) and an upper-closed target set and solves the coverability problem.

The first of the presented algorithm fits the EEC schema, and could be proved correct by reduction to EEC. Other two Algorithms ScpL and ScpU do not fit EEC exactly, but are very close.

Algorithm 2 ScpL can be seen as a further development of the classic Karp&Miller algorithm [9] to compute a coverability set of a Petri net, and Algorithm 3 ScpU resembles the *minimal coverability tree* (MCT) algorithm by A. Finkel [4] (in which an error has been found [7]) and later attempts to fix it [7,23].[10]

6 Conclusion

We presented three versions of supercompilation-based algorithms, which solve the coverability problem for monotonic counter systems. Although the algorithms are rather short they present the main notions of supercompilation: configurations, driving and configuration analysis of two kinds—with lower-node and upper-node generalization.

The idea of multi-result supercompilation was demonstrated by these algorithms and future work to develop more powerful domain-specific multi-result supercompilers that would solve the coverability problem for well-structured transition systems as well as the reachability problem for some specific classes of non-monotonic transition systems, was discussed. This seems impossible with single-result supercompilation.

Acknowledgements. I am very grateful to Sergei Abramov, Robert Glück, Sergei Grechanik, Arkady Klimov, Yuri Klimov, Ilya Klyuchnikov, Alexei Lisitsa, Andrei Nemytykh, Anton Orlov, Sergei Romanenko, Artem Shvorin, Alexander Slesarenko and other participants of the Moscow Refal seminar for the pleasure to collaborate with them and exchange ideas on supercompilation and its applications. My work and life have been greatly influenced by Valentin Turchin whom we remember forever.

References

1. Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, July 27–30, 1996*, pages 313–321. IEEE Computer Society, 1996.
2. Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST: acceleration from theory to practice. *International Journal on Software Tools for Technology Transfer*, 10(5):401–424, 2008.

[10] The master thesis by K. Luttge [23] is beyond my reach. But its essence is explained in the PhD thesis by G. Geeraerts [5, pages 172–174].

3. Ed Clarke, Irina Virbitskaite, and Andrei Voronkov, editors. *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011*, volume 7162 of *Lecture Notes in Computer Science*. Springer, 2012.
4. Alain Finkel. The minimal coverability graph for Petri nets. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993, Papers from the 12th International Conference on Applications and Theory of Petri Nets, Gjern, Denmark, June 1991*, volume 674 of *Lecture Notes in Computer Science*, pages 210–243. Springer, 1993.
5. Gilles Geeraerts. *Coverability and Expressiveness Properties of Well-Structured Transition Systems*. PhD thesis, Université Libre de Bruxelles, Belgique, May 2007.
6. Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. Expand, Enlarge and Check: New algorithms for the coverability problem of WSTS. *Journal of Computer and System Sciences*, 72(1):180–203, 2006.
7. Gilles Geeraerts, Jean-François Raskin, and Laurent Van Begin. On the efficient computation of the minimal coverability set of Petri nets. In K.S. Namjoshi et al., editor, *Proceedings of ATVA'07 – 5th International Symposium on Automated Technology for Verification and Analysis*, volume 4762 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2007.
8. Robert Glück and Michael Leuschel. Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In Dines Bjørner, Manfred Broy, and Alexandre V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99, Akademgorodok, Novosibirsk, Russia, July 6-9, 1999. Proceedings*, volume 1755 of *Lecture Notes in Computer Science*, pages 93–100. Springer, 2000.
9. Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.
10. Andrei V. Klimov. *JVer Project: Verification of Java programs by the Java Supercompiler*. Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 2008. : `http://pat.keldysh.ru/jver/`.
11. Andrei V. Klimov. A Java Supercompiler and its application to verification of cache-coherence protocols. In *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2010.
12. Andrei V. Klimov. Multi-result supercompilation in action: Solving coverability problem for monotonic counter systems by gradual specialization. In R. Glück M. Bulyonkov, editor, *International Workshop on Program Understanding (PU 2011), July 2–5, 2011, Novososedovo, Russia.*, pages 25–32. Novosibirsk: Ershov Institute of Informatics Systems, 2011.
13. Andrei V. Klimov. Yet another algorithm for solving coverability problem for monotonic counter systems. In Valery Nepomnyaschy and Valery Sokolov, editors, *Second Workshop "Program Semantics, Specification and Verification: Theory and Applications", PSSV'11, St. Petersburg, Russia, June 12–13, 2011*, pages 59–67. Yaroslavl State University, 2011.
14. Andrei V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In Clarke et al. [3], pages 193–209.
15. Ilya Klyuchnikov and Sergei Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In Clarke et al. [3], pages 207–223.

16. Michael Leuschel and Helko Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In John W. Lloyd et al., editor, *Computational Logic – CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2000.
17. Michael Leuschel and Helko Lehmann. Solving coverability problems of Petri nets by partial deduction. In *Proceedings of the 2nd international ACM SIGPLAN conference on principles and practice of declarative programming, September 20-23, 2000, Montreal, Canada*, pages 268–279. ACM, 2000.
18. Alexei P. Lisitsa and Andrei P. Nemytykh. Towards verification via supercompilation. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), 25-28 July 2005, Edinburgh, Scotland, UK*, pages 9–10. IEEE Computer Society, 2005.
19. Alexei P. Lisitsa and Andrei P. Nemytykh. *Experiments on verification via supercompilation.* Program Systems Instiute, Russian Academy of Sciences, 2007. : `http://refal.botik.ru/protocols/`.
20. Alexei P. Lisitsa and Andrei P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.
21. Alexei P. Lisitsa and Andrei P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.
22. Alexei P. Lisitsa and Andrei P. Nemytykh. Solving coverability problems by supercompilation. Invited talk. In *The Second Workshop on Reachability Problems in Computational Models (RP08), Liverpool, UK, September 15–17, 2008*, 2008.
23. K. Luttge. *Zustandsgraphen von Petri-Netzen.* Humboldt-Universität zu Berlin, Germany, 1995.
24. Andrei P. Nemytykh. The supercompiler SCP4: General structure. In Manfred Broy and Alexandre V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003. Revised Papers*, volume 2890 of *Lecture Notes in Computer Science*, pages 162–170. Springer, 2003.
25. Valentin F. Turchin. The language Refal, the theory of compilation and metasystem analysis. Courant Computer Science Report 20, Courant Institute of Mathematical Sciences, New York University, 1980.
26. Valentin F. Turchin. The use of metasystem transition in theorem proving and program optimization. In J. W. de Bakker and Jan van Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 645–657. Springer, 1980.
27. Valentin F. Turchin. The concept of a supercompiler. *Transactions on Programming Languages and Systems*, 8(3):292–325, 1986.
28. Valentin F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.

Automatic Verification of Counter Systems via Domain-Specific Multi-Result Supercompilation*

Andrei V. Klimov, Ilya G. Klyuchnikov, and Sergei A. Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Abstract. We consider an application of supercompilation to the analysis of counter systems. Multi-result supercompilation enables us to find the best versions of the analysis by generating a set of possible results that are then filtered according to some criteria. Unfortunately, the search space may be rather large. However, the search can be drastically reduced by taking into account the specifics of the domain. Thus, we argue that a combination of domain-specific and multi-result supercompilation may produce a synergistic effect. Low-cost implementations of domain-specific supercompilers can be produced by using prefabricated components provided by the MRSC toolkit.

1 Introduction

Supercompilation is a program manipulation technique that was originally introduced by V. Turchin in terms of the programming language Refal (a first-order applicative functional language) [37], for which reason the first supercompilers were designed and developed for the language Refal [35,39,29].

Further development of supercompilation led to a more abstract reformulation of supercompilation and to a better understanding of which details of the original formulation were Refal-specific and which ones were universal and applicable to other programming languages [32,33,3]. It particular, it was shown that supercompilation is as well applicable to non-functional programming languages (imperative and object-oriented ones) [6].

Also, despite the fact that from the very beginning supercompilation was regarded as a tool for both program optimization and program analysis [36], the research in supercompilation, for a long time, was primarily focused only on program optimization. Recently, however, we have seen a revival of interest in the application of supercompilation to inferring and proving properties of programs [25,12,10].

Multi-result supercompilation is a technique of constructing supercompilers that, given an input program, are able to produce a set of residual programs, rather than just a single one [13,8].

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

The purpose of the present work is to show, by presenting a concrete example, that multi-result, domain-specific supercompilation is not a theoretical curiosity, but rather a workhorse having certain advantages over general-purpose, single-result (deterministic) supercompilation. Some of the reasons are the following.

- Tautologically speaking, a general-purpose supercompiler should deal with programs written in a general-purpose subject language that, by definition, is not dedicated to a particular problem domain. Thus, for a given domain, the subject language may be too sophisticated, but, on the other hand, lacking in certain features.
- In cases where supercompilation is used for the purposes of analysis and verification, the problem of reliability and correctness of the supercompiler itself becomes rather actual. Can we trust the results produced by a (large and intricate) general-purpose supercompiler?
- On the other hand, it is only natural for a domain-specific supercompiler to accept programs in a domain-specific language (DSL) that provides domain-specific operations and control constructs whose mathematical properties may be known in advance. This domain knowledge can be hard-coded into the supercompiler, thereby increasing its power and enabling it to achieve better results at program analysis and transformation, as compared to "pure" supercompilation.
- The subject language of a domain-specific supercompiler may be very limited in its means of expression, in which case some parts of the supercompiler can be drastically simplified. For example, in some areas there is no need to deal with nested function calls in configurations. The simplifications of that kind increase the reliability of the supercompiler and make it easier to prove its correctness by formal methods (as was shown by Krustev [15]).
- The implementation of a domain-specific supercompiler may be very cheap if it is done on the basis of prefabricated components (for example, by means of the MRSC toolkit [13,14]), so that the costs of implementation can be reduced by an order of magnitude, as compared to implementations of general-purpose supercompilers.

2 Analyzing the behavior of systems by means of supercompilation

One of the approaches to the analysis of systems consists in representing systems by programs. Thus the task of analyzing the behavior of a system is reduced to the task of inferring and analyzing the properties of a program p.

The program p, modeling the original system, may in turn be analyzed using *the transformational approach*, in which case p is transformed into another program p' (equivalent to p), so that some non-obvious properties of p become evident in the program p'.

For example, suppose that the original program p is complicated in structure and contains statements `return False`. Can this program return `False`? This question is not easy to answer. Suppose, however, that by transforming p we get a trivial program p' whose body consists of a single statement `return True`. Then we can immediately conclude that p' can never return `False`. Since p' is equivalent to p, it implies that p also can never return `False`.

Initial states:

$$(i, 0, 0, 0)$$

Transitions:

$$(i, e, s, m) \,|\, i \geq 1 \longrightarrow (i - 1, 0, s + e + m + 1, 0)$$
$$(i, e, s, m) \,|\, e \geq 1 \longrightarrow (i, e - 1, s, m + 1)$$
$$(i, e, s, m) \,|\, s \geq 1 \longrightarrow (i + e + s + m - 1, 1, 0, 0)$$
$$(i, e, s, m) \,|\, i \geq 1 \longrightarrow (i + e + s + m - 1, 1, 0, 0)$$

Unsafe states:

$$(i, e, s, m) \,|\, m \geq 2$$
$$(i, e, s, m) \,|\, s \geq 1 \wedge m \geq 1$$

Fig. 1: MESI protocol: a protocol model in form of a counter system

One of the applications of the transformational approach is the verification of communication protocols modeled by counter systems [1]. For instance, let us consider a model of the MESI protocol in form of a counter system that is informally described in Fig. 1.

The states of the system are represented by quadruples of natural numbers. The specification of the system includes the description of a set of *initial states* and a set of *transition rules* of the form

$$(i, e, s, m) \,|\, p \longrightarrow (i', e', s', m')$$

where i, e, s, m are variables, p is a condition on the variables which must be fulfilled for the transition to be taken, and i', e', s', m' are expressions that may contain the variables i, e, s, m.

The system is non-deterministic, as several rules may be applicable to the same state.

The specification of a protocol model also includes the description of a set of *unsafe states*. The analysis of such a protocol model is performed for the purpose of solving the *reachability problem*: in order to prove that unsafe states are not reachable from the initial states.

As was shown by Leuschel and Lehmann [21,18,19,16], reachability problems for transition systems of that kind can be solved by program specialization techniques. The system to be analyzed can be specified by a program in a domain-specific language (DSL) [21]. The DSL program is then transformed into a Prolog program by means of a classic partial evaluator LOGEN [5,17] by using the first Futamura projection [2]. The Prolog program thus obtained is then transformed by means of ECCE [22,20], a more sophisticated specializer, whose internal workings are similar to those of supercompilers.

Lisitsa and Nemytykh [24,25,26] succeeded in verification of a number of communication protocols by means of the supercompiler SCP4 [29,28,27]. The input language of SCP4 is Refal, a first-order functional language developed by Turchin [36]. SCP4 is a descendant of earlier supercompilers for Refal [35,36,39,37,38].

According to the approach by Lisitsa and Nemytykh, protocol models are represented as Refal programs. For instance, the MESI protocol [1,27,23] is modeled by the Refal program in Fig. 2. The program is written in such a way that, if an unsafe state is reached, it returns the symbol `False` and terminates.

The supercompiler SCP4 takes this program as input and produces the residual program shown in Fig. 3, which contains no occurrences of the symbol `False`. This suggests the conclusion that the residual program is unable to return `False`. However, strictly speaking, this argument is not sufficient in the case of a dynamically typed language (like Lisp, Scheme and Refal): a program can still return `False`, even if `False` does not appear in the text of the program. Namely, the program may receive `False` via its input data and then transfer it to the output. And, indeed, "engineering solutions" of that kind are extremely popular with hackers as a means of attacking web-applications [34]. Fortunately, there exist relatively simple data flow analysis techniques that are able to compute an upper approximation to the set of results that can be produced by a function, even for dynamically-typed languages [4], and which are able to cope with Refal programs like that in Fig. 3.

```
*$MST_FROM_ENTRY;
*$STRATEGY Applicative;
*$LENGTH 0;

$ENTRY Go {e.A (e.I) =
  <Loop (e.A) (Invalid e.I)(Modified )(Shared )(Exclusive ) >;}

Loop {
  () (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    <Result (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4)>;
  (s.A e.A) (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    <Loop (e.A)
    <RandomAction s.A
    (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4)>>;
}

RandomAction {
* rh Trivial
* rm
A (Invalid s.1 e.1) (Modified e.2) (Shared e.3) (Exclusive e.4) =
   (Invalid e.1)  (Modified )  (Shared s.1 e.2 e.3 e.4 ) (Exclusive );
* wh1 Trivial
*wh2
B (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive s.4 e.4) =
  (Invalid e.1)(Modified s.4  e.2)(Shared e.3)(Exclusive e.4);
* wh3
C (Invalid e.1)(Modified e.2)(Shared s.3 e.3)(Exclusive e.4) =
  (Invalid e.4 e.3 e.2 e.1)(Modified  )(Shared )(Exclusive s.3);
* wm
D (Invalid s.1 e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
  (Invalid e.4 e.3 e.2 e.1)(Modified  )(Shared )(Exclusive s.1);
}

Result{
(Invalid e.1)(Modified s.2 e.2)(Shared s.3 e.3)(Exclusive e.4) = False;
(Invalid e.1)(Modified s.21 s.22 e.2)(Shared e.3)(Exclusive e.4) = False;

(Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) = True;
}
```

Fig. 2: MESI protocol: a protocol model in form of a Refal program

```
* InputFormat: <Go e.41 >
$ENTRY Go {
 (e.101 )  = True ;
 A e.41 (s.103 e.101 )  = <F24 (e.41 ) (e.101 ) s.103 > ;
 D e.41 (s.104 e.101 )  = <F35 (e.41 ) (e.101 ) s.104 > ;
}

* InputFormat: <F35 (e.109 ) (e.110 ) s.111 e.112 >
F35 {
 () (e.110 ) s.111 e.112  = True ;
 (A e.109 ) (e.110 ) s.111 s.118 e.112  =
    <F24 (e.109 ) (e.112 e.110 ) s.118 s.111 > ;
 (A e.109 ) (s.119 e.110 ) s.111  = <F24 (e.109 ) (e.110 ) s.119 s.111 >;
 (B ) (e.110 ) s.111 e.112  = True ;
 (B A e.109 ) (e.110 ) s.111 s.125 e.112  =
    <F24 (e.109 ) (e.112 e.110 ) s.125 s.111 > ;
 (B A e.109 ) (s.126 e.110 ) s.111  =
    <F24 (e.109 ) (e.110 ) s.126 s.111> ;
 (B D e.109 ) (e.110 ) s.111 s.127 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.127 > ;
 (B D e.109 ) (s.128 e.110 ) s.111  =
    <F35 (e.109 ) (s.111 e.110 ) s.128> ;
 (D e.109 ) (e.110 ) s.111 s.120 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.120 > ;
 (D e.109 ) (s.121 e.110 ) s.111  = <F35 (e.109 ) (s.111 e.110 ) s.121 >;
}

* InputFormat: <F24 (e.109 ) (e.110 ) s.111 e.112 >
F24 {
 () (e.110 ) s.111 e.112  = True ;
 (A e.109 ) (s.114 e.110 ) s.111 e.112  =
    <F24 (e.109 ) (e.110 ) s.114 s.111 e.112 > ;
 (C e.109 ) (e.110 ) s.111 e.112  =
    <F35 (e.109 ) (e.110 ) s.111 e.112 >;
 (D e.109 ) (s.115 e.110 ) s.111 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.115 > ;
}
```

Fig. 3: MESI protocol: the residual Refal program.

3 Domain-specific supercompilation as a means of analysis

3.1 Drawbacks of general-purpose supercompilation

An obvious advantage of general-purpose supercompilation is just its being general-purpose. Upon having designed and implemented a general-purpose supercompiler, we can apply it to various problems again and again, in theory, without any extra effort. However, there are some disadvantages associated with general-purpose supercompilation. As an example, let us consider the use of the specializer SCP4 for the analysis of counter systems [28,24,25,26], in which case the tasks for the supercompiler are formulated as Refal programs [27,23]. This causes the following inconveniences.

- Natural numbers in input programs are represented by strings of star symbols, and their addition by string concatenation. This representation is used in order to take into account the behavior of some general-purpose algorithms embedded in SCP4 (the whistle, the generalization), which know nothing about the specifics of counter systems. Thus, the representation of data has to conform to subtle details of the internal machinery of SCP4, rather than comply with the problem domain.
- The programs modeling counter systems have to be supplemented with some directions (in form of comments) for SCP4, which control some aspects of its behavior. In this way SCP4 is given certain information about the problem domain, and without such directions, residual programs produced by SCP4 would not possess desirable properties. Unfortunately, in order to be able to give right directions to SCP4, the user needs to understand its internals.
- There remains the following question: to what extent can we trust the results of the verification of counter systems, obtained with the aid of SCP4? The internals of SCP4 are complicated and the source code is big. Thus the problem of verifying SCP4 itself seems to be intractable.

3.2 Domain-specific algorithms of supercompilation

Which techniques and devices embedded into SCP4 are really essential for the analysis of counter systems? This question was investigated by Klimov who has developed a specialized supercompilation algorithm that was proven to be correct, always terminating, and able to solve reachability problems for a certain class of counter systems [6,7,10,8].

It was found that, in the case of counter systems, supercompilation can be simplified in the following ways.

- The structure of configurations is simpler, as compared to the case of classic supercompilation for functional languages.
  - There are no nested function calls.
  - There are no multiple occurrences of variables.

  - A configuration is a tuple, all configurations consisting of a fixed number of components.
  - A component of a configuration is either a natural number n, or the symbol ω (a wildcard, representing an arbitrary natural number).
- The termination of the supercompilation algorithm is ensured by means of a very simple generalization algorithm: if a component of a configuration is a natural number n, and $n \geq l$, where l is a constant given to the supercompiler as one of its input parameters, then n must be replaced with ω (and in this way the configuration is generalized). It can be easily seen that, given an l, the set of all possible configurations is finite.

3.3 Domain-specific supercompilers for domain-specific languages

The domain-specific supercompilation algorithm developed by Klimov [6,7,10,8] turned out to be easy to implement with the aid of the MRSC toolkit [13,14]. The simplicity of the implementation is due to the following.

- We have only to implement a simplified supercompilation algorithm for a domain-specific language, rather than a sophisticated general-purpose algorithm for a general-purpose language.
- The MRSC toolkit is based on the language Scala that provides powerful means for implementing embedded DSLs. The implementations can be based either on interpretation (shallow embedding) or on compilation (deep embedding).
- The MRSC toolkit provides prefabricated components for the construction of graphs of configurations (by adding/removing graph nodes), for manipulating sets of graphs and pretty-printing graphs. When implementing a supercompiler, it is only necessary to implement the parts that depend on the subject language and on the structure of configurations.

When we develop a domain-specific supercompiler, it seems logical for its subject language also to be domain-specific, rather than general-purpose.

In this case the formulations of problems that are submitted to the supercompiler can be concise and natural, since the programs written in the subject DSL may be very close to the informal formulations of these problems. For instance, consider the 3 specifications of the MESI protocol: the informal one (Fig. 1), the one in form of a Refal program (Fig. 2), and the one written in a domain-specific language (Fig. 4).

A protocol model encoded as a DSL program is, in terms of Scala, an object implementing the trait `Protocol` (Fig. 5). Thus this program is not a first-order value (as is implicitly assumed in the classic formulation of the Futamura projections [2]), but rather is a mixture of first-order values (numbers, lists) and higher-order values (functions). This approach is close to the DSL implementation technique known as "shallow embedding".

By supercompiling the model of the MESI protocol, we obtain the graph of configurations shown in Fig. 6.

```
object MESI extends Protocol {
  val start: Conf = List(Omega, 0, 0, 0)
  val rules: List[TransitionRule] = List(
    {case List(i, e, s, m) if i>=1 => List(i-1, 0, s+e+m+1, 0)},
    {case List(i, e, s, m) if e>=1 => List(i, e-1, s, m+1)},
    {case List(i, e, s, m) if s>=1 => List(i+e+s+m-1, 1, 0, 0)},
    {case List(i, e, s, m) if i>=1 => List(i+e+s+m-1, 1, 0, 0)})
  def unsafe(c: Conf) = c match {
    case List(i, e, s, m) if m>=2 => true
    case List(i, e, s, m) if s>=1 && m>=1 => true
    case _ => false
  }
}
```

Fig. 4: MESI protocol: a protocol model in form of a DSL program

```
package object counters {
  type Conf = List[Expr]
  type TransitionRule = PartialFunction[Conf, Conf]
  ...
}

sealed trait Expr { ... }

trait Protocol {
  val start: Conf
  val rules: List[TransitionRule]
  def unsafe(c: Conf): Boolean
}
```

Fig. 5: DSL for specifying counter systems: the skeleton of its implementation in Scala

4 Using multi-result supercompilation for finding short proofs

When analyzing a transition system, a graph of configurations produced by supercompilation describes an upper approximation of the set of reachable states. This graph can be transformed in a human-readable proof that any reachable state satisfy some requirements (or, in other words, cannot be "unsafe").

The smaller the graph the easier it is to understand, and the shorter is the proof that can be extracted from this graph. However, a traditional single-result supercompiler returns a single graph that may not be the smallest one.

However, a multi-result supercompiler returns a set of graphs, rather than a single graph. Thus the set of graphs can be filtered, in order to select "the best"

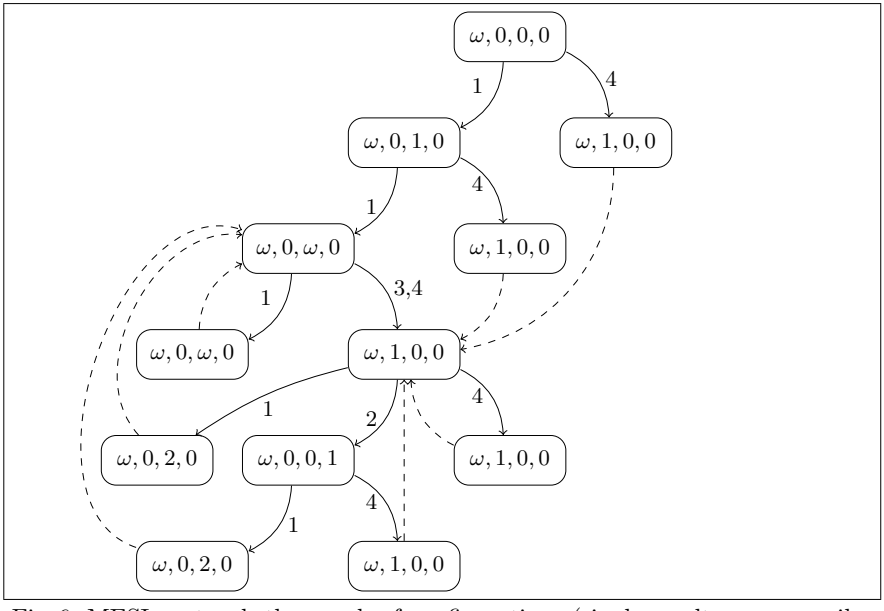


Fig. 6: MESI protocol: the graph of configurations (single-result supercompilation)

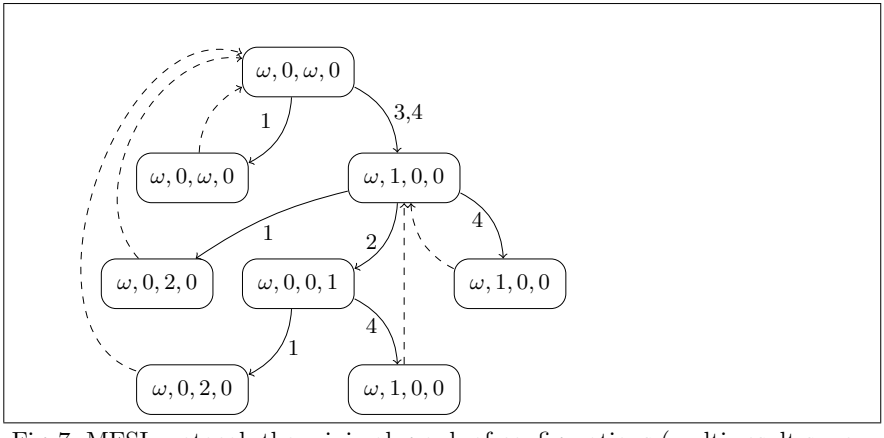


Fig. 7: MESI protocol: the minimal graph of configurations (multi-result supercompilation)

ones. In the simplest case, "the best" means "the smallest", although the graphs can be filtered according to other criteria (for example, we may select the graphs that are, in a sense, "well-structured", to transform them into "well-structured" proofs).

For example, the graph produced for the MESI protocol (see Fig. 6) by single-result positive supercompilation [32,33] contains 12 nodes, while, by filtering the set of graphs produced by multi-result supercompilation, we can find the graph shown in Fig. 7, which only contains 8 nodes.

The point is that single-result supercompilers, especially those meant for program optimization try to avoid the generalization of configurations by all means. This strategy is reasonable and natural in the case of optimizing supercompilation, but it is unlikely to produce minimal graphs. In the above example, single-result supercompilation starts from the configuration $(\omega, 0, 0, 0)$ and, after a while, comes to the configuration $(\omega, 0, \omega, 0)$, which is more general than $(\omega, 0, 0, 0)$.

However, multi-result supercompilation, by "a sudden flash of inspiration", starts with generalizing the initial configuration. From the viewpoint of optimizing supercompilation, this action appears to be strange and pointless. But it leads to producing the graph shown in Fig. 7, which is a *subgraph* of the graph in Fig. 6

Another interesting point is that in the case of single-result supercompilation the whistle and the generalization algorithm are tightly coupled, since generalization is performed at the moments when the whistle blows, in order to ensure termination of supercompilation. For this reason, the whistle and the generalization algorithm, to be consistent, have to be developed together. In the case of multi-result supercompilation, however, the whistle and generalization are completely decoupled. In particular, configurations can be generalized at any moment, even if the whistle does not regard the situation as dangerous. As a result, a multi-result supercompiler can find graphs that are not discovered by single-result supercompilation.

As an example, let us consider the verification of the MOESI protocol (Fig. 8). The graph produced by single-result supercompilation (Fig. 9) contains 20 nodes, while the graph discovered by multi-result supercompilation (Fig. 10) contains 8 nodes only.

This is achieved due to a "brilliant insight" of multi-result supercompilation that the initial configuration $(\omega, 0, 0, 0, 0)$ can be immediately generalized to the configuration $(\omega, 0, \omega, 0, \omega)$. This leads to an 8-node graph that is *not contained* as a subgraph in the 20-node graph produced by single-result supercompilation. Note that the configuration $(\omega, 0, \omega, 0, \omega)$ *does not appear* in the 20-node graph, and, in general, the structure of the graphs in Fig. 9 and Fig.10 is completely different.

It should be noted that there exists a domain-specific supercompilation algorithm for counter systems (developed by Klimov [10]) that, in some cases, is able to reduce the number of nodes in the graphs, because, as compared to general-purpose optimizing supercompilers, it generalizes configurations more energetically. For instance, for the MESI protocol, it generates the same graph (Fig. 7), as that produced by multi-result supercompilation.

The idea of Klimov's algorithm [10] is the following. Suppose, in the process of supercompilation there appears a configuration c, such that c is an instance

```
case object MOESI extends Protocol {
  val start: Conf = List(Omega, 0, 0, 0, 0)
  val rules: List[TransitionRule] =
    List({ // rm
      case List(i, m, s, e, o) if i>=1 =>
        List(i-1, 0, s+e+1, 0, o+m)
    }, { // wh2
      case List(i, m, s, e, o) if e>=1 =>
        List(i, m+1, s, e-1, o)
    }, { // wh3
      case List(i, m, s, e, o) if s+o>=1 =>
        List(i+e+s+m+o-1, 0, 0, 1, 0)
    }, { // wm
      case List(i, m, s, e, o) if i>=1 =>
        List(i+e+s+m+o-1, 0, 0, 1, 0)
    })

  def unsafe(c: Conf) = c match {
    case List(i, m, s, e, o) if m>=1 && e+s+o>=1 => true
    case List(i, m, s, e, o) if m>=2 => true
    case List(i, m, s, e, o) if e>=2 => true
    case _ => false
  }
}
```

Fig. 8: MOESI protocol: a protocol model as a DSL program

of a configuration c' that is already present in the graph. Then c has to be generalized to c'.

Unfortunately, this algorithm is not always successful in generating minimal graphs. For example, in the case of the MOESI protocol, multi-result supercompilation finds such configurations that are not appear in the process of classic positive supercompilation [32,33].

The table in Fig. 11 compares the results of verifying 13 communication protocols. The column SC shows the number of nodes in the graphs produced by classic single-result positive supercompilation, and the column MRSC shows the number of nodes in the graphs produced by straightforward multi-result supercompilation derived from positive supercompilation [32,33] according to the scheme described in [14]. It is evident that, practically always, multi-result supercompilation is able to find graphs of smaller size than those produced by single-result supercompilation.

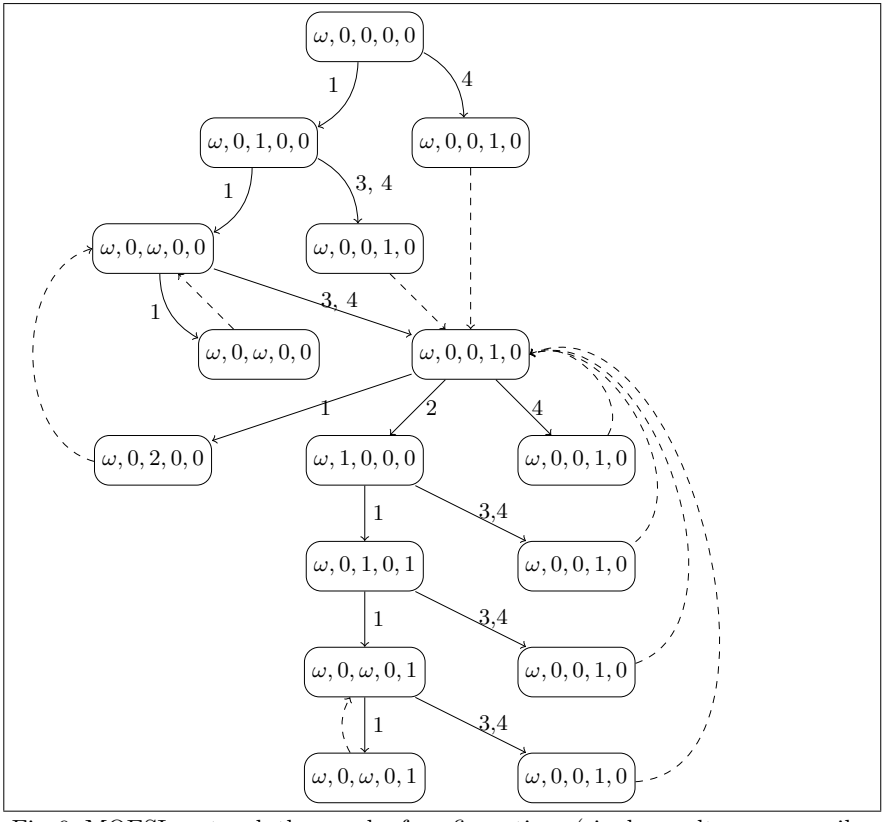


Fig. 9: MOESI protocol: the graph of configurations (single-result supercompilation)

5 Domain-specific residualization of the graphs of configurations

Traditionally, general-purpose supercompilation is performed in two steps. At the first step, there is produced a finite graph of configurations. At the second step, this graph is "residualized", i.e. transformed into a residual program. For example, the supercompiler SCP4 generates residual programs in the language Refal (see Fig. 3)

When the purpose of supercompilation is the analysis of counter systems, residual programs are not executed, but analyzed to see whether they possess some desirable properties. For example, as regards the program in Fig. 3, all that matters is whether it can return `False`, or not? This can be determined either by asking a human's opinion, or, in a more rigorous way, by submitting the program to a data flow analysis algorithm [4].

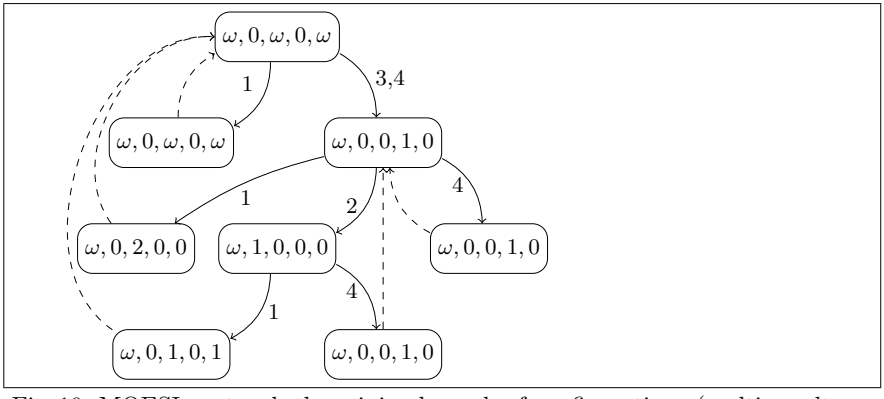


Fig. 10: MOESI protocol: the minimal graph of configurations (multi-result supercompilation)

	SC	MRSC
Synapse	11	6
MSI	8	6
MOSI	26	14
MESI	14	9
MOESI	20	9
Illinois	15	13
Berkley	17	6
Firefly	12	10
Futurebus	45	24
Xerox	22	13
Java	35	25
ReaderWriter	48	9
DataRace	9	5

Fig. 11: Single-result vs. multi-result supercompilation: the size of proofs (represented by graphs of configurations)

However, when using supercompilation for the analysis of counter systems, we can take an easier way: it turns out that graphs of configurations are easier to analyze, than residual programs. Thus, we can dispense with the generation of residual programs for the purposes of separating the good outcomes of supercompilation from the bad ones. Moreover, upon selecting a graph with desirable properties, instead of generating a residual program in a programming language, we can transform the graph into a script for a well-known proof assistant [16], in order to verify the results obtained by supercompilation.

In particular, we have implemented a domain-specific supercompiler that transforms graphs of configurations into scripts for the proof assistant Isabelle [30]. A script thus produced specifies the reachability problem for a communi-

```
theory mesi
imports Main
begin

inductive mesi :: "(nat * nat * nat * nat) => bool" where
  "mesi (i, 0, 0, 0)" |
  "mesi (Suc i, e, s, m) ==> mesi (i, 0, Suc (s + e + m), 0)" |
  "mesi (i, Suc e, s, m) ==> mesi (i, e, s, Suc m)" |
  "mesi (i, e, Suc s, m) ==> mesi (i + e + s + m, Suc 0, 0, 0)" |
  "mesi (Suc i, e, s, m) ==> mesi (i + e + s + m, Suc 0, 0, 0)"

inductive unsafe :: "(nat * nat * nat * nat) => bool" where
  "unsafe (i, e, s, Suc (Suc m))" |
  "unsafe (i, e, Suc s, Suc m)"


inductive mesi' :: "(nat * nat * nat * nat) => bool" where
  "mesi'(_, Suc 0, 0, 0)" |
  "mesi'(_, 0, 0, Suc 0)" |
  "mesi'(_, 0, _, 0)"

lemma inclusion: "mesi c ==> mesi' c"
  apply(erule mesi.induct)
  apply(erule mesi'.cases | simp add: mesi'.intros)+
done

lemma safety: "mesi' c ==> ~unsafe c"
  apply(erule mesi'.cases)
  apply(erule unsafe.cases | auto)+
done

theorem valid: "mesi c ==> ~unsafe c"
  apply(insert inclusion safety, simp)
done

end
```

Fig. 12: MESI protocol: the script for the proof assistant Isabelle produced by the domain-specific supercompiler.

cation protocol and, in addition, a number of tactics that instruct Isabelle how to formally prove that all reachable states are safe.

For example, in the case of the MESI protocol, there is produced the script shown in Fig. 12. The script comprises the following parts.

- Inductive definitions of the predicate `mesi`, specifying the set of reachable states, and the predicate `unsafe`, specifying the set of unsafe states (`unsafe`), which are the same (modulo notation) as in the source DSL program.
- An inductive definition of the predicate `mesi'`, specifying a set of states that is an upper approximation to the set specified by `mesi`. This definition (modulo notation) enumerates configurations appearing in the graph in Fig. 7. In order to reduce the size of the script, there is applied a simple optimization: if the graph contains two configurations c' and c, where c' is an instance of c, then c' is not included into the definition of the predicate `mesi'`. The definition of `mesi'` is the most important (and non-trivial) part of the script.
- The lemma `inclusion`, asserting that any reachable state belongs to the set specified by `mesi'`, or, in other words, for any state c, `mesi` c implies `mesi'` c.
- The lemma `safety`, asserting that all states in the set specified by `mesi'` are safe, or, in other words, for any state c, `mesi'` c implies ¬`unsafe` c.
- The main theorem: any reachable state is safe. In other words, for all states c, `mesi` c implies ¬`unsafe` c. This trivially follows from the lemmas `inclusion` and `safety`).

The fundamental difference between the definitions of `mesi` and `mesi'` is that `mesi` is defined inductively, while the definition of `mesi'` is just an enumeration of a finite number of cases. For this reason, the lemma `safety` can be proven by tedious, yet trivial case analysis.

Thus the rôle of supercompilation in the analysis of counter systems amounts to generalizing the description of the set of reachable states in such a way that proving the safety of reachable states becomes trivial. Therefore, supercompilation can be regarded as a useful supplement to other theorem-proving and verification techniques.

6 Improving the efficiency of supercompilation by taking into account the specifics of the domain

6.1 Exploiting the mathematical properties of domain-specific operations

As was shown by Klimov [6,7,10,8], in the case of supercompilation for counter systems it is sufficient to deal with configuration of the form $(a_1, \ldots, a_n)$, whose each component a_i is either a natural number N, or the symbol ω. As regards driving, it is sufficient to deal with tests of the form either $e = N$, or $e \geq N$, where N is a natural number and e is an arithmetic expression that can only contain the operators $+$, $-$, natural numbers and ω. All necessary operations over such expressions are easy to implement in terms of the language Scala (see Fig.13).

But, if we use a general-purpose supercompiler, dealing with programs in a general-purpose language, the supercompiler does not have any knowledge about the problem domain and the operations over domain-specific data structures. For example, when the supercompiler SCP4 is used for the verification of protocols,

```
package object counters {
  ...
  implicit def intToExpr(i: Int): Expr = Num(i)
}

sealed trait Expr {
  def +(comp: Expr): Expr
  def -(comp: Expr): Expr
  def >=(i: Int): Boolean
  def ===(i: Int): Boolean
}

case class Num(i: Int) extends Expr {
  def +(comp: Expr) = comp match {
    case Omega  => Omega
    case Num(j) => Num(i + j)
  }
  def -(comp: Expr) = comp match {
    case Omega  => Omega
    case Num(j) => Num(i - j)
  }
  def ===(j: Int) = i == j
  def >=(j: Int) = i >= j
}

case object Omega extends Expr {
  def +(comp: Expr) = Omega
  def -(comp: Expr) = Omega
  def >=(comp: Int) = true
  def ===(j: Int) = true
}
```

Fig. 13: Counter systems: operations over components of configurations implemented in Scala.

natural numbers have to be encoded as strings of the star symbol, and addition of natural numbers as concatenation of strings (see Fig. 2 and 3). As a consequence, it becomes difficult (both for humans and for supercompilers) to see that a program operates on natural numbers.

6.2 Handling non-determinism in a direct way

When a supercompiler is a general-purpose one, its subject language is usually designed for writing deterministic programs. This causes some inconveniences in cases where supercompilation is used for the analysis of non-deterministic systems. If a model of a non-deterministic system has to be encoded as a deterministic program, there arises the need for using various tricks and artificial

workarounds, which, certainly, complicates the program and obscures its meaning.

Consider, for example, the model of the MESI protocol in Fig. 2, encoded as a Refal program. The entry point of the program is the function `Go` which takes 2 parameters: `e.A` and `e.I` [27,23].

```
$ENTRY Go {e.A (e.I) =
    <Loop (e.A) (Invalid e.I)(Modified )(Shared )(Exclusive ) >;}
```

The parameter `e.I` is used for building the initial state, while the parameter `e.A` has been artificially introduced in order to simulate non-determinism. Since the rules describing the transition system are not mutually exclusive, more than one rule can be applicable at the same time, and the value of the parameter `e.A` is a sequence of rule names, prescribing which rule must be applied at each step.

Unfortunately, this additional parameter, pollutes not only the source program, but also the configurations emerging during supercompilation and, finally, the residual program (see Fig. 3), thereby obscuring its meaning.

However, if a model of a non-deterministic system is encoded as a program in a non-deterministic language (see Fig. 4), then there disappears the need for using tricks and workarounds related to non-determinism. Also note that non-determinism, by itself, does not create additional problems for supercompilation, as, unlike an ordinary interpreter, a supercompiler has to consider *all* possible ways of executing a program (for a given set of initial states) [18,19].

6.3 Filtering graphs of configurations, rather than residual programs

As has been discussed in Section 2, multi-result supercompilation can be used for finding residual programs satisfying some criteria. Since a multi-result supercompiler may produce hundreds, or even thousands of residual programs, there is a need for automatic filtering of residual programs.

For example, when applying a general-purpose supercompiler for the analysis of counter systems, we need a filter for selecting residual problems that are certain not to return `False`, and such a filter can be constructed on the basis of well-known data-flow analysis algorithms [4].

In the case of domain-specific supercompilation, however, "residual programs" may not be programs in traditional sense of the word. For instance, the result produced by analyzing a counter system can be presented as a script for an automatic proof assistant (see Section 5). So the filtering of programs should be replaced with the filtering of something else.

Fortunately, it turns out that filtering of the final results of supercompilation can be replaced with filtering of graphs of configurations. Moreover, taking into account the specifics of the domain allows the process of filtering to be optimized by discarding some graphs that are in construction, without waiting for them to be completed. This can considerably reduce the amount of work performed by a multi-result supercompiler, because discarding an incomplete graph prunes the whole set of graphs that would be generated by completing the discarded graph.

		SC1	SC2	SC3	SC4	SC5
Synapse	completed	48	37	3	3	1
	pruned	0	0	0	0	2
	commands	321	252	25	25	15
MSI	completed	22	18	2	2	1
	pruned	0	0	0	0	1
	commands	122	102	15	15	12
MOSI	completed	1233	699	6	6	1
	pruned	0	0	0	0	5
	commands	19925	11476	109	109	35
MESI	completed	1627	899	6	3	1
	pruned	0	0	27	20	21
	commands	16329	9265	211	70	56
MOESI	completed	179380	60724	81	30	2
	pruned	0	0	0	24	36
	commands	2001708	711784	922	384	126
Illinois	completed	2346	1237	2	2	1
	pruned	0	0	21	17	18
	commands	48364	26636	224	74	61
Berkley	completed	3405	1463	30	30	2
	pruned	0	0	0	0	14
	commands	26618	12023	282	282	56
Firefly	completed	2503	1450	2	2	1
	pruned	0	0	2	2	3
	commands	39924	24572	47	25	21
Futurebus	completed	-	-	-	-	4
	pruned	-	-	-	-	148328
	commands	-	-	-	-	516457
Xerox	completed	317569	111122	29	29	2
	pruned	0	0	0	0	1
	commands	5718691	2031754	482	482	72
Java	completed	-	-	-	-	10
	pruned	-	-	-	-	329886
	commands	-	-	-	-	1043563
ReaderWriter	completed	892371	402136	898	898	6
	pruned	0	0	19033	19033	1170
	commands	24963661	11872211	123371	45411	3213
DataRace	completed	51	39	8	8	3
	pruned	0	0	0	0	4
	commands	360	279	57	57	31

Fig. 14: Resources consumed by different versions of the multi-result supercompiler

As regards counter systems, the specifics of the domain are the following. The predicate `unsafe` must be monotonic with respect to configurations: for all configurations c and c', such that c is an instance of c', `unsafe` c implies `unsafe` c'. Another point is that if a configuration c has appeared in a graph of configurations, it can be removed by supercompilation only by replacing c with a more general configuration c' (such that c is an instance of c'). Thus, if c is unsafe, it can only be replaced with an unsafe configuration (due to the monotonicity of the predicate `unsafe`). Therefore, if a graph contains an unsafe configuration, it can be discarded immediately, since all graphs produced by completing that graph would also contain unsafe configurations.

The detection of unsafe configurations can be performed at various places in the supercompilation algorithm, and the choice of such places bears great influence on the efficiency of multi-result supercompilation.

The next optimization, depending on the specifics of the domain, takes into account the properties of the set of all possible generalizations of a given configuration c.

Namely, all generalizations of c can be obtained by replacing some numeric components of c with ω. Thus, the configuration $(0, 0)$ can be generalized in 3 ways, to produce $(\omega, 0)$, $(0, \omega)$ and (ω, ω). Note that (ω, ω) is a generalization with respect to $(\omega, 0)$ and $(0, \omega)$.

A naïve multi-result supercompilation algorithm, when trying to rebuild a configuration c by replacing it with a more general configuration c', considers all possible generalizations of c immediately. If a generalization c' is not a maximal one, after a while, it will be, in turn, generalized. For instance, $(\omega, 0)$, and $(0, \omega)$ will be generalized to (ω, ω). Thus the same graph of configurations will be produced 3 times: by immediately generalizing $(0, 0)$ to (ω, ω), and by generalizing $(0, 0)$ to (ω, ω) in two steps, via $(\omega, 0)$, and $(0, \omega)$.

The number of graphs, considered during multi-result supercompilation, can be significantly reduced, by allowing only minimal generalization of a configuration, which can be obtained by replacing a single numeric component in a configuration with ω.

We have studied the performance of 5 variations of a supercompilation algorithm for counter systems: SC1, SC2, SC3, SC4 and SC5. Each variant differs from the previous one in that it introduces an additional optimization.

- SC1. Filtering and generation of graphs are completely decoupled. A graph is examined by the filter only after having been completed. Thus, no use is made of the knowledge about domain-specific properties of generalization (its decomposability into elementary steps) and the predicate `unsafe` (its monotonicity). This design is modular, but inefficient.
- SC2. The difference from SC1 is that, when rebuilding a configuration c, SC2 only considers the set of "minimal" generalizations (produced by replacing a single component of c with ω).
- SC3. The difference from SC2 is that the configurations produced by generalization are checked for being safe, and the unsafe ones are immediately discarded.

- SC4. The difference from SC3 is that the configurations that could be produced by driving a configuration c are checked for being safe. If one or more of the new configurations turn out to be unsafe, driving is not performed for c.
- SC5. The difference from SC4 is that the graphs that are too large are discarded, without completing them. Namely, the current graph is discarded if there is a complete graph that has been constructed earlier, and whose size is smaller than that of the current graph. (Note that, due to the optimizations introduced in SC2, SC3 and SC4, all configurations in completed graphs are guaranteed to be safe.)

The optimization introduced in SC5 is typical for algorithms in the field of artificial intelligence, where it is known as "pruning" [31].

The table in Fig. 14 shows the resources consumed by the 5 versions of the supercompiler while verifying 13 communication protocols. For each protocol, the row *completed* shows the number of completed graphs that have been produced (with possible repetitions), the row *pruned* shows the number of discarded incomplete graphs, and the row *commands* shows the number of graph building steps that have been performed during supercompilation.

In the case of the protocols Futurebus and Java, the data are only given for the version SC5, as the resource consumption by the other versions of the supercompiler turned out to be too high, for which reason data were not obtained.

The data demonstrate that the amount of resources consumed by multi-result supercompilation can be drastically reduced by taking into account the specifics of the problem domain.

7 Conclusions

Multi-result supercompilation is not a theoretical curiosity, but rather a workhorse that, when exploited in a reasonable way, is able to produce results of practical value.

- The use of multi-result supercompilation in the field of the analysis and verification of transition systems improves the understandability of the results, by considering various versions of the analysis and selecting the best ones.
- The use of multi-result supercompilation allows the whistle and the algorithm of generalization to be completely decoupled, thereby simplifying the structure of the supercompiler. This, in turn, makes it easier to ensure the correctness of the supercompiler.

The usefulness of domain-specific supercompilation is due to the following.

- The tasks for a domain-specific supercompiler can be written in a domain-specific language that is better at taking into account the specifics of the problem domain, than a general-purpose language. (For example, this DSL may be non-deterministic, or provide domain-specific data types and operations.)

- In the case of a domain-specific supercompiler, the machinery of supercompilation can be simplified, since, in a particular domain, some complexities of general-purpose supercompilation may be of little usefulness.
- The efficiency of multi-result supercompilation can be improved by early discarding of unsatisfactory variants of supercompilation.
- The MRSC toolkit allows domain-specific multi-result supercompilers to be manufactured at low cost, making them a budget solution, rather than a luxury.

Thus, the combination of domain-specific and multi-result supercompilation produces a synergistic effect: generating multiple results gives the opportunity to select the best solutions to a problem, while taking into account the specifics of the problem domain reduces the amount of resources consumed by multi-result supercompilation.

Acknowledgements

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

References

1. G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Form. Methods Syst. Des.*, 23:257–301, November 2003.
2. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
3. N. D. Jones. The essence of program transformation by partial evaluation and driving. In *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, PSI '99, pages 62–79, London, UK, UK, 2000. Springer-Verlag.
4. N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci.*, 375(1-3):120–136, 2007.
5. J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 238–262. Springer, 1996.
6. A. Klimov. An approach to supercompilation for object-oriented languages: the Java supercompiler case study. In *First International Workshop on Metacomputation in Russia*, 2008.
7. A. V. Klimov. A Java supercompiler and its application to verification of cache-coherence protocols. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2009.
8. A. V. Klimov. Multi-result supercompilation in action: Solving coverability problem for monotonic counter systems by gradual specialization. In *International Workshop on Program Understanding, PU 2011, Novososedovo, Russia, July 2–5, 2011*, pages 25–32. Ershov Institute of Informatics Systems, Novosibirsk, 2011.

9. A. V. Klimov. Yet another algorithm for solving coverability problem for monotonic counter systems. In V. Nepomnyaschy and V. Sokolov, editors, *Second Workshop "Program Semantics, Specification and Verification: Theory and Applications", PSSV 2011, St. Petersburg, Russia, June 12–13, 2011*, pages 59–67. Yaroslavl State University, 2011.
10. A. V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In E. Clarke, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011, Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011*, volume 7162 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2012.
11. A. V. Klimov, I. G. Klyuchnikov, and S. A. Romanenko. Implementing a domain-specific multi-result supercompiler by means of the MRSC toolkit. Preprint 24, Keldysh Institute of Applied Mathematics, 2012.
12. I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.
13. I. Klyuchnikov and S. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In *Ershov Informatics Conference*, volume 7162 of *LNCS*, pages 210–226, 2012.
14. I. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute of Applied Mathematics, 2011.
15. D. Krustev. A simple supercompiler formally verified in Coq. In *Second International Workshop on Metacomputation in Russia*, 2010.
16. H. Lehmann and M. Leuschel. Inductive theorem proving by program specialisation: Generating proofs for Isabelle using Ecce. In M. Bruynooghe, editor, *LOPSTR*, volume 3018 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.
17. M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. *Electr. Notes Theor. Comput. Sci.*, 30(2):157–162, 1999.
18. M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 101–115, London, UK, 2000. Springer-Verlag.
19. M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 268–279, New York, NY, USA, 2000. ACM.
20. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Trans. Program. Lang. Syst.*, 20:208–258, January 1998.
21. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In *Selected papers from the 9th International Workshop on Logic Programming Synthesis and Transformation*, pages 62–81, London, UK, 2000. Springer-Verlag.
22. M. Leuschel and D. D. Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. D. Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 1996.
23. A. Lisitsa. Verification of MESI cache coherence protocol. `http://www.csc.liv.ac.uk/~alexei/VeriSuper/node5.html`.

24. A. Lisitsa and A. P. Nemytykh. Towards verification via supercompilation. *Computer Software and Applications Conference, Annual International*, 2:9–10, 2005.

25. A. Lisitsa and A. P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.

26. A. Lisitsa and A. P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.

27. A. Nemytykh. SCP4 : Verification of protocols. `http://refal.botik.ru/protocols/`.

28. A. P. Nemytykh. The supercompiler SCP4: General structure. In M. Broy and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 162–170. Springer, 2003.

29. A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: Experiments with a supercompiler. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 249–260, 1996.

30. T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

31. D. Poole and A. K. Mackworth. *Artificial Intelligence - Foundations of Computational Agents*. Cambridge University Press, 2010.

32. M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Dept. of Computer Science, University of Copenhagen, 1994.

33. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

34. Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 372–382, New York, NY, USA, 2006. ACM.

35. V. F. Turchin. A supercompiler system based on the language refal. *SIGPLAN Not.*, 14(2):46–54, 1979.

36. V. F. Turchin. *The Language Refal: The Theory of Compilation and Metasystem Analysis*. Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.

37. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

38. V. F. Turchin. Supercompilation: Techniques and results. In *Perspectives of System Informatics*, volume 1181 of *LNCS*. Springer, 1996.

39. V. F. Turchin, R. M. Nirenberg, and D. V. Turchin. Experiments with a supercompiler. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 47–55, New York, NY, USA, 1982. ACM.

A An implementation

Here we show the source code of the multi-result supercompiler for counter systems implemented by means of the MRSC toolkit. More detailed explanations about the code may be found in [11].

A.1 Operations over configurations

First of all, the supercompiler has to perform the following operations over configurations: testing whether a configuration c_1 is an instance of a configuration c_2, and enumerating all possible generalizations of a configuration c. An implementation of these operations is shown in Fig. 15.

```
package mrsc.counters

object Conf {
  def instanceOf(c1: Conf, c2: Conf): Boolean =
    (c1, c2).zipped.forall((e1, e2) => e1 == e2 || e2 == Omega)

  def gens(c: Conf) =
    product(c map genExpr) - c

  def oneStepGens(c: Conf): List[Conf] =
    for (i <- List.range(0, c.size) if c(i) != Omega)
      yield c.updated(i, Omega)

  def product[T](zs: List[List[T]]): List[List[T]] = zs match {
    case Nil       => List(List())
    case x :: xs => for (y <- x; ys <- product(xs)) yield y :: ys
  }

  private def genExpr(c: Expr): List[Expr] = c match {
    case Omega            => List(Omega)
    case Num(i) if i >= 0 => List(Omega, Num(i))
    case v                => List(v)
  }
}
```

Fig. 15: Operations over configurations: testing for instances and building generalizations

The function `instanceOf` tests whether a configuration `c1` is an instance of a configuration `c2`.

The function `genExpr` generates all possible generalization of an expression (which is a component of a configuration). Note that the original expression is

```
trait GraphRewriteRules[C, D] {
  type N = SNode[C, D]
  type G = SGraph[C, D]
  type S = GraphRewriteStep[C, D]
  def steps(g: G): List[S]
}

case class GraphGenerator[C, D]
  (rules: GraphRewriteRules[C, D], conf: C)
  extends Iterator[SGraph[C, D]] { ... }
```

Fig. 16: MRSC "middleware" for supercompiler construction

included into the set of generalization. The set of generalization of the symbol ω contains only the symbol ω, while the set of generalizations of a number N consists of two elements: N and ω.

The function `gens` generates the set of all possible generalizations of a configuration c. Note that c is not included into this set.

The function `oneStepGens` generates the set of all generalizations of a configurations c that can be produced by generalizing a single component of c. This function will be used in the optimized version of the supercompiler shown in Fig. 20.

A.2 Graph builder

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad \neg dangerous(g, \beta) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad c' \in rebuildings(c)}{g \to rebuild(g, \beta, c')}$$

Notation:

- g – a current graph of configurations
- β – a current node in a graph of configurations
- c – a configuration in a current node β

Fig. 17: Multi-result supercompilation specified by rewrite rules

Technically, a supercompiler written using MRSC is based upon two components shown in Fig. 16: `GraphRewriteRules` and `GraphGenerator` [14].

The trait `GraphRewriteRules` declares the method `steps`, which is used in the main loop of supercompilation for obtaining all graphs that can be derived from a given incomplete graph g by applying the rewrite rules Fold, Drive and Rebuild [14] shown in Fig. 17. Namely, `steps(`g`)` returns a list of "graph rewrite steps" [14]. Then the graph generator applies each of these "steps" to the graph g to produce the collection of the descendants of g.

A concrete supercompiler is required to provide an implementation for the method `steps`. The class `GraphGenerator`, by contrast, is a ready-to-use component: it is a constituent part of any supercompiler built on top of MRSC.

```
package mrsc.counters

class MRCountersRules(protocol: Protocol, l: Int)
  extends GraphRewriteRules[Conf, Unit] {

  override def steps(g: G): List[S] =
    fold(g) match {
      case None    => rebuild(g) ++ drive(g)
      case Some(s) => List(s)
    }

  def fold(g: G): Option[S] = {
    val c = g.current.conf
    for (n <- g.completeNodes.find(n => instanceOf(c, n.conf)))
      yield FoldStep(n.sPath)
  }

  def drive(g: G): List[S] =
    if (dangerous(g)) List()
    else List(AddChildNodesStep(next(g.current.conf)))

  def rebuild(g: G): List[S] =
    for (c <- gens(g.current.conf))
      yield RebuildStep(c): S

  def dangerous(g: G): Boolean =
    g.current.conf exists
      { case Num(i) => i >= l; case Omega => false }

  def next(c: Conf): List[(Conf, Unit)] =
    for (Some(c) <- protocol.rules.map(_.lift(c)))
      yield (c, ())
}
```

Fig. 18: Graph rewrite rules: an implementation for counter systems

In the case of supercompilation for counter systems the method `steps` can be straightforwardly implemented as shown in Fig. 18.

The methods `fold`, `drive` and `rebuild` correspond to the rewrite rules Fold, Drive and Rebuild [14]. Since the rewrite rules are independent from each other, the body of the method (steps) could have been defined in the following trivial way:

```
fold(g) ++ rebuild(g) ++ drive(g)
```

However, we have preferred to slightly optimize the implementation by taking into account that the rule Fold is mutually exclusive with the rules Drive and Rebuild. Another subtle point is that, in general, the rule Fold is non-deterministic, because the current configuration may be foldable to several configurations in the graph. Thus, the rule Fold may be applicable in zero, one or more ways. However, in the case of counter systems, all variants of folding are equally good. For this reason, in the implementation in Fig. 18, the method `fold` returns no more than one variant of folding, the type of the results being `Option[S]`, rather than `List[S]`. And the rules Drive and Rebuild are only applied if `fold` returns zero results.

The implementations of the methods `fold` and `rebuild` are straightforward.

The method `dangerous` implements the whistle suggested by Klimov [10,9]: a configuration is considered as "dangerous" if it contains a number N, such that $N \geq l$, where l is a constant given to the supercompiler as one of its input parameters.

The implementation of the method `drive` uses an auxiliary method `next`, which tries to apply all transition rules to a configuration c. If a rule is applicable, it returns a configuration c', in which case the pair $(c', ())$ is included in the list returned by `next`. In general, this pair has the form c', d, where c' is the new configuration and d the label for the edge entering the node containing the configuration c'. But, in the case of counter systems, edges need not be labeled, for which reason we put the placeholder () in the second component of the pair.

A.3 Optimizations

Fig. 19 shows the supercompiler for counter systems that has been produced from the supercompiler in Fig. 18 by implementing the aforementioned optimizations. Technically, the improved supercompiler is implemented as the class `FastMRCountersRules`, which is a subclass of `MRCountersRules`.

The main loop of the optimized supercompiler is shown in Fig. 20. Complete graphs are produced by the iterator `graphs` by demand. Since the goal is to find a graph of minimum size, the variable `minGraph` contains the smallest of the graphs that have been encountered.

Now let us consider the internals of the class `FastMRCountersRules`.

The variable `maxSize` holds the maximum size of graphs that are worth considering: if the supercompiler encounters a graph whose size exceeds `maxSize`, this graph is discarded (see the definition of the method `steps`).

```
package mrsc.counters

class FastMRCountersRules(protocol: Protocol, l: Int)
  extends MRCountersRules(protocol, l) {

  var maxSize: Int = Int.MaxValue

  override def drive(g: G): List[S] =
    for (AddChildNodesStep(ns) <- super.drive(g)
         if ns.forall(c =>!protocol.unsafe(c._1)))
      yield AddChildNodesStep(ns)

  override def rebuild(g: G): List[S] =
    for (c <- oneStepGens(g.current.conf) if !protocol.unsafe(c))
      yield RebuildStep(c): S

  override def steps(g: G): List[S] =
    if (protocol.unsafe(g.current.conf) || size(g) > maxSize)
      List()
    else
      super.steps(g)

  private def size(g: G) =
    g.completeNodes.size + g.incompleteLeaves.size
}
```

Fig. 19: Graph rewrite rules: an optimized implementation for counter systems

```
    val rules = new FastMRCountersRules(protocol, l)
    val graphs = GraphGenerator(rules, protocol.start)

    var minGraph: SGraph[Conf, Unit] = null
    for (graph <- graphs) {
      val size = graphSize(graph)
      if (size < rules.maxSize) {
        minGraph = graph
        rules.maxSize = size
      }
    }
```

Fig. 20: Optimized implementation of the main loop of multi-result supercompilation

The method `rebuild` is redefined: now, instead of considering all possible generalization (produced by the method `gens`), it only considers one-step generalizations (produced by the method `oneStepGens`).

All other modifications are related to detecting unsafe configurations: the goal is to detect unsafe configurations as soon as possible. This is achieved by applying the predicate `unsafe` to configurations at several places.

Formalizing and Implementing Multi-Result Supercompilation*

Ilya G. Klyuchnikov and Sergei A. Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Abstract. The paper explains the principles of multi-result supercompilation. We introduce a formalism for representing supercompilation algorithms as rewriting rules for graphs of configurations. Some low-level technical details related to the implementation of multi-result supercompilation in MRSC are discussed. In particular, we consider the advantages of using spaghetti stacks for representing graphs of configurations.

1 Introduction

1.1 Growing variety in the field of supercompilation

Supercompilation is a program manipulation technique that was originally introduced by V. Turchin in terms of the programming language Refal (a first-order applicative functional language) [36], for which reason the first supercompilers were designed and developed for the language Refal [34,38,25].

It might create the impression that supercompilation is a specific technique only applicable to Refal (and Refal-like languages).

Further development of supercompilation lead to a more abstract reformulation of supercompilation and to a better understanding of which details of the original formulation were Refal-specific and which ones were universal and applicable to other programming languages [28,32,6]. It particular, it was shown that supercompilation is as well applicable to non-functional programming languages (imperative and object-oriented ones) [9].

As a result, the distinction between "supercompilation" and a "supercompiler" was realized. Supercompilation is a general *method*, while a supercompiler is a program *transformer* (based on the principles of supercompilation). Thus the transition from the idea of supercompilation to a specific supercompiler involves making a number of decisions. Namely, we have to:

– Select an *input language*: programs in this language will be dealt with by the supercompiler. (Note that the supercompiler may produce programs in another language, in which case we have as well to select an *output language*.)

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

- Choose, for the selected input language, some kind of its *operational semantics*. This step is necessary because driving is a "generalized" form of program execution using partially known input data, which degenerates into ordinary program execution in the case of the completely known input data, and whose correctness is defined with respect to the underlying operational semantics.
- Develop (or select) a *language of configurations* (for representing sets of execution states). Implement operations over configurations (such as testing two configurations for equivalence or subclass relation).
- Develop a *driving* algorithm (based on the previously selected kind of operational semantics).
- Develop (or choose) an algorithm of recognizing "dangerous" (potentially infinite) branches in the trees of configurations produced by driving. In the field of supercompilation such algorithms are traditionally referred to as *whistles*.
- Develop (or choose) an algorithm of *generalization* that replaces a configuration with a more general one (which represents a larger set of states).
- Develop an algorithm for generating an output (residual) program from a finite graph of configurations.

Recently, in addition to "traditional" supercompilation, there have emerged new kinds of supercompilation, such as *distillation* [4,5], *two-level supercompilation* [20,16] and *multi-result supercompilation* [21,10]. Thus, while initially the topic of research was believed to be *the supercompiler*, it became apparent later that the true interest is in investigating the ways of constructing *supercompilers*.

Hence, there is an obvious increase in diversity among the various forms of supercompilation (both in terms of object languages and different supercompilation algorithms). It can be seen as a manifestation of the general law of "branching growth of the penultimate level" [33].

Also, despite the fact that from the very beginning supercompilation was regarded as a tool for both program optimization and program analysis [35], the research in supercompilation, for a long time, was primarily focused only on program optimization. Recently, however, we have seen a revival of interest in the application of supercompilation to inferring and proving properties of programs [22,19,11].

So there are some reasons to believe that we are witnessing the emergence of such research directions as *language-specific* supercompilation (LSSC) and *domain-specific* supercompilation (DSSC), technical details of the implementation of the general idea of supercompilation being dependent on both the object language and the intended usage of a supercompiler.

As a consequence, the paradigm of research in supercompilation changes. Until recently, the goal of the research was to find "the best" combination of various components in order to produce "the best" supercompiler. However, it has become apparent that "the best" supercompiler just does not exist, because what is good for a particular programming language and/or a domain may not be appropriate for other languages and/or domains.

Thus there arises a new field of research: the systematic study of *various forms and techniques* of supercompilation, as well as of their applicability (in various combinations) in *different areas*.

For example, there is a paper comparing 64 (!) variations of a supercompiler in order to investigate how changes in different parts of the supercompiler affect its ability to prove the equivalence of higher-order expressions [14].

1.2 The goal of MRSC

Obviously, to carry out such experiments one needs some tools for producing a large number of supercompilers or, at least, a large number of variations of a supercompiler. However, it often takes several years for a supercompiler to be constructed by traditional techniques as they are based on "manual labor". This is hardly adequate for research purposes!

> Thus the goal of the MRSC project is to provide a set of "prefabricated components" or, in other words, a toolkit that could facilitate *rapid design and prototyping* of supercompilers.

1.3 The approach of MRSC

The term "multi-result supercompilation" (section 3) implies that, given an input program, a supercompiler may produce a (non-empty) set of residual programs, rather than just a single residual program.

During the development of MRSC it was found that, although it was possible to provide separate implementations for both multi-result and single-result supercompilation, a simpler solution is to regard single-result supercompilation as a special case of the multi-result one (by throwing away all residual programs, except for the first one). This approach is acceptable in terms of efficiency, provided that the set of residual program is generated incrementally, in a lazy way.

In the context of MRSC, the arguments for considering multi-result supercompilation to be "the main case" are the following.

- Traditional supercompilation can be regarded as a special case of multi-result supercompilation. This allows us to treat various kinds of supercompilation in a uniform way. In particular, by describing them by sets of rewriting rules (see sections 2 and 3).
- As will be shown later, multi-result supercompilation enables the components of a supercompiler to be, to a large extent, decoupled from each other. In the first place, this is true of the whistle and the generalization algorithm. So, in the case of multi-result supercompilation, it is easy to perform comparative study of the relative "power" of whistles by considering all possible generalization. This is not possible in the case of traditional supercompilation, because modifications in the whistle bring about modifications in the generalization algorithm, so that the effects produced by changes in different parts of the supercompiler cannot be separated from each other. Thus multi-result supercompilation provides new opportunities for comparative studies in the field of supercompilation.

– Finally, during the development, it became clear that building MRSC on the basis of multi-result supercompilation leads to a more modular, flexible and declarative design of the whole toolkit.

That is why the paper focuses on multi-result supercompilation. Accordingly, MRSC stands for Multi-Result SuperCompilation toolkit.

1.4 The structure of MRSC

Most existing supercompilers have common parts, which do not depend on the object language of a supercompiler, or on the domain of a supercompiler. For example, the overall structure of the graph of configurations and the implementation of operations to work with this graph do not depend on the representation of configurations. Or, for example, different types of whistles and algorithms of generalization can be formulated in abstract form, without the use of information about the details of the language of configurations.

One of the goals of MRSC is to provide some generic data structures and operations that can be used as ready-to-use building blocks for rapid development and prototyping of supercompilers: that is, to provide some basic set of components. On the other hand, a client should have some possibilities of creating additional components and modifying the logic of prefabricated components.

To meet these requirements, we chose Scala [26] as an implementation language of MRSC (although it would be interesting to try to implement a similar toolkit using other programming languages).

Technically, the building blocks provided by MRSC are structured as traits. So a class implementing the main logic of a supercompiler – the construction of graphs of configurations – is assembled from a number of traits.

This paper describes MRSC 1.0. The source code is available at `https://github.com/ilya-klyuchnikov/mrsc`.

1.5 What is in this paper

Due to size limitations we are not able to include all the stuff we would like to present: this paper is only a start of a *series* of publications on MRSC and multi-result supercompilation.

It should be noted that the work on a toolkit implementing the principles of multi-result supercompilation resulted in a revision of some traditional design decisions related to the most low-level part of a supercompiler – the representation of graphs of configurations and the implementation of some operations over them. It has affected the low-level components of MRSC.

This paper considers in detail only the core components of MRSC and the "theoretical foundations" of MRSC. In particular:

– Formal definitions of several kinds of supercompilation in terms of rewriting rules for graphs of configurations. Namely, traditional (single-result, deterministic) supercompilation, non-deterministic supercompilation (as a transformation relation) and multi-result supercompilation.

- Sufficient conditions ensuring the finiteness of any set of completed graphs.
- Internal representation of graphs of configurations based on spaghetti-stacks.
- A method of generating (possibly huge, yet finite) sets of completed graphs of configurations.

Other components of MRSC will be discussed in detail in upcoming papers.

1.6 The structure of the paper

The paper is structured as follows:

- Section 2 introduces a formalism for presenting supercompilation in terms of rewriting rules for graphs of configurations. There are then given two sets of rewriting rules that provide generic specifications for a traditional supercompilation algorithm (corresponding to deterministic supercompilation) and for a transformation relation (corresponding to non-deterministic supercompilation). By comparing these sets of rules one may get some insights about the key differences between the two kinds of supercompilation. In the case of traditional supercompilation, the rewriting rules ensure the generation of a single completed graph of configurations, while the rewriting rules specifying a transformation relation allow the generation of a (possibly infinite) set of completed graphs of configurations.
- Section 3 gives a set of rewriting rules that provide a generic specification for multi-result supercompilation. These rules ensure the generation of a *finite* set of completed graphs of configurations. By inspecting the sets of rules, one can see that multi-result supercompilation can be regarded as a crossbreed between deterministic (traditional) supercompilation and non-deterministic supercompilation (specified by a transformation relation).
- Section 4 describes the core of MRSC. The base level of MRSC implements a few low-level operations over graphs of configurations. MRSC provides two data-structures meant for representing graphs of configurations: `TGraph` (based on trees) and `SGraph` (based on spaghetti-stacks). An explanation is given as to why `SGraph` is more appropriate in the case of multi-result supercompilation. MRSC provides a very simple set of 5 basic "rewriting steps" for transforming graphs of configurations and an abstraction `GraphRewriteRules` for encoding the logic of supercompilation in terms of rewriting rules. The component `GraphGenerator`, when given a set of rewrite rules, incrementally produces *all* possible graphs of configurations.
- Section 5 gives an overview of related works and concludes the paper.

We assume that the reader is familiar with the basics of supercompilation – driving, whistle, generalization and residuation (the paper [32] provides a good introduction into supercompilation).

2 Schemes of traditional supercompilation

In the supercompilation community, there are two well-established approaches to describing and implementing supercompilers.

The first approach formulates supercompilation in terms of the construction of a graph of configurations that is then transformed (residuated) into an output (residual) program [34,28,32,18,13,5]. The origin of this approach goes back to V. Turchin [36].

The second approach [24,23,2,7] considers a supercompiler as an expression transformer that produces output programs "directly", avoiding the construction of intermediate data structures (graphs of configurations)[1]. This "direct-style" approach works especially well if a supercompiler is written in a lazy language (like Haskell) and is required to meet strong performance requirements. A drawback of this approach, however, is that the components of the supercompiler tend to become more strongly coupled: an effect that is hardly desirable in the case of MRSC.

For this reason, our presentation of supercompilation, as well as the design of MRSC, follow the first tradition (based on the explicit construction of graphs of configurations[2]).

The following sections give (generic) specifications of 3 kinds of supercompilation. Namely: traditional (deterministic, single-result) supercompilation, supercompilation transformation relation (or, in other words, non-deterministic supercompilation) and multi-result supercompilation.

These generic specifications describe the construction of graphs of configurations in a language-agnostic way, being parameterized with respect to a set of abstract basic operations: driving, folding, rebuilding and whistle (close to that used by Sørensen [30,31,29]).

2.1 Rewrite rules for graphs of configurations

In the following, it will be assumed that the main result produced by a supercompiler is a *completed* graph of configurations, which is constructed with respect to a program p and an initial configuration c. The process starts by constructing a graph whose single node contains the initial configuration c.

Then the construction of the graph proceeds, step-by-step, by applying graph rewrite rules written in the following form:

$$\frac{precondition}{g \to g'}$$

Let g denote a current graph and g' a graph produced by a single step of rewriting. The rewriting step $g \to g'$ is written under the horizontal bar. Above the horizontal bar there is a *precondition* that should be satisfied in order for this step to be applicable. We assume that there is a predicate $complete(g)$ for checking whether a graph g is completed.

[1] At least in an explicit way.

[2] Creating a toolkit similar to MRSC on the basis of the "direct-style" approach is an interesting, yet open problem for further research.

Transforming operations	
$fold(g,\ \beta,\ \alpha) : Graph$	Folding: looping back from the current node β to a node α in a graph of configurations.
$addChildren(g,\ \beta,\ cs) : Graph$	Adding new nodes: a node is created for each configuration from the list cs, created nodes become children of the current node β.
$rebuild(g,\ \beta,\ c') : Graph$	Rebuilding of a graph: a configuration in an active node β is replaced with a configuration c'.
$rollback(g,\ \alpha,\ c') : Graph$	Another type of rebuilding of a graph: a configuration in a node α (which is not a current node) is replaced with a configuration c', the whole subgraph for which α is a root node is deleted.
Inspecting operations	
$foldable(g,\ \beta,\ \alpha) : Bool$	Predicate, recognizing the possibility for folding of a node β to a node α.
$dangerous(g,\ \beta) : Bool$	(Whistle) Predicate, recognizing a potentially dangerous situation (potentially infinite branch in a graph of configurations).
$complete(g) : Bool$	Predicate, determining whether a graph of configurations g is completed or not.
$rebuilding(g,\ c) : C$	Rebuilding of a configuration c (that is in the current node β) with respect to the whole graph g.
$driveStep(c) : List[C]$	Driving step. Next configurations for a given configuration c.
$rebuildings(c) : List[C]$	The set of rebuildings of a configuration c.

Fig. 1: Operations on graphs of configurations

Some rules in a set may be overlapping. It means that, given a graph, there may be zero, one or more rules that are applicable. For this reason, the initial graph of configurations may, in principle, be rewritten into any number of completed graphs: from zero to infinity.

It turns out that traditional, non-deterministic and multi-result supercompilation can be specified by means of a set consisting of 3 (generic) rules: *Fold*, *Drive* and *Rebuild*. Note that the rules corresponding to different kinds of supercompilation are similar, but differ in some important details, which facilitates the comparison of the 3 kinds of supercompilation.

2.2 Basic operations

Figure 1 presents a set of basic operations that allow supercompilers to be specified in a generic way. The concrete definitions of the operations may vary for different supercompilers. (As an example, see the description of the internals of the supercompiler HOSC [13].) These operations can be naturally divided into two groups: operations that transform a graph of configurations ($fold$, *addChildren*,

rebuild) and operations that only inspect a graph of configurations (*foldable*, *dangerous*, *rebuilding*, *driveStep*, *rebuildings*).

In fact, generic formulations of supercompilation do not depend on the exact meaning of "inspecting" operations: it is enough to know the types of their results and how the results are used. Also note that the names of some operations we use in the paper differ from those used by Sørensen: *addChildren* corresponds to *drive* and *rebuild* corresponds to *abstract* [30,31,29].

The operations *rebuild*, *rebuilding* and *rebuildings* deserve a special comment. Unfortunately, in supercompilation the term *generalization* is "overloaded", which can be illustrated by the following two quotations.

From [31]:

> Note that we now use the term generalization in two distinct senses: to denote certain operations on trees performed by supercompilation, and to denote the above operation on expressions. The two senses are related: generalization in the former sense will make use of generalization in the latter sense.

From [37]:

> A reduction from node N_1 to N_2 is an assignment of such values to $var(N_2)$ in terms of $var(N_1)$ that after their substitution the configuration in N_2 becomes identical to that in N_1. The node N_2 may be either (1) a generalization of N_1 [...]. Transition by a reduction edge includes no computational steps of the machine: *the exact state of the computing machine remains the same; only its representation gets changed.*

On the one hand, a configuration c' is said to be a generalization of a configuration c if $c \sqsubset c'$ (which means that the set represented by c' contains the set represented by c). On the other hand, let us consider three configurations in the language SLL [18]:

$f(Nil, g(y))$ (c_1)
$f(x, g(y))$ (c_2)
$let\ x = Nil\ in\ f(x, g(y))$ (c_3)

Here $c_1 \sqsubset c_2$, i.e. c_2 is a generalization of c_1. Note that c_2 does not contain enough information for the initial configuration c_1 to be restored. Now suppose that c_1 and c_2 appear in a graph of configurations, and c_1 is the current node. Then we cannot perform generalization just by replacing c_1 with c_2! Actually, during supercompilation, c_1 is replaced with c_3 (which contains c_2 as a subexpression). For this reason it is c_3, rather than c_2 that is sometimes referred to as a generalization of c_1.

This ambiguity in terminology is no good, as it may be a source of confusion. For this reason, we will use a more technical term *rebuilding* (quite popular in supercompilation folklore), giving it a precise meaning.

A rebuilding of a configuration is an alternative representation of the configuration (in accordance with the above quotation from Turchin). The original

configuration can be uniquely restored from a rebuilding. For example, c_3 is a rebuilding of c_1. For a given language of configurations, the set of all possible rebuildings of a given configuration is usually finite.

A lower rebuilding of a graph of configurations is the replacement of a configuration c in the current node with a configuration c'.

The upper rebuilding of a graph of configurations (a rollback to α) is the deletion of all successors of the node α, followed by the replacement of a configuration c in α with a configuration c'.

It will be assumed that $rebuilding(g,\ c) \in rebuildings(c)$.

2.3 Scheme of supercompilation algorithm

The generic scheme of traditional supercompilation is specified by the SC-rules shown in Figure 2a. The determinacy follows from the fact that, given a graph that is not completed, there is exactly one rule that can be applied (in an unambiguous way).

These rules can be interpreted as a step-by-step algorithm:

- While a graph of configurations is not completed:
  - If there is a node for looping back, then make the corresponding folding ($Fold$),
  - else if the current state of the graph is considered to be dangerous ("the whistle blows"), then deterministically find a rebuilding of the current configuration with respect to the current graph and then perform the lower rebuilding of the graph ($Rebuild$),
  - otherwise, make a step of driving ($Drive$).

2.4 Scheme of transformation relation

A supercompilation transformation relation does not use whistle and allows any possible rebuilding to be performed, provided that the $Fold$ rule is not applicable.

The generic scheme of non-deterministic supercompilation is specified as a transformation relation by the NDSC-rules shown in Figure 2b. Technically, there are two differences from the case of traditional (deterministic) supercompilation:

1. If there is no possibility for folding, then both a driving step and a rebuilding are allowed.
2. A rebuilding of the current configuration can be done non-deterministically, by using any configuration from $rebuildings(c)$.

Since we assume that $rebuilding(g, c) \in rebuildings(c)$, it can be easily seen that, given a set of operations over graphs of configurations, the transformation supercompilation relation is an extension with respect to traditional supercompilation. In other words, if the deterministic supercompiler produces a residual program for a given input program, then the non-deterministic supercompiler is also able to produce this residual program.

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad \neg dangerous(g, \beta) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad dangerous(g, \beta) \quad c' = rebuilding(g, c)}{g \to rebuild(g, \beta, c')}$$

(a) SC: Deterministic (traditional) supercompilation

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad c' \in rebuildings(c)}{g \to rebuild(g, \beta, c')}$$

(b) NDSC: Non-deterministic supercompilation (transformation relation)

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad \neg dangerous(g, \beta) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad c' \in rebuildings(c)}{g \to rebuild(g, \beta, c')}$$

(c) MRSC: Multi-result supercompilation

Notation:

- g – a current graph of configurations
- β – a current node in a graph of configurations
- c – a configuration in a current node β

Fig. 2: Schemes of different types of supercompilation

In general, for a given input program, a transformation relation defines a (possibly) infinite set of completed graphs of configurations and a (possibly) infinite set of incomplete graphs of configurations.

Transformation relations are useful for proving the correctness of supercompilation algorithm and for formulating some abstract properties of supercompilation [12,15,27].

3 Multi-result supercompilation

Essentially, multi-result supercompilation can be regarded as a crossbreed between deterministic (traditional) supercompilation and non-deterministic supercompilation (specified by a transformation relation)

3.1 Scheme of multi-result supercompilation

The scheme of multi-result supercompilation is specified by the MRSC-rules shown in Fig. 2c.

It can be seen that the MRSC-rules can be regarded as a combination of the SC-rules and the NDSC-rules. The rule *Fold* is the same for all sets of rules. The rule *Drive* is taken from the SC-rules and the rule *Rebuild* from the NDSC-rules.

Note that in the case of the SC-rules, the whistle and rebuilding are strongly coupled: if the whistle blows, there has to be done a rebuilding, but if the whistle does not blow, rebuilding is prohibited and a driving step has to be done.

However, this is not true of the MRSC-rules, because a rebuilding may be performed even if the whistle does not blow. But the subtle point is that there may arise a situation when the rule *Fold* is not applicable, the whistle blows, thereby making the *Drive* inapplicable, and the set $rebuildings(c)$ is empty, for which reason no rebuilding is possible. It means that the process of supercompilation has come to an impasse, and the graph of configurations is "unworkable" and has to be discarded.

Let us recall that applying the SC-rules results in producing a single completed graph, the NDSC-rules a (possibly) infinite set of completed graphs, and the MRSC-rules a finite set of completed graphs.

Theorem 1 (Finiteness of sets of completed graphs). *If*

1. *any infinite branch in a graph of configurations is detected by the predicate dangerous,*
2. *for any configuration c the set* $rebuildings(c)$ *is finite,*
3. *the number of successive rebuildings cannot be infinite (i.e. the chain* $c_1, c_2, c_3, \ldots$, *where* $c_{k+1} \in rebuildings(c_k)$ *is always finite),*

then the application of the MRSC-rules produces a finite set of completed graphs of configurations.

Proof. Collapse all successive rebuildings into one rebuilding. Everything else follows from König lemma [8] (using arguments similar to those in the Sørensen's proof [29]).

In the same way one can show that the MRSC-rules always produce a finite set of dead-end graphs (to which no rule is applicable).

The third condition in the assertion of the theorem may seem to be superfluous. However, this is not true. Let us consider a supercompiler, such that (1)

numbers are allowed as variable values in its input language, and (2) configurations may impose restrictions on variable values having the form $x < N$, where x is a variable and N is a natural number.

Suppose that the finiteness of $rebuildings(c)$ is ensured by the following requirement: if all number constants in c do not exceed N, then all number constants appearing in any $c' \in rebuildings(c)$ do not exceed $N + 1$. Then the number of rebuildings for any configuration will be finite, but the number of successive rebuildings can be infinite. For example:

$$f(x)|_{\{x<5\}} \to let\ y = x|_{\{x<5\}}\ in\ f(y)|_{\{y<6\}} \to let\ z = y|_{\{y<6\}}\ in\ f(z)|_{\{z<7\}} \to \dots$$

If $f(x)|_{\{x<5\}}$ is the initial configuration, then an infinite number of completed graphs of configurations can be generated.[3]

3.2 Tree of graphs

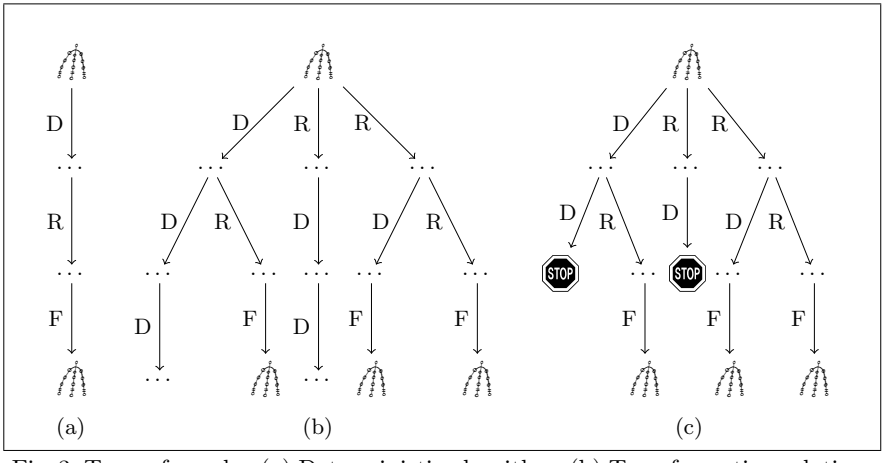


Fig. 3: Trees of graphs. (a) Deterministic algorithm, (b) Transformation relation, (c) Multi-result supercompilation.

Suppose, we are given an initial configuration. Then the rules shown in Fig. 2 specify the process of supercompilation as a sequence of rewriting steps. A sequence of rewritings will be called "successful" if it leads to a completed graph of configuration, and "unsuccessful" if it leads to a dead end (i.e. to a graph such that no rule is applicable).

Note that (1) the SC-rules define a single successful finite sequence of rewritings, (2) the NDSC-rules define an infinite tree of rewriting steps containing finite

[3] It may happen that this infinite number of graphs is residuated into a finite set of really different output programs.

successful branches, finite unsuccessful branches and infinite branches, and (3) MRSC-rules define a finite tree of rewriting steps with finite successful branches and finite unsuccessful branches (see Fig. 3).

Thus, Theorem 1 can be reformulated as follows: multi-result supercompilation defines a finite tree of graph rewriting.

3.3 Decoupling whistle and generalization

Let us take a closer look at the differences between deterministic (traditional) supercompilation and multi-result supercompilation.

Comparing the SC-rules and the MRSC-rules in Fig. 2 reveals that these two kinds of supercompilation only differ in the rule *Rebuild*. In the case of the SC-rules, driving and generalization (rebuilding) are mutually exclusive, and the decision whether to drive or generalize is taken by the whistle, while in the case of the MRSC-rules a configuration can be rebuilt even if the whistle is silent. The consequence is that the MRSC-rules completely decouple the whistle from the generalization algorithm: the whistle does not have to bother about whether a configuration declared to be "dangerous" can be rebuilt, or not?

Hence, as regards the whistle and generalization, multi-result supercompilation provides a better separation of concerns, than traditional supercompilation, and this is especially important when doing research work in the field of supercompilation. Since a whistle does not have to take into account generalization/rebuilding, it becomes easier to give a try to a variety of unusual whistles. On the other hand, an algorithm of generalization is no longer required to guess "the best" generalization: it is sufficient for it to produce $rebuildings(c)$, a finite set of rebuildings.

Certainly, even if $rebuildings(c)$ is finite for any configuration c, it may be still too large, so that a huge number of residual programs may be produced. However, this is acceptable if we need to understand, first of all, whether a whistle is *in principle* able to produce good results, or not. After that we may proceed to the next task: how to reduce the size of $rebuildings(c)$ by only selecting "reasonable" rebuildings.

3.4 Multi-result supercompilation as branching growth of the penultimate level

The idea of multi-result supercompilation is quite simple. The fact that, until recently, it has not been explicitly formulated can be due to two reasons.

First, for a long time, supercompilation has been primarily considered as a program optimization technique, for which reason it was believed to be "natural" for a supercompiler to produce a single result (the "best possible" one). However, in the case of program analysis, it is not clear, what is the "best" residual program? Thus we come to the idea of a supercompiler producing a set of residual programs.

Second, multi-result supercompilation reveals its true potential only in combination with higher-level supercompilation (in particular, two-level supercompilation). While, in the case of traditional supercompilation, the transition from single-result supercompilation to multi-result supercompilation gives only quantitative change. Combining two-level supercompilation with multi-result supercompilation produces fundamentally new results [21].

4 The core of MRSC

Now let us consider which technical issues arise when developing a multi-result supercompiler and how these issues are addressed in MRSC.

The most sophisticated technical task of a supercompiler is the construction of a graph of configurations. A supercompiler constructs this graph in a top-down manner, starting from an initial configuration. In the case of traditional, single-result supercompilation, when a single graph of configurations is to be constructed, the internal representation of this graph is not of importance. One may choose to use a mutable data structure for graph representation and modify it step-by-step as it was done in [37] (imperative style). Another option is to use an immutable data structure: if the implementation language of a supercompiler is a call-by-value one, a new structure will be generated at each step [18]. Or we can use a lazy implementation language, in which case graphs of configurations can be constructed in a lazy manner [17].

In any case, as can be seen from the literature, most supercompilers based on the explicit construction of graphs of configurations, have represented graphs by top-down trees. This representation is convenient for the generation of residual programs, since, traditionally, a residual program is constructed by traversing a graph in a top-down manner[4].

But, as will be shown in the next subsection, the tree-based representation of graphs of configurations is inconvenient for multi-result supercompilation. Therefore, MRSC uses another representation for graphs, based on *spaghetti-stacks* [1].

4.1 Two data structures for a graph of configurations

MRSC uses two representations for graphs of configurations: T-representation (tree-based) and S-representation (based on spaghetti-stacks [1]). The Scala encoding of these representation is shown in Fig. 4.

T-representation is used when transforming a graph into a residual program. S-representation is used during the step-by-step construction of a graph of configurations. When a graph is completed, it can be either used as it is (in S-representation), or it may be transformed from S-representation into T-representation (to be then residuated).

[4] It is interesting to find an elegant way to construct a residual program using bottom-up traversal.

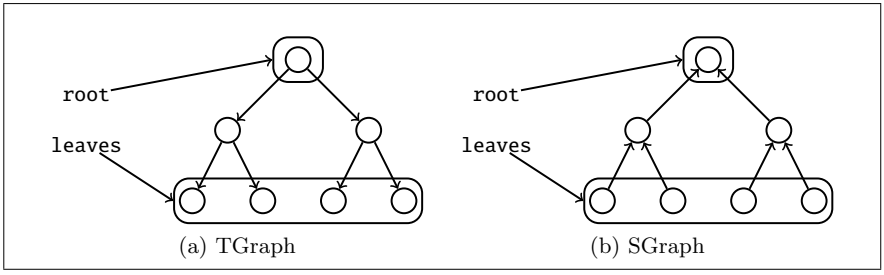
```
type TPath = List[Int]
type SPath = List[Int]

case class TNode[C, D](
  conf: C, outs: List[TEdge[C, D]],
  base: Option[TPath], tPath: TPath)

case class TEdge[C, D](
  node: TNode[C, D], driveInfo: D)

case class TGraph[C, D](
  root: TNode[C, D], leaves: List[TNode[C, D]])

case class SNode[C, D](
  conf: C, in: SEdge[C, D],
  base: Option[SPath], sPath: SPath)

case class SEdge[C, D](
  node: SNode[C, D], driveInfo: D)

case class SGraph[C, D](
  incompleteLeaves: List[SNode[C, D]],
  completeLeaves: List[SNode[C, D]],
  completeNodes: List[SNode[C, D]]) {

  val isComplete = incompleteLeaves.isEmpty
  val current = if (isComplete) null else incompleteLeaves.head
}
```

Fig. 4: Graphs

Graphs in T-representations are objects of the class `TGraph[C, D]` holding information of the following kinds:

1. `C` (configuration) – configurations labeling nodes of a graph.
2. `D` (driving info) – information labeling graph edges. This information describes the "evolution" of configurations (a transient step of driving, a branching, a decomposition, etc). This information is useful for producing residual programs.

Every node in a T-graph is represented by an object of class `TNode[C, D]` which holds information about its configuration and its output edges. We also store a path from the root node to this node: it facilitates some manipulations with the graph and can be used as a unique identifier of the node inside its graph. The information about folding is stored as an (optional) path to the base node. So, in a sense, `TGraph` is a tree with additional information about cycles (foldings) in some leaves of this tree.



(a) TGraph (b) SGraph

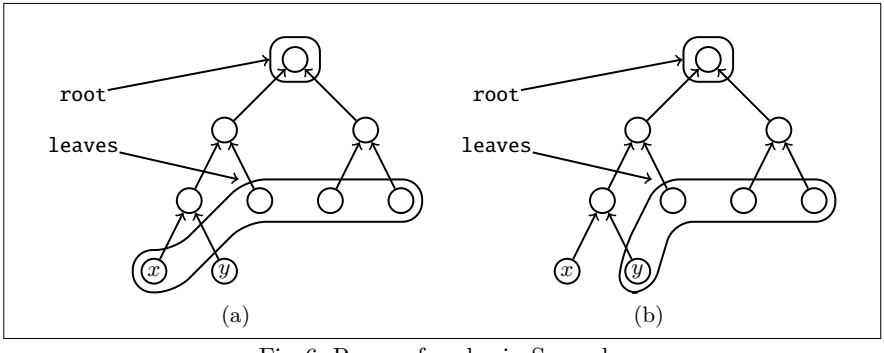
Fig. 5: MRSC data structures

The edges of a graph are coded as `TEdge[C, D]`, which are unidirectional, an edge only storing the information about its destination.

The "entry point" of `TGraph[C, D]` is its root node. Also there is additional information about leaves, which may be useful for residuation.

As was mentioned above, T-representation is convenient for top-down traversal of graphs. However, if we need to make additions to a T-graph in two different ways, we have to do some copying. But, in the case of multi-result supercompilation, we have to do divergent additions to the current graph nearly at every step. So, T-graphs seem to be impractical for multi-result supercompilation. It is easier to turn T-graphs upside down, to obtain S-graphs represented by `SGraph[C, D]`.

Thus `TGraph[C, D]` is totally dual to `SGraph[C, D]`. The two data structures are schematically shown in Fig. 5.



(a) (b)

Fig. 6: Reuse of nodes in S-graphs

Both data structures are immutable. Let us go into details of how S-graphs allow different additions to graphs to be made in a functional way.

Suppose there are two rewriting steps applicable for the graph shown in Fig. 5: adding a child node with a configuration x to the leftmost leaf or adding

a child node with a configuration y to the leftmost leaf. In the case of S-graphs, it is sufficient to create two nodes and to reuse some parts of the previous graph to make two new graphs! This sharing of nodes is shown schematically in Fig. 6.

It should be noted that S-representation is more convenient for the implementation of whistles, than T-representation: the majority of whistles traverses a branch of a graph in the bottom-up way starting from the current node.

Despite these differences, many supercompilers (for historical reasons?) use T-representation when building graphs of configurations.

So a graph of configurations being constructed is represented by the class `SGraph[C, D]`: the field `current` represent the current node, `incompleteLeaves` represent leaves that are not yet processed, `completeLeaves` represent completed leaves. Also there is an additional list `completeNodes` representing a completed part of a graph.

4.2 Basis of operations on S-graphs

One of the main goals of MRSC is to allow a programmer to concentrate on writing the *logic* of a (multi-result) supercompiler saving him the trouble of coding routine operations. In a sense, the lowest level of a supercompiler's logic is the definition of rewriting rules for graphs of configurations. MRSC allows these rules to be encoded in a semi-declarative way.

MRSC provides a basis consisting of five "build steps", denoting rewriting operations over graphs of configurations in S-representation. This basis is shown schematically in Fig. 7.

Each step is represented as a Scala value of type `GraphStep[C, D]` and is assumed to be executed over the *current* graph of configurations (Fig. 8):

1. `CompleteCurrentNodeStep` – marks the current leaf as a completed one. Used in driving.
2. `FoldStep` – performs a folding.
3. `AddChildNodesStep` – adds child nodes to the current node. Used in driving.
4. `RebuildStep` – performs a lower rebuilding of the graph (by replacing the configuration in the current node).
5. `RollbackStep` – performs an upper rebuilding of the graph (deleting the corresponding sub-graph).

The process of constructing any graph of configurations that is producible by supercompilation can be represented by a sequence of the above build steps. The build steps are executed by an interpreter that is provided by MRSC as part of the graph generator (see below). The supercompilers implemented by means of MRSC never transform graphs of configurations directly: they instead generate build steps that are interpreted by the graph generator. This, to some extents, ensures the correctness of transformations over graphs of configurations.

Note that the use of S-graphs allows rollback operation to be performed in an elegant functional way (see the MRSC source code)[5].

[5] In [3] rollbacks are implemented by means of the mechanism of exceptions.

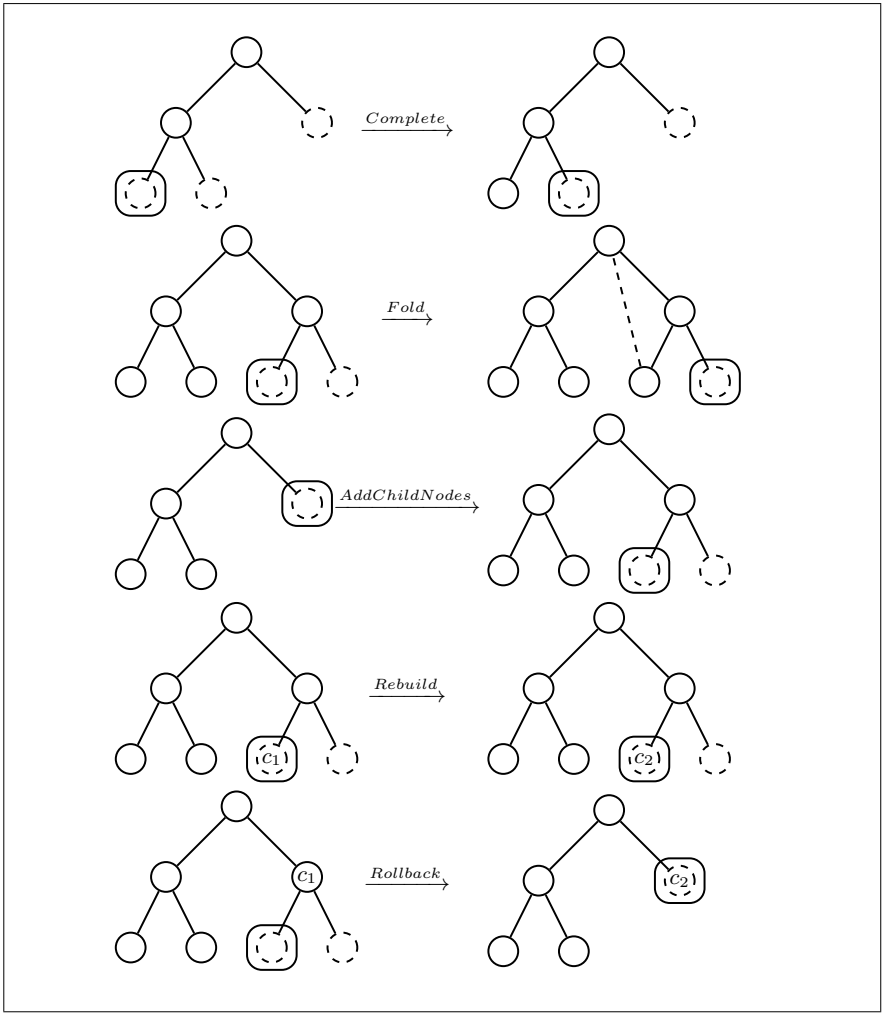


Fig. 7: Basis of operations on graphs schematically

Another useful feature of encoding build steps as first-order values is that they can be easily serialized and stored for future use. Then they can be submitted to another software tool, such as a validator of sequences of build steps. Given a start graph (with a single node) and a sequence of graph rewriting steps, the validator will be asked to check whether this sequence of steps may be generated by a certain supercompiler (or even by a transformation relation), or not.

```
sealed trait GraphRewriteStep[C, D]

case class CompleteCurrentNodeStep[C, D]
  extends GraphRewriteStep[C, D]

case class AddChildNodesStep[C, D](ns: List[(C, D)])
  extends GraphRewriteStep[C, D]

case class FoldStep[C, D](to: SPath)
  extends GraphRewriteStep[C, D]

case class RebuildStep[C, D](c: C)
  extends GraphRewriteStep[C, D]

case class RollbackStep[C, D](to: SPath, c: C)
  extends GraphRewriteStep[C, D]
```

Fig. 8: Rewrite steps for S-graphs

```
trait GraphRewriteRules[C, D] {
  type N = SNode[C, D]
  type G = SGraph[C, D]
  type S = GraphRewriteStep[C, D]
  def steps(g: G): List[S]
}

case class GraphGenerator[C, D]
  (rules: GraphRewriteRules[C, D], conf: C)
  extends Iterator[SGraph[C, D]] { ... }
```

Fig. 9: MRSC "middleware" for supercompiler construction

4.3 Generating graphs of configurations

Technically, a supercompiler written using MRSC is based upon two components shown in Fig. 9: `GraphRewriteRules` and `GraphGenerator`.

The trait `GraphRewriteRules` describes the logic of a multi-result supercompiler in a form similar to that in Fig. 2c. This trait only declares the method `steps`. A concrete supercompiler is required to provide an implementation for this method. So the trait `GraphRewriteRules` only provides an interface for using the rules.

The class `GraphGenerator`, by contrast, is a ready-to-use component: it is a constituent part of any supercompiler built on top of MRSC.

`GraphGenerator` for a given initial configuration `conf` and rewriting rules `rules`, generates *all* completed graphs of configurations defined by these rules.

If `rules` represent the logic of a traditional single-result supercompiler, then (of course) the generator will produce a single graph.

In general, the number of graphs may be huge. Thus, to keep memory consumption within reasonable limits, the graph generator is implemented as an iterator and produces graphs on demand.

The internals of the graph generator are extremely simple (see the source code). It maintains a set of incomplete S-graphs and a queue of completed graphs. If a client requests the next graph and the queue is not empty, then the first graph from this queue is returned. Otherwise, a graph `g` from the set of incomplete graphs is chosen, and `steps(g)` is called, to produce a set of graph build steps (which may be empty). Then each of the steps is applied to `g`, to obtain a set of new graphs. Some of the new graphs are completed and some are incomplete. The completed graphs are added to the queue of completed graphs, while the incomplete ones are added to the current set of incomplete graphs.

(There may be implemented other strategies, producing the completed graphs in other orders. The current implementation is straightforward, and makes the depth-first traversal of the "tree of graphs".)

What should a client do with the graphs generated by `GraphGenerator`? In the case of a traditional supercompiler, a client may transform them into T-graphs and then residuate these T-graphs into output programs. However, other variants are possible. For example, a client may filter out completed graphs in order to find graphs with specific properties. In some cases the fact of existence or absence of graphs with specific properties may be of a special interest (when supercompilation is used for program analysis).

Note, that the interface to the definition of graph rewriting rules shown in Fig. 9 is quite abstract and does not depend on the input languages of supercompilers. This enables the graph generator to be completely language-agnostic.

5 Conclusion

This paper describes only the internal structure and technical design of the MRSC core. Further papers will present concrete examples of rapid prototyping of supercompilers by means of MRSC and the use of MRSC for implementing domain-specific supercompilers.

The first work addressing the problem of developing a general "abstract" framework for specifying and implementing supercompilers was [37], which introduced a domain-specific language SCPL for describing graph transformations. Unfortunately, later there has been no active development in this field.

To some extent, the core of MRSC follows the spirit of SCPL, but there are some significant differences.

First, MRSC is focused on multi-result supercompilation, which is a superset of traditional supercompilation. The main idea of multi-result supercompilation is the multiplicity of possible results. This idea is extended naturally into the thesis about the variety and multiplicity of (multi-result) supercompilers that can be used for a variety of purposes.

The second difference is that MRSC is designed and implemented in functional style: the core data-structures (S-graph) of MRSC are immutable, which makes it possible to generate thousands of graphs, while still keeping memory consumption within reasonable limits. In addition, it allows, in principle, to develop a parallelized version of MRSC, so that the divergent versions of a graph of configuration can be processed simultaneously.

Of course, the first version of the MRSC toolkit is far from ideal. But we hope that further improvements in MRSC will be driven by experience gained by using it for implementing language- and domain-specific multi-result supercompilers.

Acknoledgements

The authors express their gratitude to all participants of Refal seminar at Keldysh Institute for useful comments and fruitful discussions of this work and to Natasha and Lena for their love and patience.

References

1. D. G. Bobrow and B. Wegbreit. A model and stack implementation of multiple environments. *Commun. ACM*, 16:591–603, October 1973.
2. M. Bolingbroke and S. L. Peyton Jones. Supercompilation by evaluation. In *Haskell 2010 Symposium*, 2010.
3. M. Bolingbroke and S. L. Peyton Jones. Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation, 2011. Rejected by ICFP 2011.
4. G. W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70. ACM Press New York, NY, USA, 2007.
5. G. W. Hamilton. A graph-based definition of distillation. In *Second International Workshop on Metacomputation in Russia*, 2010.
6. N. D. Jones. The essence of program transformation by partial evaluation and driving. In *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, PSI '99, pages 62–79, London, UK, UK, 2000. Springer-Verlag.
7. P. Jonsson and J. Nordlander. Taming code explosion in supercompilation. In *PEPM'11*, 2011.
8. S. Kleene. *Mathematical logic*. Dover books on mathematics. Dover Publications, 2002.
9. A. Klimov. An approach to supercompilation for object-oriented languages: the Java supercompiler case study. In *First International Workshop on Metacomputation in Russia*, 2008.
10. A. Klimov. Multi-result supercompilation in action: Solving coverability problem for monotonic counter systems by gradual specialization. In *International Workshop on Program Understanding (PU 2011)*, 2011.
11. A. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In E. Clarke, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics, 8th Andrei Ershov Informatics Conference, PSI 2011,*

Akademgorodok, Novosibirsk, Russia, June 27 – July 01, 2011, volume 7162 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2012.
12. A. V. Klimov. A program specialization relation based on supercompilation and its properties. In *First International Workshop on Metacomputation in Russia*, pages 54–77, 2008.
13. I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.
14. I. Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. Preprint 62, Keldysh Institute of Applied Mathematics, 2010.
15. I. Klyuchnikov. Supercompiler HOSC: proof of correctness. Preprint 31, Keldysh Institute of Applied Mathematics, Moscow, 2010.
16. I. Klyuchnikov. Towards effective two-level supercompilation. Preprint 81, Keldysh Institute of Applied Mathematics, 2010.
17. I. Klyuchnikov. The ideas and methods of supercompilation. *Practice of Functional Programming*, 7, 2011. In Russian.
18. I. Klyuchnikov and S. Romanenko. SPSC: a simple supercompiler in Scala. In *PU'09 (International Workshop on Program Understanding)*, 2009.
19. I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.
20. I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.
21. I. Klyuchnikov and S. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In *PSI 2011*, 2011.
22. A. Lisitsa and A. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.
23. N. Mitchell. Rethinking supercompilation. In *ICFP 2010*, 2010.
24. N. Mitchell and C. Runciman. A supercompiler for core haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes In Computer Science*, pages 147–164, Berlin, Heidelberg, 2008. Springer-Verlag.
25. A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: Experiments with a supercompiler. In *Perspectives of System Informatics*, volume 1181 of *LNCS*, pages 249–260. Springer, 1996.
26. M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2nd edition, 2010.
27. D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1-2):193–233, 1996.
28. M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Dept. of Computer Science, University of Copenhagen, 1994.
29. M. H. Sørensen. Convergence of program transformers in the metric space of trees. In *Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 315–337, 1998.
30. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Logic Programming: The 1995 International Symposium*, pages 465–479, 1995.
31. M. H. Sørensen and R. Glück. Introduction to supercompilation. In *Partial Evaluation. Practice and Theory*, volume 1706 of *LNCS*, pages 246–270, 1998.

32. M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
33. V. F. Turchin. *The phenomenon of science. A cybernetic approach to human evolution.* Columbia University Press, New York, 1977.
34. V. F. Turchin. A supercompiler system based on the language refal. *SIGPLAN Not.*, 14(2):46–54, 1979.
35. V. F. Turchin. *The Language Refal: The Theory of Compilation and Metasystem Analysis.* Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.
36. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.
37. V. F. Turchin. Supercompilation: Techniques and results. In *Perspectives of System Informatics*, volume 1181 of *LNCS*. Springer, 1996.
38. V. F. Turchin, R. M. Nirenberg, and D. V. Turchin. Experiments with a supercompiler. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 47–55, New York, NY, USA, 1982. ACM.

A Metacomputation Toolkit for a Subset of F♯ and Its Application to Software Testing

Towards Metacomputation for the Masses

Dimitur Krustev

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

Abstract. We present an on-going experiment to develop a practical metacomputation toolkit for F#. There are – apart from the better known supercompilation – other mature and potentially useful methods stemming from metacomputation theory: program inversion and tabulation, neighborhood analysis. Although implementations of these methods have existed since many years, they are typically experimental tools, treating specifically designed small languages. We investigate if such methods can be made more readily available for practical use, by re-developing them for a reasonably large subset of a mainstream programming language. Practical technical challenges – together with possible solutions – are discussed. We also hint at a potential practical application – automatic generation of software test sets based on user specifications of "interesting" input partitioning.

1 Introduction

The metacomputation theory, developed originally by Turchin, gives rise to several interesting techniques. The best known is supercompilation [26], but there are others as well – such as neighborhood analysis, neighborhood testing, program tabulation and inversion [4]. While many supercompiler implementations already exist, some of them for large subsets of popular programming languages, the situation is different for other metacomputation techniques like neighborhood analysis or program tabulation/inversion. Apart from the pioneering work done in the context of Refal, there are few (publicly available) experimental implementations [4,1,2,15], treating specially designed small languages (S-Graph, TSG, XSG, SLL). While these implementations work well, and can perform impressive tasks, they are hardly usable by someone without extensive knowledge in metacomputation theory. For one thing, they require learning a new language, specific to the tool, typically first-order functional, often even flat (without nested function calls). While parsers and pretty-printers often exist, other tools like IDEs or even debuggers are not available. The main motivation for the current work is to try to transfer existing metacomputation techniques in the context of a larger and more popular programming language. We chose to work with a subset of F#, as F# has a clean and relatively simple functional core

(stemming from its ((O)Ca)ML heritage), together with some facilities greatly simplifying the creation of meta-programs, while at the same time it runs on a very popular platform, and is well supported by several powerful IDEs (Visual Studio, SharpDevelop, MonoDevelop [1]).

In line with the overall pragmatic orientation of this experiment, we also wanted an interesting and useful practical application to serve as a use case for the metacomputation machinery. While a very sophisticated testing method based on metacomputation already exists – neighborhood testing [4] – we guessed that it would pose more technical challenges to implement fully, and settled at first on a simpler (and less powerful) method for test generation. In particular, we are interested in ways to apply the equivalence partitioning method for black-box testing [5]. Given a partitioning specification, encoded in the subset of F# that our tool supports, we can apply program tabulation (a key first step in the URA-technique for program inversion) to generate test skeletons for each equivalence class of the chosen data partition. Afterward it would be easy to fill the skeleton holes with arbitrary values of the appropriate type.

To take a really simple example, consider a program (or a set of programs) dealing with binary trees. Important quantitative characteristics of trees are their size and height, so we may consider a partitioning based on the pair (size, height). We can easily create (Fig. 1), in F#, a description of the data domain, and the partitioning specification [2]. If we then make a program tabulation request for the expression shown in Fig. 2, we obtain the table shown in Fig. 3. We group input trees [3] by the result they produce, and we have filtered out those with result `None`.

```
type BinTree<'T> =
    | EmptyTree
    | Node of 'T * BinTree<'T> * BinTree<'T>

[<ReflectedDefinition>]
let rec treeSize t =
    match t with
    | EmptyTree -> NZero
    | Node(_, l, r) -> NSucc (natAdd (treeSize l) (treeSize r))

[<ReflectedDefinition>]
let rec treeHeight t =
    match t with
    | EmptyTree -> NZero
    | Node(_, l, r) -> NSucc (natMax (treeHeight l) (treeHeight r))
```

Fig. 1. Binary tree with size, height

[1] All product names mentioned in the article are trademarks of their respective owners.

[2] We assume some familiarity with the syntax of F#, or other languages in the ML family.

[3] Actually expressions representing sets of input trees.

```
<@ fun t ->
    let size = treeSize t
    let height = treeHeight t
    let b1 = natLE size (NSucc(NSucc(NSucc(NZero))))
    let b2 = natLE height (NSucc(NSucc(NZero)))
    if boolAnd b1 b2 then Some (size, height) else None @>
```

Fig. 2. Binary tree tabulation request by (size, height)

```
[(Some (Tuple2 (NSucc (NSucc (NSucc (NZero))),
  NSucc (NSucc (NZero)))),
  [map [(t_0, Node (__3,Node (__6,EmptyTree,EmptyTree),
    Node (__9,EmptyTree,EmptyTree)))]]);
 (Some (Tuple2 (NSucc (NSucc (NZero)), NSucc (NSucc (NZero)))),
  [map [(t_0, Node (__3,
    Node (__6,EmptyTree,EmptyTree),EmptyTree))];
   map [(t_0, Node (__3,EmptyTree,
     Node (__6,EmptyTree,EmptyTree)))]]);
 (Some (Tuple2 (NSucc (NZero),NSucc (NZero))),
  [map [(t_0, Node (__3,EmptyTree,EmptyTree))]]);
 (Some (Tuple2 (NZero,NZero)), [map [(t_0, EmptyTree)]])]
```

Fig. 3. Binary tree tabulation result

A couple of things to note: we have restricted both the size and the height of the trees we consider, in order to reduce the search space and to get a small number of tests. Each entry in the resulting table represents an equivalence class – a set of input values giving a particular output. The set is represented as a list of maps, each map specifying possible values for all parameters of the tabulated function. Further, the input expressions in the table contain free variables (`__3`, `__6`, ...), for places in the input trees, that bear no influence on the expression value. We could, however, instantiate those variables with suitable constants (based on the type) and get concrete tests as a final result. For example, the table entry `(Some (Tuple2 (NZero, NZero)), [map [(t_0, EmptyTree)]])` shows, that only the empty tree has both size and height 0.

We shall see a couple of more realistic examples of partitioning-based testing in Sect. 5. Before that, we discuss how F# features like code quotations facilitate the creation of "embedded" meta-programs operating on other parts of the F# program (Sect. 2). We also motivate the choice of the particular F# subset covered. Next we outline the implementation of the basic metacomputation algorithms – driving and process-tree creation (Sect. 3). We mostly follow an approach similar to existing implementations, but also discuss practical challenges and implementation tricks. The treatment of program tabulation and inversion (Sect. 4) is based on the classical approach used in URA [4,1,2], but the fact that we deal with a non-flat, higher-order functional language poses some unexpected obstacles. In fact, the current implementation is – as far as we know – the first that lifts URA-like methods to a higher-order language.

2 "Decompiling" F# quotations

While typical meta-program implementations require re-implementing some phases of a standard compiler front-end (like lexing, parsing, de-sugaring, type-checking), with F# we can take a shortcut by using its built-it facilities for meta-programming – called "code quotations". In brief, F# quotations permit to instruct the compiler to store a high-level, abstract-syntax-tree (AST) representations of parts of the program, alongside (or instead of) the compiled low-level byte-code [24]. There are basically 2 ways to achieve this:

- by placing `[<ReflectedDefinition>]` in front of a top-level function (or method) definition (producing both a normal compiled definition and an AST representation);
- by enclosing any (syntactically complete) expression in `<@ ... @>` (which returns only the AST representation of the expression).

The use of both methods was already demonstrated in the introductory example. There are special facilities (the `MethodWithReflectedDefinition` active pattern, for example) for retrieving the AST corresponding to a reflected top-level function. The design of F# quotations resembles similar designs in MetaML and Template Haskell. There is no need to explicitly require reflecting type definitions – it is done by default in languages sitting on top the .NET run-time.

While in principle it would be possible to base driving and further metacomputation algorithms directly on the internal AST representation of quotations, we take a two-step approach: we first translate quotation ASTs (if possible) to another representation (Fig. 4) on which driving is then performed. The advantages of this approach are:

- we can precisely specify the subset of F# we treat;
- we can use a language representation more suitable for driving.

```
type BindPattern = PVar of VName | PWildcard
type Pattern = PCon of CName * BindPattern list
type Exp =
    | EVar of VName
    | EApp of Exp * Exp
    | ELam of BindPattern * Exp
    | ELet of VName * Exp * Exp
    | ELetRec of (VName * Exp) list * Exp
    | ECon of CName * Exp list
    | ECase of Exp * (Pattern * Exp) list
```

Fig. 4. F# subset definition

One can see immediately that we retain most typical features of modern functional languages (higher-order functions, let- and letrec-expressions, algebraic data types with pattern matching). Readers well familiar with F# can already deduce what is left out:

- all features related to object-oriented programming, and ensuring interoperability with other .NET languages (classes, structs, methods, etc.);
- all features related to mutable data structures, and side-effects in general

The inclusion of wild-card patterns adds some complexity to the metacomputation algorithms, but it greatly simplifies the conversion from the reflected AST representation.

Typing deserves a special note. Our intermediate language is untyped. On the other hand we rely on the fact, that F# quotations are type-checked by the compiler [4]. This fact gives 2 advantages:

- during driving (and other metacomputation algorithms) we can take fewer precautions if we can rely on an initially type-checked input;
- driving and other transformations preserve typing implicitly, so at the end we can recover a typed expression relatively easily, if needed.

As already hinted by the first example, the F# subset we accept features a rich type system, including tuples, records (with immutable fields) [5], unions, parametric polymorphism.

One more important restriction of our subset is the lack of access to the F# standard libraries. There are two main reasons for this decision:

- standard-library code is already compiled without reflection, and there is no easy way to make it reflected without modifying the library sources (where available) and recompiling a customized version of the standard libraries;
- metacomputation algorithms like driving are traditionally designed for algebraic data types. Adding support for some primitive data types (like `int` or `float`) would probably require the use of external constraint solver, and would in general complicate the implementation by at least an order of magnitude. Besides, many standard-library data types, such as arrays, are essentially mutable. Support for mutability is also deemed too complicated to contemplate at this point.

Still, we support some of the basic built-it types – like `bool`, `option`, `list` – because they can be treated as algebraic. The lack of standard-library support is compensated by a "reflected prelude" containing definitions of some useful data types and functions. (`natAdd` and `natMax` in the introductory example come from this prelude.)

The conversion from code quotation representation to our `Exp` type is mostly straightforward, as the F# compiler has already done its parsing, type inference, pattern-matching compilation and other de-sugaring, before emitting the quotation AST. Actually the amount of de-sugaring is more than we need here: F# match-expressions are converted to a combination of more primitive operations (if-expressions, predicates for testing for a particular head constructor, tuple projections). Our conversion maps those primitive operations back to special kinds of match-expressions, but without any non-local optimizations. The

[4] Unless built by direct calls to AST constructors, but we ignore that possibility here.
[5] Records are de-sugared into tuples, and do not appear explicitly in Fig. 4

net result can be seen in Fig. 5, which shows the conversion of the `treeSize` function (pretty-printed as F# code). The program in this form clearly contains redundancies, but driving is able to remove exactly this kind of redundancies well, so we do not need to perform any special optimizations at the level of our `Exp` representation.

```
let rec treeSize = (fun t ->
 match (match t with
        | EmptyTree -> False
        | Node (_,_,_) -> True) with
 | True ->
  (let r = match t with
           | Node (_,_,x) -> x in
   (let l = match t with
            | Node (_,x,_) -> x in
    NSucc ((ReflectedPrelude.natAdd (treeSize l)) (treeSize r))))
 | False -> NZero)
```

Fig. 5. F# quotation of the `treeSize` function

3 Process-tree Construction

We assume readers are familiar with the basic notions of driving, and do not reiterate them here [6], but instead focus on some of the implementation technical details. We follow the standard approach [4,23] and base our metacomputation toolkit on the notion of a process tree. While some implementations of supercompilation omit the explicit construction of process trees (e.g. [18,12,6,20]), we feel it is a useful concept, unifying different metacomputation techniques and making their implementation more modular.

3.1 Driving and Process Trees

Our representation of process trees is quite standard, very similar to [23,13,15]. We separate the implementation of individual driving steps in a stand-alone function, and then the main driving function simply unfolds the results of driving steps into a tree (Fig. 6). Each driving step takes a "configuration", encoding the current state of the driving process, and produces one of 4 kinds of results, corresponding to the 4 kinds of process tree nodes described below. As the process tree is often infinite and F# is a strict language, we need to explicitly make the process-tree data type lazy enough. (We use the `Lazy<'X>` data type from the F# standard libraries for this purpose.)

We distinguish 4 kinds of process-tree nodes:

- leaves – used when driving cannot proceed further at all (because we reach an atomic value or a free variable);

[6] Good introductions can be found, for example in [23,4].

```
type DriveStep<'Conf, 'ContrHead, 'Contr> =
    | DSDone
    | DSTransient of 'Conf
    | DSDecompose of 'Conf list * ('Conf list -> 'Conf)
    | DSBranch of 'ContrHead * ('Contr * 'Conf) list
let driveStep (conf : DriveConf)
    : DriveStep<DriveConf, VName, Pattern> = ...
type PrTree<'Conf, 'ContrHead, 'Contr> =
    | PTDone of 'Conf
    | PTTransient of 'Conf * Lazy<PrTree<'Conf, 'ContrHead, 'Contr>>
    | PTDecompose of ('Conf * Lazy<PrTree<'Conf, 'ContrHead, 'Contr>>)
        list * ('Conf list -> 'Conf)
    | PTBranch of 'ContrHead * ('Contr * 'Conf
        * Lazy<PrTree<'Conf, 'ContrHead, 'Contr>>) list
let rec drive (conf : DriveConf)
    : PrTree<DriveConf, VName, Pattern> = ...
```

Fig. 6. Driving and process trees

- branches – used when driving gets blocked by a match-expression scrutinizing a free variable. The edges to the sub-trees of such a node are labeled with "contractions", which encode the conditions for selecting one of the sub-trees. It turns useful to factor out the common part of all contractions in the branch node itself – represented as `'ContrHead` in Fig. 6;
- transient nodes – when the driving step amounts to (deterministic) weak reduction;
- decomposition nodes – used in different cases:
  - when driving reaches a constructor of non-0 arity, we can continue driving the constructor arguments in parallel;
  - when we need to perform generalization during supercompilation, in order to obtain a finite (but imperfect) process tree;
  - when driving get stuck at an expression with sub-expressions, and none of the above cases applies.

In the current setting, the last subcase amounts to reaching an expression of the form `match` $fx_1x_2...x_n$ `with ...`, where f is a free variable of a function type, and $n > 0$. In the ideal case, we should create a branch node, but that would mean using a more complicated representation of contractions, and some form of higher-order unification. As in other supercompilers for higher-order languages, we take a simpler route by decomposing such expressions and continuing to drive their sub-expressions in parallel.

3.2 Driving Configurations

A technical subtlety arises with the choice of configuration representation: our F# subset preserves the possibility of the full language to have arbitrary nested let(rec) expressions. Direct treatment of let-, and especially letrec-, expressions can complicate substantially the definition of driving, so it is common to assume, that at least recursive definitions are lambda-lifted to the top level [18,20,17].

But first, we make an important decision – to stick to call-by-name (CBN) driving, even if our object language is call-by-value (CBV). This approach is classic – taken already by the original work on driving for Refal. In the absence of side effects, the semantic difference between CBN and CBV is mostly of theoretical interest. More important pragmatically is the potential loss of sharing, compared with call-by-need driving methods [18,17,6]. We argue, however, that this loss of sharing is critical mostly in the context of supercompilers used as a part of optimizing compilers. As we are interested in other applications of metacomputation techniques, the simplicity and the potential for more reductions offered by call-by-name driving is deemed more important.

Sticking to call-by-name enables us to introduce yet another intermediate representation of expressions, based on closures (Fig. 7), inspired by [7,8]. It is, obviously, easy to convert between the `Exp` and `ClosedExp` representations. The new encoding brings the following advantages:

- no need to deal explicitly with let-expressions or substitutions during driving;
- an especially unobtrusive treatment of recursion;
- no worries about variable capture, at least in the context of weak reductions.

```
type CaseAlts = (Pattern * Exp) list
type EnvEntry<'V> =
    | EEVarBind of VName * 'V
    | EERecDefs of (VName * Exp) list
type Env<'V> = EnvEntry<'V> list
type ClosedExp =
    | CEVar of VName * Env<ClosedExp>
    | CEClosure of BindPattern * Exp * Env<ClosedExp>
    | CEApp of ClosedExp * ClosedExp
    | CECon of CName * ClosedExp list
    | CECase of Exp * CaseAlts * Env<ClosedExp>
```

Fig. 7. Closure-based expression representation

To keep track of the context of the current redex we use a stack of evaluation contexts, similar to [6]. We also need a counter for generating fresh variables when reduction descends under a binder. The final representation of driving configurations is shown on Fig. 8.

```
type DriveContextEntry =
    | DCEApp of ClosedExp
    | DCECase of CaseAlts * Env<ExpOrClosed>
type DriveContext = DriveContextEntry list
type DriveConf = int * ClosedExp * DriveContext
```

Fig. 8. Driving configurations

3.3 Optimizations of the Driving Implementation

While the closure-based representation of expressions has many advantages, it comes at a cost: we have to constantly switch between the 2 representations (`Exp` and `ClosedExp`) during driving and further metacomputation algorithms (when we look up a variable binding inside an environment; when driving reaches a (weak) redex, whose head expression has a binder; when we need to perform driving under a binder; when we propagate a contraction inside the sub-trees of a branch node). The repeated re-traversals and re-creations of parts of the driven expression incur a high cost both in terms of processor time and memory consumption. We found 2 measures, that helped reduce significantly this cost.

Firstly, the closure-based representation can be modified slightly, as shown on Fig. 9. At certain key places (simple environment bindings, and the scrutinee of a match-expression) we permit both representations. The advantage of this flexibility is that we can delay in many situations the conversion from `Exp` to `ClosedExp` or vice-versa. Further, when the same sub-expression undergoes a sequence of conversions from one form to the other, adjacent pairs of such conversions can now cancel each other. We have to force the conversion typically only when driving reaches a variable reference, whose binding in the corresponding environment is not in the form needed.

The second measure involves performing a limited form of environment pruning (currently – only when converting an `EVar` into a `CEVar`). The limited form we have chosen is inexpensive both in terms of time and memory allocations, while permitting in many cases to seriously reduce the size of environments stored in closures. As these environments may need to be converted back to let(rec)-expressions at a certain point, the saving can be substantial.

```
type ClosedExp =
    | CEVar of VName * Env<ExpOrClosed>
    | CEClosure of BindPattern * Exp * Env<ExpOrClosed>
    | CEApp of ClosedExp * ClosedExp
    | CECon of CName * ClosedExp list
    | CECase of ExpOrClosed * CaseAlts * Env<ExpOrClosed>
and ExpOrClosed =
    | EOCExp of Exp
    | EOCClosed of ClosedExp
```

Fig. 9. Optimized closure-based representation

4 Process-tree Based Program Tabulation and Inversion

Let's briefly recall the idea of program tabulation, which was historically developed as a key initial step in the URA method for program inversion [4,1]. Assuming a programming language L, a data domain D, and evaluation function $[\![\_]\!] : L \to D \to D$, the tabulation of a program $p \in L$ can be defined in

general as producing – from a program and a subset of the data domain – a sequence of pairs:

$$Tab(p, D_{in}) = (D_{in}^{(1)}, f_1), (D_{in}^{(2)}, f_2), ...$$

where:

$$D_{in}, D_{in}^{(i)} \subseteq D;\ D_{in}^{(i)} \cap D_{in}^{(j)} = \emptyset \text{ for } i \neq j;\ \bigcup_{i=1}^{\infty} D_{in}^{(i)} = D_{in};$$
$$f_i : D_{in}^{(i)} \to D, i = 1, 2, ...$$
$$\forall i \forall d \in D_{in}^{(i)} (f_i(d) = [\![p]\!]\, d)$$

and with the further requirement that, given any i, for each $d \in D_{in}^{(i)}$ the computation of $[\![p]\!]\, d$ takes the same path in the branches of the process tree.

We follow the established approach for performing tabulation ([4,1,2]): first build the process tree, then traverse it in breath-first order (to ensure completeness), collecting contractions from branch nodes along the way. Each time we reach a leaf in the tree, we can produce a new input-output pair in the table: each $D_{in}^{(i)}$ is the composition of contractions along the path from the root applied to D_{in} (all $D_{in}^{(i)}$ take thus the form of first-order substitutions); and $f_i = e$, where e is the expression extracted from the leaf configuration. Note that both $D_{in}^{(i)}$ and e can contain free variables; binding them to different values gives different elements in the set $D_{in}^{(i)}$, and substituting these values in e gives the ground result associated with the corresponding input.

The algorithm outlined above considers only process trees containing leaves and branch nodes. Indeed, the original implementations of URA [4,1] dealt with first-order flat functional languages (S-Graph, TSG), Extensions were later developed for non-flat, but still first-order languages [2,15]. Still, it appears that those extensions either do not produce transient/decomposition nodes, or ignore the possibility of such nodes during tabulation. Transient nodes are easy – we can simply skip them during tabulation. As for decomposition nodes, recall that in the current setting they can be of 3 kinds:

1. corresponding to a constructor of non-0 arity;
2. corresponding to a λ-expression, which will appear in the program result;
3. corresponding to an expression of the form `match` $fx_1x_2...x_n$ `with ...`, with $n > 0$ and f a free variable of a function type

We decided to ignore the latter 2 kinds during tabulation, with one exception: nodes of the second kind immediately below the root of the process tree. Such nodes correspond to the program arguments, and we simply traverse them collecting the set of free variables, to build D_{in}. As for other non-constructor decomposition nodes appearing inside the tree, they seem not so important for the practical applications we currently have in mind, as their presence would imply at least one of the following:

– first-class functions embedded inside the result of the tabulated function;

– first-class functions appearing as (sub-values of) arguments of the tabulated function.

Our current strategy is to skip processing the corresponding sub-tree, which renders the method incomplete, but at least allows us to find other existing solutions, even if the tabulation request violates the above assumptions. Note also that this restriction in no way prevents many typical uses of higher-order functions – for example in conjunction with list functions such as `map` or `filter`. (See the examples in the next section.) We should only avoid higher-order functions used as data – nested inside input or output values. Thus our restriction is very similar to existing restrictions in most higher-order functional languages on performing input/output with values embedding higher-order functions.

This still leaves us with the problem of how to handle constructor decomposition nodes during tabulation. In fact, we have not yet found a suitable solution for the tabulation algorithm based on breadth-first search. BFS in itself proved to be a practical problem, as it can be very memory-consuming (because of the need to keep a queue with all nodes of the level of the tree currently being scanned). There is a well-known alternative to BFS, which trades a small constant-factor increase in time complexity for big asymptotic improvement in memory consumption – iterative deepening [21]. While we have not implemented yet a version of tabulation that performs full iterative-deepening search, we have one based on depth-limited search, which is the key component of iterative deepening. To further optimize memory consumption, we do not build the process tree explicitly before tabulation, but use directly `driveStep` (Fig. 6), producing a "virtual" version of the process tree for traversal.

This version of depth-limited tabulation also enables an easy (even if somewhat brute-force) solution to the problem of inner constructor nodes. The basic idea is as follows:

– build all input-output tables – tab_i $(i = 1..n)$ – corresponding to the n subtrees of the constructor node; as we are using depth-limited search, each tab_i is guaranteed to be finite;
– construct the Cartesian product of all tab_i:

$$cp = \{((s_1, e_1), ..., (s_n, e_n)) | \ \forall i (s_i, e_i) \in tab_i\}$$

– for each $((s_1, e_1), ..., (s_n, e_n)) \in cp$, if `mergeManySubsts [` s_1; ...; s_n `] = Some` s, we can yield a new table entry for the constructor node: $(s, C(e_1, ..., e_n))$, where C is the corresponding constructor.

The function `mergeManySubsts` simply combines several first-order substations, ensuring that if the same variable is bound in several substitutions, the corresponding most general unifier exists, and the variable is bound to the unification result.

5 Black-box Tests Based on Partitioning Specifications

We return to the discussion of one potential application of our F# metacomputation toolkit – generation of partition-based black-box tests. Black-box testing

requires creation of "interesting" test sets for programs $p \in L$, whose internal structure is unknown. One well-proven heuristics for building such tests is the method of "equivalence partitioning" [5]. The idea is to devise a partition (typically – finite) of the data domain: $D = \bigcup_i D_i$ $(i \neq j \rightarrow D_i \cap D_j = \emptyset)$; each such partition defines an equivalence relation over D. The intuition behind the method is that if we define a suitable partition, the data points in each of the corresponding equivalence classes will be treated in a similar manner by the program under test, and so it will be sufficient to take only one or a few tests from each equivalence class. One way to specify such a partition is by a function $f : D \rightarrow X$, where $X = \{x_1, x_2, ..., x_n\} \subseteq D$ is some finite data type. Then take $D_i := \{d \in D \mid f(d) = x_i\}$. If we also assume that the specification is complete, in the sense that $\forall i \exists d \in D \ : \ f(d) = x_i$, we can use the program tabulation method described in the previous section to generate representatives of the equivalence classes. It suffices to tabulate f, and to group entries in the resulting table by output value (x_i). Then, for each x_i, we can select one or several input tests from the corresponding $D_{in}^{(i)}$.

Let's look at a few more examples illustrating this idea. First, consider a program dealing with a very simple imperative language, containing assignments, sequences and while-loops (Fig. 10). We leave unspecified the data types for variables and expressions. Two simple quantitative measures on programs are introduced – statement count, and loop-nesting depth, and we use them to define partitioning of the space of programs – the expression used in the tabulation request is shown on Fig. 11. (Note that we limit the search to programs of nesting ≤ 2 and statement count ≤ 3.) The result of tabulation appears in Fig. 12; we can see 6 equivalence classes, and by instantiating the free variables with some suitable values we can obtain test inputs for each class.

```
type Stmt<'V, 'E> =
    | Assign of 'V * 'E
    | Seq of Stmt<'V, 'E> * Stmt<'V, 'E>
    | While of 'E * Stmt<'V, 'E>

[<ReflectedDefinition>]
let rec stmtCount (s: Stmt<'V, 'E>) : Nat = ...

[<ReflectedDefinition>]
let rec loopNesting (s: Stmt<'V, 'E>) : Nat = ...
```

Fig. 10. Simple imperative language

The next example is from a different domain (a scaled-down version of a real-world use case): programs dealing with electrical diagrams often have wire-lists as inputs. Wire connections usually form an acyclic graph, with each connected component representing an equipotential. We can thus represent wire-lists as forests, with each tree being a set of electrically connected wires and pins (Fig. 13). A possible tabulation request – defining a partition by (equipotential-count,

```
<@ fun s ->
    let sCnt = WhileL.stmtCount s
    let nestCnt = WhileL.loopNesting s
    if boolAnd (natLE nestCnt (NSucc(NSucc(NZero))))
       (natLE sCnt (NSucc(NSucc(NSucc(NZero)))))
    then Some (nestCnt, sCnt) else None
    @>
```

Fig. 11. Tabulation request for imperative language programs

```
[(Some (Tuple2 (NSucc (NSucc (NZero)),NSucc (NSucc (NSucc (NZero))))),
  [map [(s_0, While (__2,While (__4,Assign (__6,__5))))]]);
 (Some (Tuple2 (NSucc (NZero),NSucc (NSucc (NSucc (NZero))))),
  [map [(s_0, While (__2,Seq (Assign (__6,__5),Assign (__8,__7))))];
   map [(s_0, Seq (While (__4,Assign (__8,__7)),Assign (__6,__5)))];
   map [(s_0, Seq (Assign (__4,__3),While (__6,Assign (__8,__7))))]]);
 (Some (Tuple2 (NSucc (NZero),NSucc (NSucc (NZero)))),
  [map [(s_0, While (__2,Assign (__4,__3)))]]);
 (Some (Tuple2 (NZero,NSucc (NSucc (NSucc (NZero))))),
  [map
     [(s_0, Seq (Seq (Assign (__6,__5),Assign (__8,__7)),Assign (__10,__9)))];
   map
     [(s_0, Seq (Assign (__4,__3),Seq (Assign (__8,__7),Assign (__10,__9))))]]);
 (Some (Tuple2 (NZero,NSucc (NSucc (NZero)))),
  [map [(s_0, Seq (Assign (__4,__3),Assign (__6,__5)))]]);
 (Some (Tuple2 (NZero,NSucc (NZero))), [map [(s_0, Assign (__2,__1))]])]
```

Fig. 12. Imperative language tabulation result

wire-count) – is shown in Fig. 14. The list of results is too long to include, but the number of different entries in each equivalence class (after taking only the first 30 non-`None` results, with a depth limit of 25) is summarized in Table 1.

Table 1. Wire-list tabulation results

Equipotential count	Wire count	Number of entries
2	3	12
2	2	5
2	1	2
2	0	1
1	3	5
1	2	2
1	1	1
1	0	1
0	0	1

What is remarkable in this example is, that the partition specification requires some slightly more involved processing than the previous ones. We can see, though, that the familiar use of higher-order library functions (like `listFoldl`, `listNubBy`) keeps the code succinct. We could no doubt code the same functions

```
type RoseTree<'N, 'E> = RTNode of 'N * ('E * RoseTree<'N, 'E>) list
type PinData<'EQ> = {pinTag: Nat; eqTag: 'EQ}
type WireData<'WC> = {wireColor: 'WC}
type Equipotential<'EQ, 'WC> = RoseTree<PinData<'EQ>, WireData<'WC>>

[<ReflectedDefinition>]
let rec equipotStats (ep: Equipotential<'EQ, 'WC>) : Nat * 'EQ list =
    match ep with
    | RTNode(pd, wds_ts) ->
        let ts = listMap (fun (_, x) ->x) wds_ts
        let wcount1 = listLength wds_ts
        let (wcount2, eqs) = listFoldl (fun (wc, eqs) t ->
            let (wc1, eqs1) = equipotStats t
            (natAdd wc1 wc, listAppend eqs1 eqs)) (NZero, []) ts
        (natAdd wcount1 wcount2, pd.eqTag::eqs)

[<ReflectedDefinition>]
let equipotsStats (eqTagEq: 'EQ -> 'EQ -> bool)
    (eps: Equipotential<'EQ, 'WC> list) : Nat * Nat * 'EQ list =
    let (wc, eqs) = listFoldl (fun (wc1, eqs1) ep ->
        let (wc2, eqs2) = equipotStats ep
        (natAdd wc2 wc1, listAppend eqs2 eqs1)) (NZero, []) eps
    (listLength eps, wc, listNubBy eqTagEq eqs)
```

Fig. 13. Wire-list as a forest, with simple statistics functions

```
<@ fun eps ->
    let (epCnt, wireCnt, eqs) = WList.equipotsStats boolEq eps
    if boolAnd (natLE epCnt (NSucc(NSucc(NZero))))
        (natLE wireCnt (NSucc(NSucc(NSucc(NZero)))))
    then Some (epCnt, wireCnt) else None
    @>
```

Fig. 14. A tabulation request for wire-list test generation

in a first-order language, but some code-size increase and loss of modularity seems inevitable, even in such small cases. We can also note the use of another trick for controlling the size of the search space: the type for equipment tags is abstracted as a parameter in the wire-list definition; we instantiate it with `bool` in the tabulation request, to limit the number of different equipment tags appearing in test cases.

We have performed further experiments, which we do not describe in detail here. Still it is worth noting that we stumbled upon certain restrictions of this test-generation technique. One of the examples defines a Church-style version of the simply-typed lambda calculus, together with a type-checker. If we try to generate directly only well-typed lambda terms for testing, with a reasonably small depth limit, the method tends to generate mostly trivially well-typed terms of the form $\lambda x_1 x_2 ... x_n . x_i$. On the other hand, the tabulation technique is also very flexible – in the same lambda-calculus sample, it is possible to "nudge" the tabulator towards more interesting well-typed terms, by partially specifying the shape of types in the typing environment of the type-checker, ensuring that there are more suitable types for forming applications.

6 Related Work

As already mentioned on several occasions, the pioneering work on metacomputation – both supercompilation and related techniques – was done by Turchin and his students, using Refal [26]. Later many key methods were re-developed using simpler languages like S-Graph [9,4,1,3]. In recent years, there is renewed interest and activity in different supercompiler implementation, for example [18,14,12,11,6,20], to cite a few. In comparison, other metacomputation methods seem neglected, maybe with the exception of [15].

There is extensive literature on software testing, and in particular, on black-box and equivalence partitioning testing [5]. Another interesting testing method is based on metacomputation techniques as well – neighborhood testing [4,3]. While a detailed comparison with the technique outlined here is out of scope, we can note important high-level differences. Neighborhood testing is a white-box testing method, requiring access to the source of the unit under test (and, if available, also its executable specification). It can be used – in principle – for any language, as long as we have an interpreter for that language, written in the language, for which a neighborhood analyzer exists. We consider neighborhood testing being a "second-order" meta-technique – requiring 2 meta-system transitions – to be the main difficulty for practical application. These two levels of interpretative overhead will probably make it harder to achieve sufficient performance in practical implementations. On the other hand, neighborhood testing is a much more powerful method (in certain formal sense subsuming all existing test coverage criteria), so its successful practical application remains an interesting area of study.

Other test-generation methods apply techniques similar to driving – usually under the umbrella term of "symbolic execution". A recent variation – dynamic

symbolic execution – is employed in successful test-generation tools like Microsoft Pex [10]. Again, we omit a detailed comparison between dynamic symbolic execution and driving, noting instead some pragmatic differences with Pex. Pex is heavily geared towards supporting well idiomatic code written in OO .NET languages like C#. It handles surprisingly well complicated program logic using a combination of different built-in types. When faced with data types exhibiting deep recursive structure, however, it usually requires help from the user to generate good tests [7]. The approach we propose here deals well with recursive algebraic data types, so it can be seen as complementary to tools like Pex.

There are many methods for generating test data with specific structure (for example grammar-based, or XML-Schema-based [5]). Specialized tools exist for generating correct test programs in particular programming languages [16,19]. A generic technique for test generation, like the one presented here, can hardly compete in efficiency with such specialized techniques. At the same time, driving-based exploration is very flexible (as seen in most of our examples) and can probably be combined with similar techniques to reduce the search space for test generation (as we hinted for the lambda-calculus example).

We have already noted, that F# code quotations [24] – which helped a lot in making our toolkit feasible with relatively small effort – are very similar in design to languages like MetaML, and language extensions like Template Haskell [25,22]. Similar facilities are starting to appear for other popular languages. We can thus imagine a version of our toolkit for such languages, leveraging on the corresponding meta-programming support.

7 Conclusions and Future Work

We describe an on-going experiment to make more accessible some of the less well-known metacomputation methods (like program tabulation/inversion or neighborhood analysis) – by re-implementing them for a large subset of a popular standard language (F#), which is supported by an efficient run-time and a rich IDE and other developer tools. The F# support for meta-programming (code quotations) greatly facilitated this effort.

To the best of our knowledge, this is the first implementation of URA-like program tabulation for a higher-order functional language. Aside from the special challenges posed by higher-order functions, we rely on already established metacomputation techniques. We described parts of the implementation in more detail, in the hope that some of the insights, gained in attempting to make driving and tabulation reasonably efficient, can be useful in similar contexts. An interesting practical application of program tabulation is also described in some detail – generating black-box tests from partitioning specifications.

Our system, in its current state, can handle successfully moderately-sized programs, and generate small-sized partition-based tests (similar in size or slightly

[7] It is perfectly possible that many of the current limitations of Pex will be lifted in future versions.

larger than the examples in Sect. 5). Our experiments indicate, that more optimization effort is needed to make the test-generation approach practical in a wider variety of scenarios. It is not clear at this point if modest low-level fine-tuning will be enough, or we need to more drastically re-think some of the key algorithms, such as the treatment of decomposition nodes during tabulation. Another interesting optimization option to try would be to re-use the underlying F# run-time for performing weak reductions, and revert to interpretative driving for treating reduction under binders, and for information propagation inside match-expression branches.

There are a lot of possibilities to make the toolkit more "user-friendly". As suggested by one of the reviewers, we could automatically convert some primitive data types (such as `int`) into suitable replacements from the reflected prelude. A non-trivial issue is how to select the best replacement in each case. (We could supply an implementation of arbitrary signed integers in the prelude, but it would be less efficient during tabulation in cases, where the user actually needs only natural numbers.) Another improvement, which is in progress, is to fill holes in generated tests by arbitrary values of the appropriate type and convert back each test into a normal F# value.

Apart from the potentially interesting future tasks hinted in the previous section, we can list a few more interesting ideas for future experiments:

- extend driving to a full supercompiler for the F# subset; check if program tabulation/inversion can be made more efficient by using the process graphs generated during supercompilation, instead of the potentially infinite process trees;
- modify driving to use call-by-need instead of call-by-name, and see what performance benefits it can bring to metacomputation methods (other than supercompilation);
- implement neighborhood analysis, and experiment with potential practical applications (like neighborhood testing).

8 Acknowledgments

The author would like to thank Ilya Klyuchnikov and Neil Mitchell for the helpful comments and suggestions for improving the presentation in this article.

References

1. Abramov, S., Glück, R.: The universal resolving algorithm and its correctness: inverse computation in a functional language. Science of Computer Programming 43(2-3), 193–229 (2002)
2. Abramov, S., Glück, R., Klimov, Y.: An universal resolving algorithm for inverse computation of lazy languages. Perspectives of Systems Informatics pp. 27–40 (2007)

3. 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)
4. Abramov, S.M.: Metavychisleniya i ih primenenie (Metacomputation and its applications). Nauka, Moscow (1995)
5. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008)
6. Bolingbroke, M., Peyton Jones, S.: Supercompilation by evaluation. In: Proceedings of the third ACM Haskell symposium on Haskell. pp. 135–146. ACM (2010)
7. Clément, D., Despeyroux, T., Kahn, G., Despeyroux, J.: A simple applicative language: Mini-ML. In: Proceedings of the 1986 ACM conference on LISP and functional programming. pp. 13–27. ACM (1986)
8. Coquand, T., Kinoshita, Y., Nordström, B., Takeyama, M.: A simple type-theoretic language: Mini-TT. From Semantics to Computer Science: Essays in Honour of Gilles Kahn (2009)
9. Glück, R., Klimov, A.V.: Occam's razor in metacomputation: the notion of a perfect process tree. In: Cousot, P., Falaschi, M., Filé, G., Rauzy, A. (eds.) Static Analysis. Proceedings. Lecture Notes in Computer Science, vol. 724, pp. 112–123. Springer-Verlag (1993)
10. Godefroid, P., de Halleux, P., Nori, A., Rajamani, S., Schulte, W., Tillmann, N., Levin, M.: Automating software testing using program analysis. Software, IEEE 25(5), 30–37 (2008)
11. Hamilton, G.: Distillation: extracting the essence of programs. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 61–70. ACM (2007)
12. Jonsson, P.A., Nordlander, J.: Positive supercompilation for a higher order call-by-value language. SIGPLAN Not. 44(1), 277–288 (Jan 2009)
13. Klyuchnikov, I.: The ideas and methods of supercompilation. Practice of Functional Programming (7) (2011), in Russian
14. Klyuchnikov, I., Romanenko, S.: Proving the equivalence of higher-order terms by means of supercompilation. In: Perspectives of Systems Informatics'09. pp. 193–205. Springer (2010)
15. Klyuchnikov, I.: GitHub project: "Metacomputation and its Applications" - now for SLL (2011), `https://github.com/ilya-klyuchnikov/sll-meta-haskell`, [Online; accessed 13-March-2012]
16. Koopman, P., Plasmeijer, R.: Systematic synthesis of functions. In: H. Nilsson (ed.), Selected papers from the Seventh Symposium on Trends in Functional Programming (TFP06), Nottingham, United Kingdom, 19-21 April 2006. pp. 35–54. Bristol: Intellect Books (2006)
17. Mitchell, N.: Rethinking supercompilation. In: ACM SIGPLAN Notices. vol. 45, pp. 309–320. ACM (2010)
18. Mitchell, N., Runciman, C.: A supercompiler for core Haskell. In: et al., O.C. (ed.) IFL 2007. LNCS, vol. 5083, pp. 147–164. Springer-Verlag (May 2008)
19. Reich, J.S., Naylor, M., Runciman, C.: Lazy generation of canonical programs. In: 23rd Symposium on Implementation and Application of Functional Languages (2011)
20. Reich, J., Naylor, M., Runciman, C.: Supercompilation and the Reduceron. In: Proceedings of the Second International Workshop on Metacomputation in Russia (2010)
21. Russell, S., Norvig, P.: Artificial intelligence: a modern approach. Prentice Hall series in artificial intelligence, Prentice Hall (2003)

22. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. pp. 1–16. ACM (2002)
23. 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)
24. Syme, D.: Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In: Proceedings of the 2006 workshop on ML. pp. 43–54. ACM (2006)
25. Taha, W., Sheard, T.: MetaML and multi-stage programming with explicit annotations. Theoretical computer science 248(1-2), 211–242 (2000)
26. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (July 1986)

Development of the Productive Forces

Gavin E. Mendel-Gleason[1] and Geoff W. Hamilton[2]

[1] School of Computing, Dublin City University ggleason@computing.dcu.ie
[2] School of Computing, Dublin City University hamiltion@computing.dcu.ie

Abstract. Proofs involving infinite structures can use corecursive functions as inhabitants of a corecursive type. Admissibility of such functions in theorem provers such as Coq or Agda, requires that these functions are productive. Typically this is proved by showing satisfaction of a guardedness condition. The guardedness condition however is extremely restrictive and many programs which are in fact productive and therefore will not compromise soundness are nonetheless rejected. Supercompilation is a family of program transformations which retain program equivalence. Using supercompilation we can take programs whose productivity is suspected and transform them into programs for which guardedness is syntactically apparent.

1 Introduction

The Curry-Howard correspondence lets us take advantage of a close correspondence between programs and proofs. The idea is that inhabitation of a type is given by demonstration of a type correct program having that type. In the traditional Curry-Howard regime type inhabitation is given with terminating programs. This requirement avoids the difficulty that non-terminating programs might allow unsound propositions to be proved.

This was extended by Coquand [6] to include potentially infinite structures. The notions of co-induction and co-recursion allow infinite structures and infinite proofs to be introduced without loss of soundness.

In the Agda and Coq theorem provers, the guardedness condition is the condition checked to ensure admissibility of coinductive programs. This condition ensures that the function is *productive* in the sense that it will always produce a new constructor finitely. The guardedness condition has the advantage of only requiring a syntactic check on the form of the program and is a sufficient condition for productivity. However, it suffers from being highly restrictive and rejects many programs which *are* in fact productive. Requiring guards is a rather rigid approach to ensuring productivity.

The purpose of this paper is to suggest that theorem provers can use a notion of pre-proofs[4], proofs for which the side conditions which ensure soundness have not been demonstrated, which can be transformed into a genuine proof using equational reasoning. The main idea is that, if a program is type-correct after program transformation using equational reasoning, it was correct prior to

transformation. This was the main idea behind the paper by Danielsson et al. [7].

The approach has a number of advantages over the traditional syntactic check or ad-hoc methods of extending the guardedness condition to some larger class. The number of programs which might be accepted automatically can be increased simply by using automated program transformers. All but one of the examples given in this paper were found by way of the supercompilation program transformation algorithm. The approach also allows one to keep the chain of equational reasoning present, allowing an audit trail of correctness.

This paper is structured as follows. First we will present a number of motivating examples. We will give a bird's-eye view of supercompilation followed by a description of cyclic proof and transformations in this framework. We will give a demonstration of how one of the motivating examples can be transformed in this framework. Finally we will end with some possibilities for future development.

2 Examples

In order to motivate program transformation as a means for showing type inhabitation, we present a number of examples in the language Agda. First, we would like to work with the natural numbers which include the point at infinity. This we write as the codata declaration with the familiar constructors of the Peano numbers. We also define the analogous codata for lists, allowing infinite streams.

```
module Productive where

codata ℕ∞ : Set where
  czero : ℕ∞
  csucc : ℕ∞ → ℕ∞

codata CoList (A : Set) : Set where
  [] : CoList A
  _::_ : A → CoList A → CoList A

infixr 40 _+_
_+_ : ℕ∞ → ℕ∞ → ℕ∞
czero + y = y
(csucc x) + y = csucc (x + y)

sumlen : CoList ℕ∞ → ℕ∞
sumlen [] = czero
sumlen (x::xs) = csucc (x + (sumlen xs))
```

When attempting to enter this program into a theorem prover, such as Coq or Agda, the type checker will point out that this program does not meet the guardedness condition. This is due to the fact that there is an intermediate application (of +) which might consume any constructors which *sumlen* produces. However, the program is in fact productive and no such problem occurs in practice. This becomes evident after supercompiling the above term, where we arrive at the following transformed term:

```
mutual
  sumlen_sc : CoList ℕ∞ → ℕ∞
  sumlen_sc [ ] = czero
  sumlen_sc (x::xs) = csucc (aux x xs)
  aux : ℕ∞ → CoList ℕ∞ → ℕ∞
  aux czero xs = sumlen_sc xs
  aux (csucc x) xs = csucc (aux x xs)
```

This term will now be accepted by Agda's type checker. We see that we have basically unrolled the intermediate implication eliminations. We will look more closely at this later in Section 4.

In the above example we managed to remove some intermediate applications for two functions which were both productive. We can also find similar results however by unrolling intermediate inductive steps. The following example, given by Komendaskaya and Bertot [3], relies on this *always eventually* type behaviour. That is, behaviour that will *always* be productive, *eventually* after some finite process. Here the finite process is guaranteed by the inductive character of the natural numbers.

```
data Bool : Set where
  true : Bool
  false : Bool
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
infix 5 if_then_else_
if_then_else_ : {A : Set} → Bool →
  A → A → A
if true then x else y = x
if false then x else y = y
infixr 40 _::_
codata Stream (A : Set) : Set where
  _::_ :A → Stream A → Stream A
le : ℕ → ℕ → Bool
le zero _ = true
le _ zero = false
le (succ x) (succ y) = le x y
pred : ℕ → ℕ
pred zero = zero
pred (succ x) = x
f : Stream ℕ → Stream ℕ
f (x::y::xs) = if (le x y)
  then (x::(f (y::xs)))
  else (f ((pred x)::y::xs))
```

Again we have the problem that the type checker is unable to accept f as being productive as the else clause makes an unguarded call to f though the function is productive as we structurally descend on the argument x which is inductively defined. However, after supercompilation we arrive at the following program.

$$
\begin{aligned}
&\mathit{mutual}\\
&\quad f\_sc : \mathit{Stream}\ \mathbb{N} \to \mathit{Stream}\ \mathbb{N}\\
&\quad f\_sc\ (\mathit{zero}{::}y\qquad {::}xs) = \mathit{zero}{::}(f\_sc\ (y{::}xs))\\
&\quad f\_sc\ (\mathit{succ}\ x{::}\mathit{zero}{::}xs) = g\ x\ xs\\
&\quad f\_sc\ (\mathit{succ}\ x{::}\mathit{succ}\ y{::}xs)\ \mathit{with}\ \mathit{le}\ x\ y\\
&\quad \ldots \mid \mathit{true} = (\mathit{succ}\ x){::}(f\_sc\ ((\mathit{succ}\ y){::}xs))\\
&\quad \ldots \mid \mathit{false} = h\ x\ y\ xs\\
&\quad g : \mathbb{N} \to \mathit{Stream}\ \mathbb{N} \to \mathit{Stream}\ \mathbb{N}\\
&\quad g\ \mathit{zero}\ xs = \mathit{zero}{::}(f\_sc\ (\mathit{zero}{::}xs))\\
&\quad g\ (\mathit{succ}\ x)\ xs = g\ x\ xs\\
&\quad h : \mathbb{N} \to \mathbb{N} \to \mathit{Stream}\ \mathbb{N} \to \mathit{Stream}\ \mathbb{N}\\
&\quad h\ \mathit{zero}\ y\ xs = \mathit{zero}{::}(f\_sc\ ((\mathit{succ}\ y){::}xs))\\
&\quad h\ (\mathit{succ}\ x)\ y\ xs\ \mathit{with}\ \mathit{le}\ x\ y\\
&\quad \ldots \mid \mathit{true} = (\mathit{succ}\ x){::}(f\_sc\ ((\mathit{succ}\ y){::}xs))\\
&\quad \ldots \mid \mathit{false} = h\ x\ y\ xs
\end{aligned}
$$

Now we have a program which passes Agda's type checker. The intermediate computations have been turned into inductive loops and the coinductive function now exhibits a guarded call.

With these motivating examples in hand, we would like to look a bit at the method of program transformation which is used to produce these final programs which are now acceptable by our type checker.

3 Language

The language we present is a functional programming language, which we will call Λ_F with a type system based on System F with recursive types. The use of System F typing allows us to ensure that transitions can be found for any term. Our term language will follow closely on the one used by Abel [2].

In Figure 3 we describe the syntax of the language. The set **TyCtx** describes *type variable contexts* which holds the free type variables in a context. The set **Ctx** describes *variable contexts* which holds information about free term variables, and what type they have. The empty context is denoted $\cdot$, and we extend contexts with variables of a type, or a type variable using $\cup$. The contexts are assumed to be sets.

The unit type is denoted $\mathbb{1}$ and is established as the type of the unit term $()$. Functions types, which are the result of a term $\lambda x\colon A.\ t$ where t has type B are given using $A \to B$. The type of a type abstraction $\Lambda X.\ t$ is given with the syntax $\forall X.\ A$ in which the type variable X may be present in the type A

Variables

$\mathbf{Var} \ni x, y, z$	Variables
$\mathbf{TyVar} \ni X, Y, Z$	Type Variables
$\mathbf{Fun} \ni f, g, h$	Function Symbols

Contexts

$$\mathbf{Ctx} \ni \Gamma ::= \cdot \mid \Gamma \cup \{x : A\}$$
$$\mathbf{TyCtx} \ni \Delta ::= \cdot \mid \Delta \cup \{X\}$$

Types

$$\mathbf{Ty} \ni A, B, C ::= 1 \mid X \mid A \to B \mid \forall X.A \mid A + B \mid A \times B \mid \nu\hat{X}.\ A \mid \mu\hat{X}.\ A$$

Terms

$$\begin{aligned}\mathbf{Tr} \ni r, s, t ::= {} & x \mid f \mid () \mid \lambda x\!:\!A.\ t \mid \Lambda X.\ t \mid r\ s \mid r[A] \mid (t, s) \\ \mid {} & \text{left}(t, A + B) \mid \text{right}(t, A + B) \\ \mid {} & \text{in}_\nu(t, A) \mid \text{out}_\nu(t, A) \mid \text{in}_\mu(t, A) \mid \text{out}_\mu(t, A) \\ \mid {} & \text{case } r \text{ of } \{x_1 \Rightarrow s \mid x_2 \Rightarrow t\} \mid \text{split } r \text{ as } (x_1, x_2) \text{ in } s\end{aligned}$$

Universal Type Variables

$$UV(\Delta \cup \{\hat{X}\}) := UV(\Delta)$$
$$UV(\Delta \cup \{X\}) := \{X\} \cup UV(\Delta)$$

Type Formation

$$\frac{UV(\Delta) \vdash A\ \ \mathtt{type} \qquad \Delta \vdash B\ \ \mathtt{type}}{\Delta \vdash A \to B\ \ \mathtt{type}} \qquad \frac{\Delta \cup \{\hat{X}\} \vdash A\ \ \mathtt{type} \qquad \alpha \in \{\nu,\ \mu\}}{\Delta \vdash \alpha\hat{X}.\ A\ \ \mathtt{type}}$$

$$\frac{\Delta \vdash A\ \ \mathtt{type} \qquad \Delta \vdash B\ \ \mathtt{type} \qquad \circ \in \{+,\ \times\}}{\Delta \vdash A \circ B\ \ \mathtt{type}} \qquad \frac{}{\Delta \vdash 1\ \ \mathtt{type}}$$

$$\frac{\Delta \cup \{\hat{X}\}\ \ \mathtt{TyCtx}}{\Delta \cup \{\hat{X}\} \vdash \hat{X}\ \ \mathtt{type}} \qquad \frac{\Delta \cup \{X\}\ \ \mathtt{TyCtx}}{\Delta \cup \{X\} \vdash X\ \ \mathtt{type}} \qquad \frac{\Delta \cup \{X\} \vdash A\ \ \mathtt{type}}{\Delta \vdash \forall X.\ A\ \ \mathtt{type}}$$

Context Formation

$$\frac{}{\cdot\ \ \mathtt{TyCtx}} \qquad \frac{\Delta\ \ \mathtt{TyCtx} \qquad X \notin \Delta}{\Delta \cup \{X\}\ \ \mathtt{TyCtx}}$$

$$\frac{}{\Delta \vdash \cdot\ \ \mathtt{Ctx}} \qquad \frac{\Delta \vdash \Gamma\ \ \mathtt{Ctx} \qquad x \notin dom(\Gamma) \qquad \Delta \vdash T\ \ \mathtt{type}}{\Delta \vdash \Gamma \cup \{x\!:\!T\}\ \ \mathtt{Ctx}}$$

Fig. 1. Language

and the term t. Sum types allow a term of one of two types, A and B, to be injected into a third type $A+B$. We can inject into this sum for terms of type A using the left injection: $\text{left}(t, A+B)$ or for terms of type B on the right using $\text{right}(s, A+B)$. The pair type $A \times B$ is introduced with a pairing term (s,t) where s has type A and t has type B. We introduce inductive types $\mu\hat{X}.\ A$ with the term $\text{in}_\mu(t, \mu\hat{X}.\ A)$. Similarly for coinductive types $\nu\hat{X}.\ A$ we introduce the term $\text{in}_\nu(t, \nu\hat{X}.\ A)$. Similarly each type (save for unit) is equiped with an elimination term, whose meaning will be clear from the dynamic semantics.

Types are introduced by way of type formation rules such that a type A is well formed if we can derive $\Delta \vdash A$ `type`. These rules ensure that only types which are strictly positive in μ and ν type variables are allowed, while universally quanitifed variables are unrestricted. This is achieved by segregating the type variables introduced using ν and μ using a hat above the type variable, $\hat{X}$.

Term Substitution

$x[x := t]$	$\equiv t$
$x[y := t]$	$\equiv x$ if $x \neq y$
$\texttt{f}[x := t]$	$\equiv \texttt{f}$
$(r\ s)[x := t]$	$\equiv (r[x := t])\ (s[x := t])$
$(\lambda y\colon A.\ r)[x := t]$	$\equiv \lambda y\colon A.\ r[x := t]$ Provided that $\lambda y\colon A.\ r$ is α-converted to use a fresh variable if $y \in \{x\} \cup FV(t)$.
$(\Lambda X.\ r)[x := t]$	$\equiv \Lambda X.\ r[x := t]$
$\text{in}_\alpha(s, A)[x := t]$	$\equiv \text{in}_\alpha(s[x := t], A)$
$\text{out}_\alpha(s, A)[x := t]$	$\equiv \text{out}_\alpha(s[x := t], A)$
$()[x := t]$	$\equiv ()$
$\text{right}(s, A)[x := t]$	$\equiv \text{right}(s[x := t], A)$
$\text{left}(s, A)[x := t]$	$\equiv \text{left}(s[x := t], A)$
$(s, u)[x := t]$	$\equiv (s[x := t], u[x := t])$
$\text{case}\ r\ \text{of}\ \{y \Rightarrow s \mid z \Rightarrow u\}[x := t]$	$\equiv \text{case}\ r[x := t]\ \text{of}\ \{y \Rightarrow s[x := t] \mid z \Rightarrow u[x := t]\}$ Provided that $\lambda y\colon A.\ s$ or $\lambda z\colon A.\ u$ are α-converted to use a fresh variable if $y \in \{x\} \cup FV(t)$ or $z \in \{x\} \cup FV(t)$ respectively.
$\text{split}\ r\ \text{as}\ (y, z)\ \text{in}\ u[x := t]$	$\equiv \text{split}\ r[x := t]\ \text{as}\ (y, z)\ \text{in}\ u[x := t]$ Provided that $\lambda y\colon A.\ \lambda z\colon A.\ u$ is α-converted to use a fresh variable for y or z if $y \in \{x\} \cup FV(t)$ or $z \in \{x\} \cup FV(t)$ respectively.

Fig. 2. Term Substitution

We describe *substitutions* which use a function $FV(t)$ to obtain the free type and term variables from a term. We also choose free variables to be *fresh*, meaning they are chosen from some denumerable set such that they are not present in a given set of variables. A variable chosen in this way we will write as $x = \mathit{fresh}(S)$

Type Substitution on Terms

$$
\begin{array}{ll}
x[X := A] & \equiv x \\
\mathtt{f}[X := A] & \equiv \mathtt{f} \\
()[X := A] & \equiv () \\
(r\ s)[X := A] & \equiv (r[X := A])\ (s[X := A]) \\
(r[A])[X := A] & \equiv (r[X := A])\ (A[X := A]) \\
(\lambda x\colon A.\ r)[X := A] & \equiv \lambda x\colon A[X := A].\ r[X := A] \\
\mathrm{in}_\alpha(s, B)[X := A] & \equiv \mathrm{in}_\alpha(s[X := A], B[X := A]) \\
\mathrm{out}_\alpha(s, B)[X := A] & \equiv \mathrm{out}_\alpha(s[X := A], B[X := A]) \\
\mathrm{right}(s, B)[X := A] & \equiv \mathrm{right}(s[X := A], B[X := A]) \\
\mathrm{left}(s, B)[X := A] & \equiv \mathrm{left}(s[X := A], B[X := A]) \\
(s, u)[X := A] & \equiv (s[X := A], u[X := A]) \\
(\text{case } r \text{ of } \{y \Rightarrow s \mid z \Rightarrow u\})[X := A] & \equiv \text{case } r[X := A] \text{ of} \\
 & \qquad \{\ y \Rightarrow s[X := A] \\
 & \qquad \mid z \Rightarrow u[X := A] \\
(\text{split } r \text{ as } (y, z) \text{ in } u)[X := A] & \equiv \text{split } r[X := A] \text{ as } (y, z) \text{ in } u[X := A]
\end{array}
$$

Type Substitution on Types

$$
\begin{array}{ll}
X[X := A] & \equiv A \\
X[Y := A] & \equiv X \text{ if } X \neq Y \\
\mathbb{1}[X := A] & \equiv \mathbb{1} \\
B + C[X := A] & \equiv B[X := A] + C[X := A] \\
B \times C[X := A] & \equiv B[X := A] \times C[X := A] \\
(B \to C)[X := A] & \equiv B[X := A] \to C[X := A] \\
(\forall Y.\ B)[X := A] & \equiv \forall Y.\ B[X := A] \\
 & \quad \text{Provided that } (\forall Y.\ B) \text{ is } \alpha\text{-converted to use} \\
 & \quad \text{a fresh type-variable if } Y \in \{X\} \cup FV(A). \\
(\Lambda Y.\ r)[X := A] & \equiv \Lambda Y.\ r[X := A] \\
 & \quad \text{Provided that } (\Lambda Y.\ r) \text{ is } \alpha\text{-converted to use} \\
 & \quad \text{a fresh type-variable if } Y \in \{X\} \cup FV(A). \\
(\alpha Y.\ r)[X := A] & \equiv \alpha Y.\ r[X := A],\ \alpha \in \{\nu,\ \mu\} \\
 & \quad \text{Provided that } (\alpha Y.\ r) \text{ is } \alpha\text{-converted to use} \\
 & \quad \text{a fresh type-variable if } Y \in \{X\} \cup FV(A).
\end{array}
$$

Fig. 3. Type Substitution

if it is *fresh* with respect to the set S. Substitutions of a single variable will be written $[X := A]$ or $[x := t]$ for type and term variables respectively. The full operational meaning of substitutions is given in Figure 2 and Figure 3.

Reduction Rules

$$(\lambda x : A.r)\ s \rightsquigarrow_1 r[x := s]\ (\Lambda X.r)\ A \rightsquigarrow_1 r[X := A]$$

$$\mathrm{out}_\alpha(\mathrm{in}_\alpha(r, U), U) \rightsquigarrow_1 r \qquad \mathtt{f} \rightsquigarrow_\delta \Omega(\mathtt{f})$$

$$\text{case left}(r, A + B) \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\} \rightsquigarrow_1 s[x := r]$$

$$\text{case right}(r, A + B) \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\} \rightsquigarrow_1 t[y := r]$$

$$\text{split } (r, s) \text{ as } (x, y) \text{ in } t \rightsquigarrow_1 t[x := r][y := s]$$

Structural Rules

$$\frac{r\ \mathcal{R}\ r'}{r\ \mathcal{R}^s\ r'} \qquad \frac{r\ \mathcal{R}^s\ r'}{r\ s\ \mathcal{R}^s\ r'\ s} \qquad \frac{r\ \mathcal{R}^s\ r'}{r\ A\ \mathcal{R}^s\ r'\ A} \qquad \frac{r\ \mathcal{R}^s\ r'}{\mathrm{out}_\alpha(r, U)\ \mathcal{R}^s\ \mathrm{out}_\alpha(r', U)}$$

$$\frac{r\ \mathcal{R}^s\ r'}{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}\ \mathcal{R}^s\ \text{case } r' \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}$$

$$\frac{r\ \mathcal{R}^s\ r'}{\text{split } r \text{ as } (x, y) \text{ in } t\ \mathcal{R}^s\ \text{split } r' \text{ as } (x, y) \text{ in } t}$$

Evaluation Relations

$$\begin{array}{l} r \rightsquigarrow_n s ::= r\ \rightsquigarrow_1^s\ s \\ r \rightsquigarrow s\ \ ::= r\ \rightsquigarrow_\delta^s\ s \vee r \rightsquigarrow_n s \\ r\ R^+\ s\ ::= r\ R\ s \vee (\exists r'.r\ R\ r' \wedge r'\ R^+\ s) \\ r\ R^*\ s\ ::= r = s \vee r\ R^+\ s \end{array}$$

Fig. 4. Evaluation

We define the evaluation relation $\rightsquigarrow$ in Figure 4. This relation is the usual normal order evaluation relation. It is deterministic and so there is always a unique redex. We take the transitive closure of the relation to be $\rightsquigarrow^*$.

We introduce recursive terms by way of function constants. Although it is possible to encoded these directly in System F, it simplifies the presentation to provide them explicitly. Function constants are drawn from a set **Fun**. We couple our terms with a function Ω which associates a function constant f with a term t, $\Omega(f) = t$, where t may itself contain any function constants in the domain of Ω. We make use of this function in the $\rightsquigarrow$ relation which allows us to unfold recursively defined functions.

For a term t with type A in a context $\Delta; \Gamma$ we write $\Delta\ ; \Gamma \vdash t : A$. A type derivation must be given using the rules given in Figure 5.

$$\frac{\Delta \vdash \Gamma \cup \{x : A\}\ \ \mathtt{Ctx}}{\Delta\,;\Gamma \cup \{x : A\} \vdash x : A}\ I^{Var} \qquad \frac{}{\Delta\,;\Gamma \vdash () : \mathbb{1}}\ I^{1}$$

$$\frac{\Delta \cup \{X\}\,;\Gamma \vdash t : A}{\Delta\,;\Gamma \vdash (\Lambda X.\ t) : \forall X.\ A}\ I^{\forall} \qquad \frac{\Delta\,;\Gamma \vdash t : \forall X.\ A \qquad \Delta \vdash B\ \ \mathtt{type}}{\Delta\,;\Gamma \vdash t[B] : A[X := B]}\ E^{\forall}$$

$$\frac{\Delta\,;\Gamma \cup \{x : A\} \vdash t : B}{\Delta\,;\Gamma \vdash (\lambda x : A.\ t) : A \to B}\ I^{\to} \qquad \frac{\Delta\,;\Gamma \vdash r : A \to B \qquad \Delta\,;\Gamma \vdash s : A}{\Delta\,;\Gamma \vdash (r\ s) : B}\ E^{\to}$$

$$\frac{\Gamma \vdash \Omega(f) : A}{\Delta\,;\Gamma \vdash f : A}\ I^{\delta} \qquad \frac{\Delta\,;\Gamma \vdash r : A \qquad \Delta\,;\Gamma \vdash s : B}{\Delta\,;\Gamma \vdash (r, s) : (A \times B)}\ I^{\times}$$

$$\frac{\Delta\,;\Gamma \vdash t : A \qquad \Delta \vdash B\ \ \mathtt{type}}{\Delta\,;\Gamma \vdash \mathrm{left}(t, A + B) : (A + B)}\ \mathrm{I}^{+}_{L} \qquad \frac{\Delta\,;\Gamma \vdash t : B \qquad \Delta \vdash A\ \ \mathtt{type}}{\Delta\,;\Gamma \vdash \mathrm{right}(t, A + B) : (A + B)}\ \mathrm{I}^{+}_{R}$$

$$\frac{\Delta\,;\Gamma \vdash t : \alpha\hat{X}.\ A \qquad \alpha \in \{\mu, \nu\}}{\Delta\,;\Gamma \vdash \mathrm{out}_{\alpha}(t, \alpha\hat{X}.\ A) : A[\hat{X} := \alpha\hat{X}.\ A]}\ \mathrm{E}^{\alpha}$$

$$\frac{\Delta\,;\Gamma \vdash t : A[\hat{X} := \alpha\hat{X}.\ A] \qquad \alpha \in \{\mu, \nu\}}{\Delta\,;\Gamma \vdash \mathrm{in}_{\alpha}(t, \alpha\hat{X}.\ A) : \alpha\hat{X}.\ A}\ \mathrm{I}^{\alpha}$$

$$\frac{\Delta\,;\Gamma \vdash r : A + B \qquad \Delta\,;\Gamma \cup \{x : A\} \vdash t : C \qquad \Delta\,;\Gamma \cup \{y : B\} \vdash s : C}{\Delta\,;\Gamma \vdash (\mathrm{case}\ r\ \mathrm{of}\ \{x \Rightarrow t \mid y \Rightarrow s\}) : C}\ E^{+}$$

$$\frac{\Delta\,;\Gamma \vdash s : A \times B \qquad \Delta\,;\Gamma \cup \{x : A\} \cup \{y : B\} \vdash t : C}{\Delta\,;\Gamma \vdash (\mathrm{split}\ s\ \mathrm{as}\ (x, y)\ \mathrm{in}\ t) : C}\ E^{\times}$$

Fig. 5. System F Proof Rules

Without further restrictions, this type system is unsound. First, the Delta rule for function constants clearly allows non-termination given $\Omega(f) = f : T$. We will deal with this potentiality later when we describe cyclic proof in Section 4.

In addition, we will need a concept of a one-hole context. This allows us to describe terms which are embedded in a surrounding term. We write this as $C[t]$ when we wish to say that the term t has a surrounding context.

4 Cyclic Proof

Typically, in functional programming languages, type checking for defined functions is done by use of a typing rule that assumes the type of the function and proceeds to check the body. This is the familiar rule from programming languages such as Haskell [10] [15] [13]. An example of such a typing rule is as follows:

$$\frac{\Delta\,;\Gamma \cup \{f : A \to B\} \vdash \Omega(f) \,:\, A \to B}{\Delta\,;\Gamma \vdash f \,:\, A \to B}\; FunRec$$

Coupled with guardedness or structural recursion and positivity restrictions on the form of recursive types to ensure (co)termination, this rule will be sound. However, it is also *opaque*. Any transformation of this proof tree will be rigidly expressed in terms of the original function declarations.

In order to allow more fluidity in the structure of our proof trees we introduce a notion of a cyclic proof. Cyclicity can be introduced simply by allowing the type rules to be a coinductive type (in the meta-logic) rather than an inductive one. However, for us to produce the cycles we are interested in, we need to add an additional term and typing rule which allows explicit substitutions, and one derived rule which makes use of the fact that we identify all proofs under the reduction relation $\rightsquigarrow$ as being equivalent. The explicit substitutions will also require an additional evaluation rule which transforms them into computed substitutions. Explicit substitutions can also be introduced at the level of type substitutions, but these are not necessary for our examples.

The Conv, E^{Ω} and the I^{θ} follow from theorems about the calculus which can be established in the meta-logic and in fact we have a formal proof of these theorems for the given calculus in Coq.

We will not prove general soundness conditions, but rely on prior work showing that structural induction and the guardedness are sufficient conditions [9]. Once these conditions have been satisfied, we can assume the correctness of the proof.

Definition 1 (Structural Ordering). *A term t is said to be* less in the structural ordering *than a term s, or* $t <^s s$ *using the relation* $<^s$ *given by the inductive definition in Figure 6.*

Definition 2 (Structural Recursion). *A derivation is said to be structurally recursive if for every sequent used in a* I^{θ} *rule, there exists a privileged variable*

Explicit Substitutions

$t\ \langle x := s \rangle$

Typing Rules

$$\frac{\Delta\,;\Gamma \cup \Gamma' \vdash u\ :\ B \qquad \Delta\,;\Gamma \cup \{x\!:\!B\} \vdash t\ :\ A}{\Delta\,;\Gamma \cup \Gamma' \vdash t\ \langle x := u \rangle\ :\ A}\ I^{\theta}$$

$$\frac{\Delta\,;\Gamma \vdash t\ :\ A \qquad t \rightsquigarrow^{*} s}{\Delta\,;\Gamma \vdash s\ :\ A}\ \text{Conv} \qquad \frac{\Delta\,;\Gamma \vdash C[\Omega(f)]\ :\ A}{\Delta\,;\Gamma \vdash C[f]\ :\ A}\ E^{\Omega}$$

Extended Evaluation

$$t\ \langle x := u \rangle \rightsquigarrow t[x := u] \qquad \frac{t \rightsquigarrow t'}{t\ \langle x := u \rangle \rightsquigarrow t'\ \langle x := u \rangle}$$

Structural Ordering

$$\frac{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}{x\ \mathcal{S}\ r} \qquad \frac{\text{case } r \text{ of } \{x \Rightarrow s \mid y \Rightarrow t\}}{y\ \mathcal{S}\ r}$$

$$\frac{\text{split } r \text{ as } (x, y) \text{ in } t}{x\ \mathcal{S}\ r} \qquad \frac{\text{split } r \text{ as } (x, y) \text{ in } t}{y\ \mathcal{S}\ r}$$

$$\frac{\text{out}_{\alpha}(t, \alpha\hat{X}.\ A)}{\text{out}_{\alpha}(t, \alpha\hat{X}.\ A)\ \mathcal{S}\ t} \qquad <^{s} := S^{*} \quad \text{Transitive closure of } S$$

Fig. 6. Explicit Substitution and Structural Ordering

x such that for all I^θ rules, with substitution σ_i, using that sequent we have that $x \in dom(\sigma_i)$ and $\sigma(x) <^s x$.

It should be mentioned that there is nothing in particular needed for this definition aside from some guarantee that we are well founded. As such this represents a particular implementation *strategy* and we could very well have used a more liberal approach. One such approach is *size-change termination* as described by Neil Jones et al. in [11].

Similarly, we must produce a rule for coinductive types which ensures that all terms of coinductive type are *productive*. We here develop a *guardedness* condition specific to our type theory of cyclic proofs. Essentially this condition ensures we encounter an introduction of a constructor which can not be eliminated on all coinductive cyclic paths. The only intermediate terms must reduce finitely through eliminations of finite or inductively defined terms, ensuring that we will not compute indefinitely prior to producing a constructor.

While structural recursion is focused on determining whether the arguments of a recursive term are subterms of some previously destructured term, the dual problem is of determining if a recursive term's context ensures that the term grows. This means we need ways of describing the surrounding context of a term. However, the contexts we have developed thus far are structured in terms of *experiments*. With coinductive terms we need exactly the opposite variety of contexts, those surrounding terms which are *not* experiments.

The key important features of the contexts we are interested in turns out to be whether or not they introduce constructors, and whether they are guaranteed not to remove them. These properties are necessary in the construction of our proof that guardedness leads to *productivity*.

We can describe the relevant features of the context by describing a *path*. This *path* is a series of constructors that allows us to demonstrate which directions to take down a proof tree to arive at a recurrence.

Definition 3 (Path). *A* path *is a finite sequence of pairs of a proof rule from Figure 5 and an index denoting which antecedent it decends from. This pair is described as a* rule-index-pair*.*

An example of such a path would be the following:

$$\text{OrIntroL}^1,\text{AndIntro}^2,\text{ImpIntro}^1$$

This denotes the context:

$$\text{left}((\lambda x : B.-, s), A)$$

With some unknown (and for the purpose of proving productivity, inconsequential) variable x, term s and types A and B.

With this in hand we can establish conditions for guardedness with recursive definitions based on constraints on paths.

Definition 4 (Admissible). *A path is called admissible if the first element c of the path $p = c, p'$ is one of the rule-index-pairs $OrIntroL^1$, $OrIntroR^1$, $AndIntro^1$, $AndIntro^2$, $AllIntro^1$, $\alpha Intro^1$, $ImpIntro^1$, $OrElim^2$, $OrElim^3$, $AndElim^2$, $AllElim^1$, $Delta^1$ and p' is an admissible path.*

Definition 5 (Guardedness). *A path is called guarded if it terminates at a Pointer rule, with the sequent having a coinductive type and the path can be partitioned such that $p = p', [c], p''$ where c is one of the rule-index-pairs $OrIntroL^1$, $OrIntroR^1$, $AndIntro^1$, $AndIntro^2$, $\nu Intro^1$, $ImpIntro^1$ which we will call* guards *and p' and p'' are admissible paths.*

The idea behind the guardedness condition is that we have to be assured that as we take a cyclic path we produce an Intro rule which will never be removed by the reduction relation. The left hand-side of an elimination rule will never cause the elimination of such an introduction and so is *safe*. However, the right hand side of an elimination rule may in fact cause the removal of the introduction rule when we use the evaluation relation.

5 Program Transformation

Supercompilation is a family of program transformation techniques. It essentially consists of *driving*, *information propagation*, *generalisation* and *folding*.

Driving is simply the unfolding and elimination of cuts. Cut-elimination involves the removal of all intermediate datastructures. This includes anything that would be normalised by evaluation in a language like Coq or Agda, including beta-elimination, case elimination or pair selection. Driving, as it removes cuts from infinite proof objects, generates potentially infinite computations.

Information propagation is the use of meta-logical information about eliminations such as case branches. For example, when a meta-variable is destructed in a case branch, the particular de-structuring may be propagated into sub-terms. This is achieved by an inversion on the typing derivation.

Folding is the search for recurrences in the driven program. A recurrence will be an expression which is a variable renaming of a former expression. Essentially, if a recurrence is found we can create a new recursive function having the same body as the driven expression with a recursive call at the recurrence.

Generalisation may be used in order to find opportunities for folding. We can abstract away arguments which would cause further driving and would not allow us to fold.

Our notion of equivalence of proofs must be quite strict if it is to preserve the operational behaviour of the program. The notion of equivalence we use here is *contextual equivalence*.

Definition 6 (Contextual Equivalence). *For all terms s, t and types A and type derivations $\cdot \vdash s : A$ and $\cdot \vdash t : A$, and given any experiment e such that $x : A \vdash e : B$ then we have that $e[x := s] \Downarrow$ and $e[x := t] \Downarrow$ then $s \sqsubseteq t$. If $s \sqsubseteq t$ and $t \sqsubseteq s$ then s is contextually equivalent to t or $s \cong t$.*

In the examples, the relation between the original and transformed proofs is simply either a compatible relation with the formation rules, or the term is related up to beta-equivalence. In the case of unfolding, it's clear that no real change has taken place since the unfolded pre-proof just extends the prefix of the potentially infinite pre-proof, and is identical by definition. The finite prefix is merely a short hand for the infinite unfolding of the pre-proof.

Dealing with reduction under the evaluation relation is more subtle. In order to establish equivalence here we need to show that if the term reduces, it reduces to an outer-most term which will itself reduce when an experiment is applied. This is essentially a *head normal form*, that is, a term whose outermost step will not reduce in any context. This idea is essential to defining productivity, since it is precisely the fact that we have *done something* irrevocable which gives us productivity.

Folding is also somewhat complex as, in our case, we will require the use of generalisation, which is essentially running the evaluation relation backwards in order to find terms which will be equivalent under reduction, and cycles in the proof which can lead to potential unsoundess. The key insight of this paper is that in fact, unsoundess can not be introduced if the cycles themselves are productive or inductive for coinduction and induction respectively.

Generally the program transformation technique itself is controlled by using some additional termination method such as a depth bound or more popularly the homeomorphic embedding. This however does not influence the correctness of the outcome. If an algorithm in the supercompilation family terminates, the final program is a faithful bisimulation of the original.

Since all of the examples given in this paper clearly follow the fold/unfold generalise paradigm, and all examples are inductive/productive, the correctness can be assumed. In a future work we hope to present the algorithm that was used to find these examples in more detail, and to show that it will in general produce contextually equivalent programs. We will see how these elements are applied in practice by using these techniques to work with cyclic proofs.

5.1 Reduction

Previously we gave a bird's-eye view of supercompilation as being a family of program transformations composed of driving, generalisation and folding. Cyclic proofs give us the tools necessary to justify folding in the context of types and driving is simply the unfolding of a cyclic pre-proof.

In order to perform folding however, we need to be able to arrive at nodes which are α-renamings of former nodes. In order to do this in general we need to be able to generalise terms. We can, using generalisation, regenerate proofs which simply make reference to recursive functions, by generalising to reproduce α-renamings of the function bodies and folding. This ensures that we can produce *at least* the proofs possible already using the original term.

For higher order functional languages, there are a potentially infinite number of generalisations of two terms, and the least general generalisation may itself consist of many incomparable terms [14]. For this reason, some heuristic

approach needs to be applied in order to find appropriate generalisations. We will not be concerned about the particular heuristic approach used to determine generalisations as this is quite a complex subject, but only that it meet the condition that the generalisation can be represented as an elimination rule in the proof tree and that will regenerate the original proof tree under evaluation.

6 Example Revisited

With the notion of cyclic proof, we now have at our disposal the tools necessary to transform pre-proofs into proofs. We will revisit the *sumlen* example given as motivation for the present work and see how we can represent the transformations.

We take again an example using the *co-natural numbers* $\mathbb{N}^\infty \equiv \nu X.1 + X$ and potentially infinite lists $[A] \equiv \Lambda A.\nu X.1 + (A \times X)$. Here we take Ω to be defined as:

```
Ω(zero) := in(left( (),N̄))
Ω(plus) := λ x y : N̄ .
  case (out(x,N̄)) of
    | z ⇒ ys
    | n ⇒
     fold(right(plus n y),N̄)
Ω(sumlen) := λ xs : [N̄] .
  case (out(xs,[N̄])) of
    | nil ⇒ zero
    | p ⇒
     split p as (n,xs′)
      in in(right(plus n (sumlen xs′)),N̄)
```

We can now produce the type derivation by performing the successive steps given explicitly in Figure . Here in the final step we have driven the proof tree to the point that we can now reference two previous nodes. One of those is labelled with a †, the other with a ∗. This final pre-proof is now a proof because it satisfies the guardedness condition. We have taken the liberty of introducing a an additional derived rule $\text{Cons}^{[\![\overline{\mathbb{N}}]\!]}$. It is merely a shorthand for the use of I_R^+ and I^ν, together with a proof that these types admissible under the formation rules.

We can then produce a *residual* program from the cyclic proof. This is simply a mutually recursive (or letrec) function definition which makes any cycle into a recursive call. The residual term will be essentially the one given in agda above.

7 Related Work

The present work uses a program transformation in the supercompilation family. This was first described by Turchin [17] and later popularised by Sørensen, Glück

$$\frac{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \text{case } xs \text{ of } \{[]\Rightarrow \mathit{czero}\mid x::xs'\Rightarrow \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\}\ :\ \overline{\mathbb{N}}}{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \texttt{sumlen}\ xs\ :\ \overline{\mathbb{N}}}\ I^{\Omega}$$

$$\Downarrow$$

$$\frac{\cdot\,;xs:[\![\overline{\mathbb{N}}]\!]\vdash xs\ :\ [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots\vdash \mathit{czero}\ :\ \overline{\mathbb{N}} \qquad \cdot\,;\{x\colon\overline{\mathbb{N}},xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\ :\ \overline{\mathbb{N}}}{\dfrac{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \text{case } xs \text{ of } \{[]\Rightarrow \mathit{czero}\mid x::xs'\Rightarrow \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\}\ :\ \overline{\mathbb{N}}}{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \texttt{sumlen}\ xs\ :\ \overline{\mathbb{N}}}\ I^{\Omega}}\ E^{+}$$

$$\Downarrow$$

$$\frac{\cdot\,;xs:[\![\overline{\mathbb{N}}]\!]\vdash xs\ :\ [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots\vdash \mathit{czero}\ :\ \overline{\mathbb{N}} \qquad \dfrac{\cdot\,;\{x\colon\overline{\mathbb{N}},xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash x+(\texttt{sumlen}\ xs')\ :\ \overline{\mathbb{N}}}{\cdot\,;\{x\colon\overline{\mathbb{N}},xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\ :\ \overline{\mathbb{N}}}\ \mathrm{Cons}^{[\![\overline{\mathbb{N}}]\!]}}{\dfrac{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \text{case } xs \text{ of } \{[]\Rightarrow \mathit{czero}\mid x::xs'\Rightarrow \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\}\ :\ \overline{\mathbb{N}}}{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \texttt{sumlen}\ xs\ :\ \overline{\mathbb{N}}}\ I^{\Omega}}\ E^{+}}\ I^{\Omega}$$

$$\Downarrow$$

$$\frac{\cdot\,;xs:[\![\overline{\mathbb{N}}]\!]\vdash xs\ :\ [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots\vdash \mathit{czero}\ :\ \overline{\mathbb{N}} \qquad \dfrac{\dfrac{\cdot\,;\{x\colon\overline{\mathbb{N}},xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash x+(\texttt{sumlen}\ xs')\ :\ \overline{\mathbb{N}}}{\cdot\,;\{x\colon\overline{\mathbb{N}},xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\ :\ \overline{\mathbb{N}}}\ \mathrm{Cons}^{[\![\overline{\mathbb{N}}]\!]}}{\dfrac{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \text{case } xs \text{ of } \{[]\Rightarrow \mathit{czero}\mid x::xs'\Rightarrow \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\}\ :\ \overline{\mathbb{N}}}{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \texttt{sumlen}\ xs\ :\ \overline{\mathbb{N}}}\ I^{\Omega}}\ E^{+}}{\cdot\,;\{x\colon\overline{\mathbb{N}},xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \text{case } x \text{ of } \{\mathit{czero}\Rightarrow \texttt{sumlen}\ xs'\mid \mathit{csucc}\ x\Rightarrow \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\}\ :\ \overline{\mathbb{N}}}\ I^{\Omega}$$

$$\Downarrow$$

$$\mathcal{D} := \frac{\dfrac{\cdot\,;\{xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash xs'\ :\ [\![\overline{\mathbb{N}}]\!] \qquad \overline{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \texttt{sumlen}\ xs\ :\ \overline{\mathbb{N}}}\ {\scriptstyle\uparrow}}{\cdot\,;\{xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \texttt{sumlen}\ xs\ \langle xs:=xs'\rangle\ :\ \overline{\mathbb{N}}}\ I^{\theta} \qquad \texttt{sumlen}\ xs\ \langle xs:=xs'\rangle \rightsquigarrow^{*} \texttt{sumlen}\ xs'}{\cdot\,;\{xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \texttt{sumlen}\ xs'\ :\ \overline{\mathbb{N}}}\ \mathrm{Conv}$$

$$\mathcal{E} := \frac{\dfrac{\cdot\,;\{xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash xs'\ :\ \overline{\mathbb{N}} \qquad \overline{\cdot\,;\{x\colon\overline{\mathbb{N}},xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \mathit{csucc}\ (x+(\texttt{sumlen}\ xs))\ :\ \overline{\mathbb{N}}}\ {\scriptstyle *}}{\cdot\,;\{x\colon\overline{\mathbb{N}},xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash (\mathit{csucc}\ (x+(\texttt{sumlen}\ xs)))\ \langle xs:=xs'\rangle\ :\ \overline{\mathbb{N}}}\ I^{\theta} \qquad (\mathit{csucc}\ (x+(\texttt{sumlen}\ xs)))\ \langle xs:=xs'\rangle \rightsquigarrow^{*} \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))}{\cdot\,;\{x\colon\overline{\mathbb{N}},xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\ :\ \overline{\mathbb{N}}}\ \mathrm{Conv}$$

$$\frac{\cdot\,;xs:[\![\overline{\mathbb{N}}]\!]\vdash xs\ :\ [\![\overline{\mathbb{N}}]\!] \qquad \cdot\,;\ldots\vdash \mathit{czero}\ :\ \overline{\mathbb{N}} \qquad \dfrac{\dfrac{\cdot\,;\{x\colon\overline{\mathbb{N}}\}\vdash x\ :\ \overline{\mathbb{N}} \qquad \mathcal{D} \qquad \mathcal{E}}{\cdot\,;\{x\colon\overline{\mathbb{N}},xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \begin{array}{l}\text{case } x \text{ of}\\ \quad \{\ \mathit{czero}\Rightarrow \texttt{sumlen}\ xs'\\ \quad\ \mid \mathit{csucc}\ x\Rightarrow \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\end{array}\ :\ \overline{\mathbb{N}}}\ E^{+}}{\cdot\,;\{x\colon\overline{\mathbb{N}},xs'\colon[\![\overline{\mathbb{N}}]\!]\}\vdash x+(\texttt{sumlen}\ xs')\ :\ \overline{\mathbb{N}}}\ I^{\Omega}}{\dfrac{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \begin{array}{l}\text{case } xs \text{ of}\\ \quad \{\ []\Rightarrow \mathit{czero}\\ \quad\ \mid x::xs'\Rightarrow \mathit{csucc}\ (x+(\texttt{sumlen}\ xs'))\end{array}\ :\ \overline{\mathbb{N}}}{\cdot\,;\{xs\colon[\![\overline{\mathbb{N}}]\!]\}\vdash \texttt{sumlen}\ xs\ :\ \overline{\mathbb{N}}}\ I^{\Omega}\ \dagger}\ \mathrm{Cons}^{[\![\overline{\mathbb{N}}]\!]}\ {\scriptstyle *}}\ E^{+}$$

Fig. 7. Sumlen Cyclic Proof

and Jones [16]. We essentially use the same algorithms with the addition of the use of type information to guide folding.

The use of cylic proofs was developed by Brotherston [4]. We extend this work by dealing also with coinductive types and make use of it in a Curry-Howard settings.

The correspondence between cyclic proof and functional programs has previously been described by Robin Cockett [5]. His work also makes a distinction between inductive and coinductive types. Our work differs in using super compilation as a means of proving type inhabitation.

Various approaches to proving type inhabitation for coinductive types have appeared in the literature. Bertot and Komendanskaya give a method in [3]. A method is also given using sized types is given by Abel in [1]. The approach in this paper differs in that we use transformation of the program rather than reasoning about the side conditions.

8 Conclusion and Future Work

The use of program transformation techniques for proofs of type inhabitation is attractive for a number of reasons. It gives us the ability to mix programs which may or may not be type correct to arrive at programs which are provably terminating. We can keep an audit trail of the reasoning by which the programs were transformed. And we can admit a larger number of programs by transformation to a form which is syntactically correct, obviating the need for complex arguments about termination behaviour. For these reasons we feel that this work could be of value to theorem provers in the future.

To the best of the authors knowledge, no examples of a supercompilation algorithm have yet been given for a dependently typed language. The authors hope to extend the theory to dependent types in the future such that the algorithm might be of assistance to theorem provers.

Currently work is being done on a complete automated proof of correctness of a supercompilation algorithm for the term language described in this paper in the proof assistant Coq. The cyclic proofs are represented using a coinductive data-type, rather than the usual inductive description.

The technique as presented works well for many examples, however there are some examples in which direct supercompilation is insufficient. The following program tries to capture the notion of a semi-decidable existential functional which takes a semi-decidable predicate over the type A. The usual way to write this in a functional language is to use the Sierpinski type [8], the type of one constructor. Here truth is represented with termination, and non-termination gives rise to falsehood.

```
data S : Set where
  T : S
```

However since languages such as Coq and Agda will not allow us to directly represent non-termination, we will embed the Sierpinski type in the delay monad.

```
codata Delay (A : Set) : Set where
  now : A → Delay A
  later : Delay A → Delay A
```

The clever reader might notice that this is in fact isomorphic to the conatural numbers and that join is simply the minimum of two potentially infinite numbers.

```
join : Delay S → Delay S → Delay S
join (now T) x = now T
join x (now T) = now T
join (later x) (later y) = later (join x y)
ex : {A : Set} → (A → Delay S) →
  Stream A → Delay S
ex p (x::xs) = join (p x) (ex p xs)
```

By unfolding *join* and *ex* we eventually arrive at a term:

```
-- join x' (join (p x) (ex p xs))
```

This term is a repetition of the original body of *ex*, with $p\ x$ abstracted, provided that join is associative. Unfortunately, using direct supercompilation, we are unable to derive a type correct term automatically. However, using ideas presented by Klyutchnikov and Romanenko [12], the technique might be extended in such a way to provide an automated solution for this example as well. Using the fact that the recurrence is contextually equivalent, we can fold the proof to obtain the following program, which is productive, and admissible into Agda.

```
mutual
  ex_trans : {A : Set} →
    (A → Delay S) → Stream A →
    Delay S
  ex_trans p (x::xs) = later (j (p x) p xs)
  j : {A : Set} →
    Delay S → (A → Delay S) →
    Stream A → Delay S
  j (now T) p _ = now T
  j (later n) p (x::xs) = later (j (join n (p x)) p xs)
```

References

1. Andreas Abel. Termination checking with types. Technical report, Institut fr Informatik, Ludwigs-Maximilians-Universitt Mnchen, 2002.
2. Andreas Abel. Typed Applicative Structures and Normalization by Evaluation for System F^{ω}. In Erich Grdel and Reinhard Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2009.

3. Yves Bertot and Ekaterina Komendantskaya. Inductive and Coinductive Components of Corecursive Functions in Coq. *Electron. Notes Theor. Comput. Sci.*, 203(5):25–47, 2008.
4. James Brotherston. Cyclic Proofs for First-Order Logic with Inductive Definitions. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: Proceedings of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.
5. J. Robin B. Cockett. Deforestation, program transformation, and cut-elimination. *Electr. Notes Theor. Comput. Sci.*, 44(1), 2001.
6. Thierry Coquand. Infinite objects in type theory. In *TYPES 93: Proceedings of the international workshop on Types for proofs and programs*, pages 62–78, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.
7. Nils A. Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 41, pages 206–217, New York, NY, USA, January 2006. ACM.
8. M. H. Escardo. Synthetic topology of data types and classical spaces. *ENTCS, Elsevier*, 87:21–156, 2004.
9. Eduardo Gimnez. Structural Recursive Definitions in Type Theory. In *ICALP 98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 397–408, London, UK, 1998. Springer-Verlag.
10. Andrew D. Gordon. Bisimilarity as a theory of functional programming. *Theor. Comput. Sci.*, 228(1-2):5–47, 1999.
11. Neil D. Jones. Program termination analysis by size-change graphs (abstract). In *IJCAR*, pages 1–4, 2001.
12. Ilya Klyuchnikov and Sergei Romanenko. Proving the Equivalence of Higher-Order Terms by Means of Supercompilation. In *Perspectives of Systems Informatics*, volume 5947, pages 193–205. Springer, 2009.
13. Simon L. Peyton Jones and David R. Lester. *Implementing functional languages.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
14. Frank Pfenning. Unification and anti-unification in the calculus of constructions . In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science (LICS 1991)*, pages 74–85. IEEE Computer Society Press, July 1991.
15. Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002.
16. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. A Positive Supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.
17. Valentin F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

Reversivity, Reversibility and Retractability

Nikolai N. Nepejvoda

Program Systems Institute of RAS, Pereslavl-Zalessky, Yaroslavl Region, Russia, 152020
nepejvodann@gmail.com,
WWW home page: http://site.u.pereslavl.ru/Personal/NikolaiNNepeivoda/

Abstract. Three essentially different but usually mixed notions of program invertibility are considered. Reversivity when each action has a full inverse. Reversibility when each action can be undone (right inverse). Retractability when erroneous result can be retracted down to error in data. Constructive logical, algebraic, programming and realization aspects are considered for those types of programs.

Keywords: constructive logics, reversible computing, reversive logic, program inversion

1 Origins of reversivity

R. Landauer [1961] pointed out that there is an important kind of computations not investigated earlier and not intensively studied now (though not forgotten). Consider, for example, a superconductor based computer. Its elements are to be cooled by liquid hydrogen to provide superconductor properties of elements. Hot pollution could be fatal for this kind of processor. Thus we need all actions of our computer not to induce heat pollution. Landauer showed that it is physically possible if all actions are *invertible*.

The following formula describes the state of arts here:

Generalized Landauer – von Neumann principle

$$E_{diss} \geqslant T \times k_B \times \ln P \tag{1}$$

Here T is the temperature in K°, k_B is the Boltzmann's constant, P is the number of states of atomic computing element (assuming that energy needed to transfer between any two states is equal). A fresh work on this topic which contains an attempt of experimental verification of Landauer's principle is [2012L].

Bennett [1973] proposed in 1973 to make logically reversible computer (in the sense of boolean logic). He at first time made a big error which was inherited by further works.

In the common sense invertible action is that which can be wholly undone. For example adding block of text to Word file we always can delete it. This is semi-invertible action: we can undo it after it is done but cannot prevent it by doing an inverse action before it. For example accepting and rejecting changes in

Word document is a semi-invertible action (inside one Word session). Accepting and rejecting them in 'track changes' mode is a fully invertible one. This shows a deep difference between reversive (invertible) and reversible (semi-invertible) computing. The second one cannot diminish heat pollution and cannot be applied if our basic computing units are mostly invertible (e.g. quantum or DNA).

Toffoli [1980)] and Fredkin [1982] showed that it is possible to perform the full set of boolean computations on superconductor based computations almost without hot generation by the cost of doubled memory. A superconductor elements based cell can change state without hot generation if numbers of zeros and ones do not change. Thus Toffoli proposed to consider computations where each cell contains the same number of zeros and ones forever but they danced inside during computation process. Inevitable hot generators are only setting the initial state of a computer and reading the final result from some parts of superconductor cells.

Merkle [1992] proposed another variant of reversible binary logic devices. His goal is to beat physical barrier for computer productivity. His description of a problem is brilliant [1996]:
"If the exponential trends of recent decades continue, energy dissipation per logic operation will reach kT (for $T = 300$ Kelvins) early in the next century. Either energy dissipation per logic operation will be reduced significantly below $3 \cdot 10^{-21}$ joules, or we will fail to achieve computers that simultaneously combine high packing densities with gigahertz or higher speeds of operation. There are only two ways that energy dissipation can be reduced below $3 \cdot 10^{-21}$ joules: by operating at temperatures below room temperature (thus reducing kT), or by using thermodynamically reversible logic. Low temperature operation doesn't actually reduce total energy dissipation, it just shifts it from computation to refrigeration. Thermodynamically reversible logic elements, in contrast, can reduce total energy dissipation per logic operation to $\ll kT$."

The main error here is that almost all authors forget that to get a new memory location is also an energy-dissipating action. Moreover those who in concrete considerations do not allow this action (e.g. Merkle, Toffoli) do not mention this restriction in their comments.

There is a spread (rather little but slowly growing until this day) of works considering "reversible logic" and reversible computing. Too narrow point of view of *all* these works is to consider reversible computing only as boolean reversible transformations. Thus brilliant but restricted ideas of Merkle, Toffoli and Fredkin are accepted as "axiomatic" which cannot be discussed but only developed (as a good example see [2003]). We emphasize that boolean elements are only a *tradition* of current hardware but not its *sine qua non* property. Thus those aspects of reversivity which will become crucial if reversive super processors would be made technically: "How to program those exotic units?" and "What we cannot put into them without destroying their potentially best sides?" are not touched and even not seen.

And we are to emphasize the following:

REVERSIVITY
To overcome the Landauer principle we need full invertibility, not only possibility of undoing: each action f must have an action f^{-1} s.t. $f \circ f^{-1} = f^{-1} \circ f = e$ where e is an empty action.

This was assumed in the original works and repeatedly pointed out by independent researchers (see e.g. [2001Bub,2005Mar,2012L])

Thus *we need more general mathematical and more logical consideration of this extremely interesting and specific domain.* Constructive logic of reversivity is the first step into a new realm.

Now we consider three notions of (semi) invertibility in the order from an almost traditional up to the most striking, leading to a fresh paradigm and therefore totally inadequately treated.

2 Retractability

Main peculiarity of constructive logics is that they are not truth-value based. Formulas are treated as problems and we are interested in solution of this problem which can be for different logics and theories whether a mathematical object or a program or another entity. If a is a solution of a problem represented by a logical formula A we say 'a realizes A' (a ® A).

I don't know a systematic survey of all branches of constructive mathematic written in English. Constructivism now is a system where different main resources and different resource restrictions lead to very exotic, mutually inconsistent and fine systems (see [2011NNN]). The logic of knowledge is the intuitionistic logic; the logic of money is the linear logic of Girard; the logic of time, automata and real actions is the nilpotent logic; the logic of reversivity and soul is partially described below. One of main philosophical consequences of constructivism is that it is a mortal trick of a society to allow those people which are thinking inside the logic of money to govern the real things or the knowledge discovering process.

Some formulas in constructive logics can have a trivial underlying problem and thus be treated as descriptive ones (usually as classical). For example formulas $\neg A$ in intuitionistic logic are descriptive and classical logic can be isomorphically embedded into intuitionistic (but not vice versa).

In this section we use almost forgotten and not developed further results of constructive logic applications to programming. "*Sturm und Drang*" of this topic was in last 70-ths — early 80-ths. There were many fine algorithms and strong results (see e. g. [NNN1982,Mar1982,1991,1998]) but their practical applications meet some obstacles.

Obstacle 1. Swamp. As usually, after big advance there arose a huge boring work to develop techniques and technologies of practical work with: new possibilities opened (it is easier but often demands a full change of traditional manner of actions); with new dangers and shortcomings which accompany each innovation. Big benefits always go together with big disadvantages. To lead new

footpath through a swamp is hard and boring work. We cannot make here big promises (if we didn't lost remains of honor). We cannot show any spectacular results here. We cannot easily explain significance of our work to outsiders (= peer reviewers). Thus we cannot get grants and our work dies before reaching the another bank of a swamp and a new living space. Constructive logics had faced with the necessity to develop essentially another technique of formalization because traditional one did not works good. This work had become their swamp.

Obstacle 2. Disorientation. The theoretical works are mostly disorienting here because they usually treated so "practical and important" examples as factorial or Ackermann's function. These examples are both too primitive in their structure and too connected with the specific data types and the traditional set of atomic operations but easily understood by theoristsoutsiders. They were out of mainstream of both theoretical and practical informatics which now works with abstract and varying structures and constructs new programs not from the computer primitives but from modules, objects and patterns.

Obstacle 3. Fake advertising. Japan 'Fifth generation project' used constructive logic as one of its banners. In reality they have adopted more traditional but leading into deadlock tools (e.g. Prolog). A reputation of a fallen project was transferred into its 'used' theory.

A program is *retractable* if it allows to retract from properties and/or errors in the result to properties or errors in the data. The logic of retractable structured program is symmetric intuitionistic logic (SIL) investigated by I. Zaslavsky [1979].

In this logic there are only constructive connectives $\Rightarrow \vee \& \sim \forall \exists$. Their semantic is defined through the two notions of realizability: positive and negative one.

Definition 1. *Realizabilities for SIL.*

1. $\langle a, b\rangle \; ®^+ \; A \,\&\, B \equiv a \; ®^+ \; A \wedge b \; ®^+ \; B$;
   $\langle i, c\rangle \; ®^- \; A \,\&\, B \equiv (i = 1 \wedge c \; ®^- \; A)$ *or* $(i = 2 \wedge c \; ®^- \; B)$;
2. $\langle i, c\rangle \; ®^+ \; A \vee B \equiv (i = 1 \wedge c \; ®^+ \; A)$ *or* $(i = 2 \wedge c \; ®^+ \; B)$;
   $\langle a, b\rangle \; ®^- \; A \vee B \equiv a \; ®^- \; A \wedge b \; ®^- \; B$;
3. $\langle f, g\rangle \; ®^+ \; A \Rightarrow B \equiv$ *for all* $a \; (a \; ®^+ \; A \supset !(a \; f) \wedge (a \; f) \; ®^+ B) \wedge$
   for all $b \; (b \; ®^- \; B \supset !(b \; g) \wedge (b \; g) \; ®^- \; A)$;
   $\langle a, b\rangle \; ®^- \; A \Rightarrow B \equiv a®^+ \; A \wedge b®^- \; B$;
4. $a \; ®^+ \sim A \equiv a \; ®^- \; A$;
   $a \; ®^- \sim A \equiv a®^+ \; A$;
5. $f \; ®^+ \; \forall x \, A(x) \equiv$ *for all* $a \; (a \in U \supset !(a \; f) \wedge (a \; f) \; ®^+ \; A(a))$;
   $\langle u, a\rangle \; ®^- \; \forall x \, A(x) \equiv$ *exists* $u \; (u \in U \wedge a®^- \; A(u))$;
6. $\langle u, a\rangle \; ®^+ \; \exists x \, A(x) \equiv$ *exists* $u \; (u \in U \wedge a®^+ \; A(u))$;
   $f \; ®^- \; \exists x \, A(x) \equiv$ *for all* $a \; (a \in U \supset !(a \; f) \wedge (a \; f) \; ®^- \; A(a))$;

Here $!t$ denotes «value of t exists»; U is the set of all primitive objects of our model.

The usual rules for negation are valid for SIL. There is an extraction algorithm which can extract two procedures from a proof of formula of the form:

$$\forall x_1 \ldots x_n \, (A_1 \& \ldots \& A_m \Rightarrow \exists y_1 \ldots y_k \, (B_1 \& \ldots B_l))$$

The first procedure finds y for all x satisfying A. The second one shows how to find such j that $\sim A_j(x_0)$ having $\sim B_i(x_0, y_0)$. Thus we have both a program and a routine to analyze its errors.

Now we consider an example. Let in a subtheory (essentially constructive formulas are specified by their realizations)

$$\begin{array}{l}\forall x\,((A(x) \Rightarrow N(x)), \qquad \varphi \circledR \forall y\,(N(y) \Rightarrow \sim \exists x\,M(x)),\\ g \circledR \forall x(C(x) \Rightarrow L(x) \vee E(x) \vee M(x)),\\ \forall x\,(L(x) \Rightarrow D(x)), \qquad \forall x\,(H(x) \Rightarrow T(x, (x\ f)))\end{array}$$

which is a part of a constructive theory describing some packages of programs we proved a formula

$$\forall x\,(A(x)\ \&\ (\forall x\,(C(x) \Rightarrow D(x) \vee E(x)) \Rightarrow \exists y\,H(y)) \Rightarrow \exists y\,H(y)) \Rightarrow \exists z\,T(y, z))$$

by the following proof:

$$\begin{array}{l}* \ A(z),\ \forall x\,(C(x) \Rightarrow D(x) \vee E(x)) \Rightarrow \exists y\,H(y),\ z \text{ is arbitrary}\\ \quad N(z)\\ \quad \sim \exists x\,M(x)\\ \quad * \ C(u),\ u \text{ is arbitrary}\\ \quad\quad L(u) \vee E(u) \vee M(u)\\ \quad\quad \sim M(u)\\ \quad\quad * \ L(u) \qquad * \ E(u)\\ \quad\quad \mid D(u)\\ \quad \forall x\,(C(x) \Rightarrow D(x) \vee E(x))\\ \quad H(c_1)\\ \quad T(z, (c_1\ f))\end{array}$$

$$\begin{array}{l}* \ \sim T(y, z),\ y, z \text{ are arbitrary}\\ \quad \sim A(x) \vee \sim (\forall x\,(C(x) \Rightarrow D(x) \vee E(x)) \Rightarrow \exists y\,H(y))\\ \quad * \ \sim (\forall x\,(C(x) \Rightarrow D(x) \vee E(x)) \Rightarrow \exists y\,H(y))\\ \quad\quad \sim H(x),\ x \text{ is arbitrary}\\ \quad\quad \exists x\,(C(x)\ \&\sim D(x)\ \&\sim E(x))\\ \quad\quad L(c_2) \vee E(c_2) \vee M(c_2)\\ \quad\quad \sim L(c_2) \qquad \sim E(c_2) \qquad * \ \sim A(y)\\ \quad\quad M(c_2)\\ \quad\quad \sim N(y)\\ \quad\quad \sim A(y)\\ \quad \sim A(y)\end{array}$$

Here our direct program is

$$\begin{array}{l}\Phi: \ \mathbf{func}\ (\mathbf{obj}, \mathbf{func}(\mathbf{func}(\mathbf{obj})\mathbf{void} \oplus \mathbf{void})\ \mathbf{obj})\ \mathbf{obj} =\\ \lambda x, \Psi.\,((\lambda x.\,\mathbf{case}\ (x\ g)\ \mathbf{in}\ 1:1, 2:2, 3:\ \mathbf{error\ esac}\ \ \Psi)\ f)\end{array}$$

If its result is wrong, an error is in A. The reason of this trouble is probably a wrong value of x which formally does not enter into a resulting program.

A procedure of backward analysis given originally for error diagnosis can be used for other kinds of backward computations of program condition. We have to stress that during this kind of backward computations we are interested not in restoring of values but in information about initial values which had been lost or not taken into account before program computation.

Retraction is first order process even for functional programs in the majority of practical situations (roughly speaking if our positive suppositions do not include a demand to grant an erroneous result of a procedure).

Moreover here we have an interesting duality. G. S. Tseytin pointed out in 1970 that program values are not sufficient to analyze a program. Program is surrounded by *ghosts* which are necessary to understand and to transform a program but are at least useless during its computation. During retraction ghosts become computable entities while values of direct program become ghosts.

3 Reversibility

Invertibility cannot be considered as a property of some exotic classes of hardware computations. In business and legal practice it is sometimes necessary to provide a possibility to restore easily the precise state of the whole system for *each* given moment in the past. In many interactive program systems it is necessary to provide unrestricted undoing.

Program allowing unrestricted undoing is called *reversible*.

Definition 2. *Let X be an enumerated set. Let $\mathfrak{C}(X, X)$ be a set of all total computable functions $f : X \to X$. A semigroup $R \subset \mathfrak{C}(X, X)$ having a neutral element $e = \lambda x.x$ and having a right inverse f^{-1} for each f (i.e. such f^{-1} that $f \circ f^{-1} = e$) is called* reversible computability *upon set of objects X.*

We emphasize once more that reversibility has no relation to problem of heat generation and programming physically reversive units. Nevertheless it is practically essential and interesting.

Main results on ≪possibility to make any Turing computable function invertible≫ consider only reversibility. In publications two notions ≪reversivity≫ and ≪reversibility≫ are systematically muddled together.

A reversible processor can in principle work autonomously. But it is necessary to remember lower bounds of extra resources needed for reversibility as stated in [1973].

$$\text{Time} > 3^k \cdot 2^{O\left(\frac{T}{2^k}\right)} \qquad \text{Store} > S \cdot (1 + O(k)) \tag{2}$$

where k can be chosen between 1 and $\log_2 T$. Here we have somewhat shifted notions. Bennett result and its further generalizations consider a problem how to simulate an irreversible computation on reversible processor. In practice we are interested in the same result but not in the same computation flow. Thus this bound is theoretically correct but practically somewhat misleading.

Moreover we don't need to assure undoing down to atomic actions in reversible computing because reversibility is needed only for external reasons (say many legal and business program must be able to reconstruct the state of the system for any previous time moment). Hence *a reversible program can use modules written in irreversible manner if we grant undoing of their results.*

From this point we can see strategic mistakes made in the design of reversible language Janus [2007]. For example, there is a brilliant invention of Janus authors that each unary function f is extended up to its reversible extension

$$(x\ y\ g) = \langle x * (y\ f), y\rangle$$

where $\forall x, y, z\,(x * z = y * z \supset x = y)$. They showed that each unary function can be extended in such manner. But this excellent shot had a wrong goal and is missed. Of course it is too much for reversibility but too less for reversivity (it grants only undoing).

Now we will outline some reasons why there is no need of a reversible programming languages. Reversibility can be reached by a discipline of programming in a traditional structured language.

It is necessary to prohibit completely any invisible side effects of functions and procedures.

Each procedure f can have only **in** and **inout** parameters and have a dual procedure undo_f which makes undoing of its effects.

Each function can have only **in** parameters and needs no undoing procedure.

Let us remember the remark of E. Dijkstra and D. Gries [2010Dij,1981Gr] that natural assignment statements are to have a form $x_1, \ldots, x_n \leftarrow t_1, \ldots, t_n$.

Let now classify all variables in all program points as virgin, used and monk ones. Now each assignment must have one of two forms. A multi form $x_1, \ldots, x_n \leftarrow y_1, \ldots, y_n$ where the right side contains no virgin variables and each used variable from the left side must occur at least once in the right side. A single form $x \leftarrow t$ where x is virgin. After this x becomes used.

Case statements and loops are unified in a spider statement form proposed in [NNN1982] (this is an extension of Dijkstra's loops and guarded commands and had used e.g. by Bel'tyukov in his language КЬІЛ):

for i,x1,. . . ,xn **do** N1: S1,. . . , Nk: Sk **out** L1: T1,. . . , Lm: Tm **new** U **od**.

Here i is an integer variable, Nj, Lj are integers, $i, x1, \ldots, xn$ are frozen inside each loop step and can be changed only in **new** part. Inside loop they are called *monks*.

A spider loop is executed as follows. If the value of i does not equal to any of Nj, Lj there is an error. If it is equal to some Nj, the corresponding statement Sj is executed, then U is executed and the loop continues. If it is equal to some Lj, the loop is finished after execution of Tj and U.

Those conditions (easy to formulate as a discipline and technology of programming and easy to check) are sufficient to grant reversibility.

Thus conditionals and loops do not hinder reversibility and only force us sometimes to introduce some additional information.

Constructive logic of reversibility is a good problem for a new research.

4 Reversivity

Constructive reversive logic (CRL) was described and investigated in [2009]

For a mathematical semantic we consider an arbitrary group G. One more important step was proposed and successfully developed by J.-Y. Girard in his linear logic (using commutative monoid to represent money-spending actions). For our case it sounds as follows:

States are the same group as actions.

Thus G is called both *the group of actions* and *the group of states*.

Each propositional letter corresponds to a subset of the group[1], and each element α of the group represents the function $\lambda x.\, x \circ \alpha$.

Thus in the functional language based on groups application of f to a is to be written (a f) in contrary to usual (f a). Composition of group elements $a \circ b$ can be understood by any of three ways:

1. We perform the state-transfoming action a then the action b;
2. We apply the function b to a;
3. We construct a composition of functions a and b.

All those interpretations are compatible and fully interoperable. This is the main peculiarity of group as a space of elements and actions.

CRL is a propositional logic. The primitives of reversive logic language are propositional symbols A, B, $C\ldots$, five connectives of classical logic ($\supset$, $\equiv$, $\wedge$, $\vee$, $\neg$) called here *descriptive connectives*, four constructive logical connectives $\Rightarrow, \&, \sim, E$. E is null-ary, $\neg$ and $\sim$ are unary, all others are binary. We adopt the following priority of binary connectives (form the weakest one up to strongest): $\Rightarrow$ & $\equiv\supset$ $\wedge\vee$ but we use parenthesis rather than priorities of $\equiv\supset$ and $\wedge\vee$ for the needs of easy reading. Let *signature* Σ be a nonempty set of propositional symbols.

Classical connectives are read and understood in standard way. $\Rightarrow$ reads "can be transformed", $A\&B$ reads "*sequential conjunction*" or "A then B"[2], $\sim A$ is a preventive negation which can be read in different contexts as "undo A" or "prevent A".

Classical and constructive connectives are fully interoperable[3] and can be mixed arbitrarily. This is not the case in other constructive logics of restricted constructions.

A formula A is *descriptive* if there are no constructive connectives. A formula A is *pure constructive* if there are no classical connectives. Therefore propositional letters are both classical and pure constructive formulas.

[1] Attention! This subset is not obliged to be a subgroup or stable in the sense of the linear logic. This is a principal distinction from quantum logics and other algebraic logics.

[2] Of course we can read this "and" in the sense of famous Kleene's examples: "Mary married and born a child", "Mary born a child and married".

[3] As it is called in modern programming.

Our main semantic notion is "element *a realizes* a formula A in an interpretation I" ($I \models a\circledR A$). If an interpretation is fixed we omit I.

$\triangleq$ means «is equivalent by definition»

Definition 3. Interpretation *of a signature Σ is a pair consisting of a group G and of a function $\zeta : \Sigma \to \mathbb{P}G$ which maps propositional letters into power set of G. A subset which is assigned to a propositional symbol A in I is denoted $\zeta_I(A)$. If I is fixed we omit the index.*

Definition 4. Realization *of a formula in the interpretation I.*

1. $a \circledR A \triangleq a \in \zeta(A)$ *where A is propositional letter and $A \in \Sigma$.*
2. *For all classical connectives their definitions are standard. E.g.* $a \circledR (A \wedge B) \triangleq a\circledR A$ *and* $a \circledR B$.
3. $a \circledR (A \Rightarrow B) \triangleq \forall b \in G\ (b \circledR A \supset b \circ a \circledR B)$. *Thus a transforms solutions of A into solutions of B.*
4. $a \circ b \circledR (A \,\&\, B) \triangleq a \circledR A \wedge b \circledR B$. *$A$ solution of B is applied to a solution of A.*
5. $a \circledR \sim A \triangleq a^{-1} \circledR A$. *$a$ undoes a solution of A or prevents it.*
6. $a \circledR E \triangleq a = e$.

The set of realizations for A is denoted $\circledR A$.

Whenever an interpretation I is mentioned it is assumed that I is an interpretation for the signature of our formulas.

Definition 5. A is true *in I if $\circledR A = G$. A* is valid *if A is true in each I. Validity of A is denoted $\models A$.*
A is realizable *in I if $\circledR A \neq \emptyset$. A* is totally realizable *if A is realizable in each interpretation I.*

CRL diverges from other constructive logics. Say, both $A\&A \Rightarrow A$ and $A \Rightarrow A\&A$ are invalid. A is realizable iff $\sim A$ is but these formulas do not imply one another. $A \Rightarrow A$ is totally realizable but is true iff A or $\neg A$ is true. Quantifiers can be expressed here on propositional level. For example

$$\forall A \equiv (A \vee \neg A) \Rightarrow A.$$

Here we have no constructive disjunction. If introduced it demands an «interleaving product» of groups: a group of all products $a_1 \circ b_1 \circ \cdots \circ a_n \circ b_n$ where a_i are from realizations of A and b_i are from one of B. This destroys finiteness and means that conditionals demand increasing memory. Analyzing constructions of Fredkin and Toffoli we see that it is.

So pure reversive programming language is to be without conditionals and loops but from the very beginning functional one [2009A,2009Izh,2009VIZ,2010]. In practice we are to use irreversive operations (at least initializing and result writing) and very restricted use of conditionals and loops. Of course there are no recursions and reversive language is not Turing-complete. Atomic computing elements for reversive computer are to be group-valued not binary.

A basic skeleton for reversive programming language can be the following.

Programs can be pure, conditional, cyclic and generic. Atomic actions are pure. Compositions of pure programs are pure. Functions with pure body are pure.

Pure programs are conditional. A construction

if P **then** t **else** r **fi**

is conditional if t and r are conditional. Compositions of conditional programs and functions with conditional bodies are conditional.

Pure programs are cyclic. A construction

to N **do** t **od**

is cyclic if t is cyclic and N is a constant natural number. Compositions of cyclic programs are cyclic. Functions with cyclic bodies are cyclic.

If pr is a program of each three above classes then -(pr) is a program of the same class. Brackets around atoms and functions can be omitted.

Generic program is a composition of programs of types above. Reversive module has a form
<definitions> <input> <generic program> <output>.
Forms of <definitions>, <input> and <output> parts we will not consider at the moment.

Consider an example of program scheme with one type only and without parameters of functions.

```
DEFINITIONS # All names used in a program are specified here
atom a1, a2, a3, a4, a5, a6, a7
predicate p1, p2
function f=(a1; if p1 then -a2; a3; a1 else a4; -a1 fi)
function g=(a1; to 51 do -a1 od)
function h=(a1; a3; -a1)
END DEFINITIONS
INPUT
   # initial values of all atoms and predicates are given here;
   # usually they are computed by external program
   # and transferred into
p1=¬(a4,a6) # if the domain of a predicate
   # or the value of an atom is fixed for all executions
   # it can be defined inside
   ...
END INPUT
-{to 14 do
   -g; h; a7;
   od; a2}; # we take an inverse of the whole program block
if p2 then -f; h else f fi
```

```
f; -g; -a4; h;
OUTPUT # a substructure transferred to external processor
   # is defined here
   ...
END OUTPUT
```

Thus conditional parts cannot be used inside cyclic parts and vice versa.

The problem of data types for reversive programs is fine and interesting. For example we cannot restrict ourselves by direct products of some standard groups. Consider an example. Let in a conditional statement **if** P **then** t **else** r **fi** functions t and r are computed dynamically. Then if G is a group of programs and H is a group of data we are to represent the corresponding group as follows. Its underlying set is $\mathbb{Z}_2 \times G \times G \times H$, and the group operation is

$$\begin{gathered}\langle z, a_1, b_1, c_1\rangle \circ \langle 0, a_2, b_2, c_2\rangle = \langle z, a_1 \circ a_2, b_1 \circ b_2, c_1 \circ c_2\rangle \\ \langle z, a_1, b_1, c_1\rangle \circ \langle 1, a_2, b_2, c_2\rangle = \langle z \oplus 1, a_1 \circ b_2, b_1 \circ a_2, c_1 \circ c_2\rangle\end{gathered} \tag{3}$$

Say, our program scheme with atoms from a group G needs much more complex group to be executed. The technique of computation of this group during translation of reversive program needs somewhat sophisticated algebraic technique and will be published in a separate paper. Here only note that

1. pure programs do not change a group;
2. each written loop adds an additive constant to the number of the group elements;
3. each executed conditional (roughly speaking) doubles the number of elements in a group.

During computation flow a group remains the same. Each change of a program can change its group.

Therefore a practical reversive processor can be a mill which makes a large amount of transformations with a low number of branches.

Each proof of $A \Rightarrow B$ in a reversive constructive logic gives a pure program to reach a state where B holds from any state with A. By the standard technique of precondition computation described in Gries [1981Gr] we can extend specifications of pure segments up throughout our program by conditions which hold in given points inside our programs. The whole process demands a very sophisticated and original logical tools and was partially described in [2009,2009Izh,2009VIZ].

So we see that realization of reversive programming demands a new theoretical and practical paradigm. It is to compose non-standard logical and programming tools together with algebraic ones into a single system. Underestimation of complexity and misunderstanding of nature of this problem are two main sources of 30-year stagnation in this domain, both in theory and in practice.

And last but not least. A reversive program must be reversive down to the atomic constructions and here we cannot use modules written in other languages.

Moreover reversive modules also cannot be used by other reversive programs because input and output destroy reversivity. We can use only function definitions given in a reversive language.

Applying our considerations of program retraction to the case of reversivity we see that the reversive constructive logic provides an instrument of effective retraction but this retraction is in its essence irreversive. Therefore error or condition analysis for a reversive processor must be performed by traditional one.

5 Conclusion

1. There are three substantially different but usually mixed notions of inverse computability. They need different tools and use different logics.
2. A reversive computation demands full invertibility of actions. Only it can grant minimization of heat pollution.
3. Reversive computability is not Turing-complete and a reversive processor can work only as specialized unit of an usual (for example von Neumann) computer.
4. Binary elements are maybe the worst choice for reversive computation. This process demands group-based elements.
5. It is necessary to compute in a reversive program the algebraic structures of data types and of the whole data space before program compilation because each modification of programs changes all data structures in it. This algebraic computation can be somewhat sophisticated.
6. A reversible computing (unrestricted undoing) can be implemented in traditional computers by traditional programming languages as a discipline of programming.
7. A program retraction (computation of precondition which hold or fail for the given result) can be made by means of almost traditional logic. During retraction values and ghosts are interchanged.

References

1961. Landauer. R: Irreversibility and heat generation in the computing process. IBM J. of R & D, 5, 183–191 (1961)

2001Bub. Bub, Jeffrey: Maxwell's Demon and the Thermodynamics of Computation. Stud. Hist. Phil. Mod. Phys., **32**, No. 4, pp. 569—579 (2001)

2005Mar. Maroney, O J E: The (absence of a) relationship between thermodynamic and logical reversibility. Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics. **36**, Issue 2, 355-–374 (2005)

1973. Bennett, C. H.: Logical reversibility of computation. IBM J. of R & D, 17, no. 6, 525–532 (1973)

1980). Toffoli, T. Reversible Computing. MIT TR MIT/LCS/TM-151 (1980)

1982. Fredkin, E. and Toffoli, T.: Conservative logic. Int.l J.l of Theor.l Phys., 21, 219–253 (1982)

1992. Merkle, R.C.: Towards Practical Reversible Logic. Workshop on Phys. and Comp., PhysComp '92, October, Dallas Texas; IEEE press (1992)
1996. Merkle R. C.: Helical logic. Nanotechnology, 7, 325–339, (1996)
2003. Shende, V. V. Prasad, A.K. Markov, I. L., Hayes, J. P.: Synthesis of Reversible Logic Circuits. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 22(6), 710–722 (2003)
2012L. Antoine Berut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider & Eric Lutz: Experimental verification of Landauer's principle linking information and thermodynamics. Nature, March 2012.
2007. Yokoyama T., Glück R.: A reversible programming language and its invertible self-interpreter. Partial Evaluation and Program manipulation. (2007)
2010Dij. Dijkstra, Edsger W.: A Discipline of Programming. Prentice Hall (2010)
1981Gr. Gries, D.: The Science of Programming. (1981)
2011NNN. Nepejvoda, N.N.: Уроки конструктивизма.
Geidelberg: Lambert Academic Publishing, 98 pp. (2011)
2009. Nepejvoda, N.N.: Реверсивные конструктивные логики. Логические исследования, 15, 150—168 (2009)
2009A. Непейвода А. Н.: О сюръективной импликации в реверсивной логике. VI Смирновские чтения по логике (2009)
2009Izh. Непейвода А. Н. Элементы реверсивных вычислений Управление большими системами труды VI всероссийской школы-семинара молодых ученых, Ижевск (2009))
2009VIZ. Непейвода А. Н.: О реверсивной альтернативе традиционным вычислениям. Трехмерная визуализация научной, технической и социальной реальности. Технологии высокополигонального моделирования : труды Второй междунар. конф., Ижевск (2010).)
2010. Непейвода А. Н.: Функциональное программирование над группой. Системный анализ и семиотическое моделирование: труды первой всероссийской конференции, 2011, Казань (2011)
1979. Заславский И.Д.: Симметрическая конструктивная логика. Ереван, (1979)
NNN1982. Nepejvoda N. N. Logical approach to Programming. Logic, methodology and philosophy of science VI. 109–122 (1982)
Mar1982. Martin-Löf P. Constructive mathematics and computer programming. Logic, methodology and philosophy of science VI. 153–179 (1982)
1991. Nepejvoda N. N. A bridge between constructive logic and computer programming. Theoretical Computer Science, **90** 253–270 (1991)
1998. Nepejvoda N. N. Some analogues of partial and mixed computations in the logical programming approach. New Generation Computing, **17**,309–327 (1999)

Inverting Dynamic Programming

Nikolay 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. We suggest and discuss a formalization of Dynamic Programming. A methodological novelty consists in an *explicit* treatment (interpretation) of ascending Dynamic Programming as least fix-point computation (according to Knaster-Tarski fix-point theorem). This interpretation leads to a uniform approach to classical optimization problems as well as to problems where optimality is not explicit (Cocke - Younger - Kasami parsing algorithm for example) and to problem of algorithm inversion (i.e. computing inverse function).

Keywords: Dynamic programming, Tarsky-Knaster fixpoint theorem, algorithm inversion

1 Introduction

We would like to continue study of algorithm inversion started in [11]. The cited paper [11] is about a fake coin puzzle to be solved in three programming paradigms: logic, functional and imperative. It can be considered as a case study of algorithm inversion, since we start with logic algorithm, that answers the question "*Is balancing M times sufficient for detecting the fake coin?*", and finishes with imperative algorithm, that effectively computes the minimal number of balancing that is sufficient for detection the fake; functional paradigm is used for developing an intermediate functional algorithm that also computes the minimal number of balancing, but inefficiently. Basically, the functional and the imperative solutions of the puzzle are two (recursive and iterative) versions of dynamic programming algorithm that inverts a logical program. In the present paper we generalize background ideas to inversion of recursive dynamic programming.

1.1 Dropping Bricks from a High Tower

Let us start with the following *Dropping Bricks Puzzle*[1].

[1] When draft of this paper was ready, Prof. Teodor Zarkua (St. Andrew University of Georgian Patriarch) informed the author that the problem is known already and has been used for programming contests (check, for example, the problem at URL http://acm.timus.ru/problem.aspx?space=1&num=1223). Recently a variant of the problem has been added to Wikipedia article Dynamic Programming (available at http://en.wikipedia.org/wiki/Dynamic_programming#Egg_dropping_puzzle).

> Let us characterize mechanical stability (strength) of a brick by an integer h that is equal to the height (in meters) that is safe for the brick to fall down, while height $(h + 1)$ meters is unsafe (i.e. the brick breaks). You have to define stability of bricks of a particular kind by dropping them from different levels of a tower of H meters. (You may assume that mechanical stability does not change after safe fall of a brick.) How many times do you need to drop bricks for it, if you have 2 bricks in the stock? What is the optimal number (of droppings) in this case?

Basically the question that we need to answer is how to compute the optimal number of droppings G_H, if the height of the tower is H, and you have 2 bricks in the stock. In the next subsection we sketch descending Dynamic Programming solution of the above problem as a gentle introduction to Dynamic Programming approach to optimization, design its implementation in terms of functional pseudo-code and conclude with historic remarks.

The rest of the paper is organized as follows. In the next section 2 we introduce (what we call) a *scheme of recursive Dynamic Programming* and discuss in brief how to improve efficiency of recursive Dynamic Programming by memoization. In the section 3 we convert recursive Dynamic Programming into the iterative form and interpret Dynamic Programming as a computation of the least fix-point of an appropriate monotone functional. In turn we get an opportunity to design, specify and verify a *unified template for ascending Dynamic Programming*. Two examples of the template specialization are presented in section 4, one of which is context-free parsing. In the section 5 we suggest an approach how to invert algorithms that are based on Dynamic Programming. We discuss some concluding remarks in the last section 6.

1.2 Recursive Method for Optimization Problems

Dropping Bricks Puzzle is a particular and explicit example of optimization problems. Originally Dynamic Programming has been designed as a recursive search (or construction) of an optimal program (or plan) that remains optimal at every stage. For example let us consider below the puzzle.

Any optimal method (to define mechanical stability) should start with some step (command) that prescribes to drop the first brick from some particular (but optimal) level h. Hence the following equality should hold for this particular h:

$$G_H = 1 + \max\{(h-1), G_{H-h}\},$$

where (in the right-hand side)

1. '1+' corresponds to the first dropping,
2. $(h - 1)$ corresponds to the case when the first brick breaks after the first dropping (and we have to drop the remaining second brick from levels 1, 2, ... $(h - 1)$ in the sequence),
3. G_{H-h} corresponds to the case when the first brick is safe after the first dropping (and we have to define stability by dropping pair of bricks from $(H - h)$ levels in $[(h + 1)H]$),

4. 'max' corresponds to the worst in cases 2 and 3 above.

Since the particular value h is *optimal*, and *optimality* means *minimality*, hence the above equality transforms to the next one:

$$G_H = \min_{1\le h\le H}(1+\max\{(h-1), G_{H-h}\}) = 1 + \min_{1\le h\le H}\max\{(h-1), G_{H-h}\}.$$

Besides we can add one obvious equality $G_0 = 0$.

One can remark that sequence of integers G_0, G_1, ... G_H, ..., that meet these two equalities, is unique, since G_0 is defined explicitly, G_1 is defined by G_0, G_2 — by G_0 and G_1, G_H — by G_0, G_1, ... G_{H-1}. Hence it is possible to move from the sequence G_0, G_1, ... G_H, ..., to a function $G : \mathbb{N} \to \mathbb{N}$ that maps every natural H to G_H and satisfies the following system of functional equations for the *objective function* G:

$$\begin{cases} G(0) = 0 \\ G(H) = 1 + \min_{1\le h\le H}\max\{(h-1), G(H-h)\} \end{cases}.$$

This system has unique solution as it follows from the uniqueness of the sequence G_0, G_1, ... G_H, ... Hence this system can be adopted as a recursive definition of a function, i.e. a recursive algorithm presented by its functional pseudo-code. This is an example of the first (historically) face of Dynamic Programming — a recursive method for optimization problems.

Dynamic Programming was introduced as a recursive method for optimization problems by Richard Bellman in 1950s [5]. At this time the noun *programming* had nothing in common with more recent computer programming, but meant *planning* (compare: *linear programming*). The adjective *dynamic* points out that Dynamic Programming is related to *change of state* (compare: *dynamic logic*, *dynamic system*). Functional equations for the objective function (like in the above) are called after Richard Bellman *Bellman equations* as well as the following *Bellman Principle of Optimality* (that we have use already): *an optimal program (or plan) remains optimal at every stage.*

2 Recursion & Memoization vs. Dynamic Programming

If to analyze the recursive Dynamic Programming methodology accumulated in Bellman Principle and the above recursive solution for Dropping Bricks Puzzle, then it is possible to suggest the following scheme of recursive Dynamic Programming.

Definition 1. *Let scheme of recursive Dynamic Programming be the following recursive program scheme [7,8]*

$$G(x) \;=\; if\ p(x)\ then\ f(x)\ else\ g(x,\ (G(t_i(x)),\ i \in [1..n(x)])), \qquad (1)$$

where functional symbol $G : X \to Y$ stays for the objective function, predicate symbol $p \subseteq X$ stays for (i.e. represents or is interpreted by) a known predicate,

functional symbol $f : X \to Y$ stays for a known function, functional symbol $g : X^* \to X$ stays a known function with a variable (but finite) number of arguments $n(x)$, and all functional symbols $t_i : X \to X$, $i \in [1..n(x)]$ stays for known functions also.

It is well-known [8] that the recursive program scheme (with *uninterpreted* symbols) *is not equivalent* to any *standard program scheme* (i.e. a flowchart with uninterpreted symbols with fixed amount of memory), but can be translated to a *program scheme with stack*. Let us remark that in this paper we consider the scheme of recursive Dynamic Programming with *interpreted* symbols. Usually this interpretation is explicit. For example, in Dropping Bricks Puzzle we have

$$G(H) \;=\; if\ H = 0\ then\ 0\ else\ (1 + \min_{1 \le h \le H} \max\{(h-1), G(H-h)\}).$$

Let us compute a value of this function G for a particular argument by exercising the above recursive algorithm in the left-recursive order:

$$\begin{aligned}
&G(4) \;=\; 1 + \min_{1 \le h \le 4} \max\{(h-1), G(4-h)\} \;= \\
&=\; 1 + \min\{\max\{0, G(3)\}\ ,\ \max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ ,\ \max\{3, G(0)\}\} \;= \\
&=\; 1 + \min\{\max\{0, 1 + \min\{\max\{0, G(2)\}\ ,\ \max\{1, G(1)\}\ ,\ \max\{2, G(0)\}\}\}\ , \\
&\qquad\qquad \max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ ,\ \max\{3, G(0)\}\} \;= \\
&=\; 1 + \min\{\max\{0, 1 + \min\{\max\{0, 1 + \min\{\max\{0, G(1)\}\ ,\ \max\{1, G(0)\}\}\}\ , \\
&\qquad\qquad \max\{1, G(1)\}\ ,\ \max\{2, G(0)\}\}\}\ ,\ \max\{1, G(2)\}\ ,\ \max\{2, G(1)\}\ , \\
&\qquad\qquad\qquad\qquad \max\{3, G(0)\}\} \;= \ldots \;=\; 3.
\end{aligned}$$

This exercise illustrates what is called a *descending* Dynamic Programming.

One can remark that in the above example we *recompute* values of G for some arguments several times ($G(2)$ and $G(1)$ in particular). This observation leads to an idea to compute function values for new argument values once, then save them (in a hash-table for example), and use them on demand (i.e. instead re-computation). This technique is known in Functional Programming as *memoization* [4].

Some authors claim that *Recursion + Memoization = Dynamic Programming* [4], but we do not think so due to the following reasons. The first one is historical, since the foundational paper [5] did not discuss memoization at all. The second counterargument relies upon observation that recursion in Dynamic programming has a very special form (never nesting in particular). And finally, there exists also an *iterative form* of Dynamic Programming, that we discuss below.

Definition 2. *Let us consider a function $G : X \to Y$ that is defined by scheme (1) of recursive Dynamic Programming. For every argument value $v \in X$, such that $p(v)$ does not hold, let* base *be the following set $bas(v)$ of values $\{t_i(v) : i \in [1..n]\}$. For every argument value v let* support *be the set $spp(v)$ of all argument values that occur in the computation of $G(v)$.*

Proposition 1. *Let us consider a function $G : X \to Y$ that is defined by interpreted scheme (1) of recursive Dynamic Programming. For every argument value $v \in X$, if the objective function G is defined for v, then $spp(v)$ is finite*

and it is possible to pre-compute *(i.e. compute prior to computation of $G(v)$) the support $spp(v)$ according to the following recursive algorithm*

$$spp(x) \ = \ if \ p(x) \ then \ \{x\} \ else \ \{x\} \cup (\bigcup_{y \in bas(x)} spp(y)). \tag{2}$$

Proof. Correctness of the recursive algorithm 2 can be proved by induction on *recursion depth* in computation of $G(v)$. Ii $G(v)$ is defined then finiteness of $spp(v)$ follows from *König's lemma*, since $bas(u)$ is finite for every argument value u where bas is defined. ■

Definition 3. *Let us consider a function $G : X \to Y$ that is defined by scheme (1) of recursive Dynamic Programming. 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.*

In the case when support or its upper approximation is easy to compute, it makes sense to use *iterative ascending Dynamic Programming* instead of recursive descending Dynamic Programming with memoization.

Ascending Dynamic Programming comprises the following steps.

1. Input argument value v and compute $SPP(v)$. Then compute and save values of the objective function G for all argument values u that are in $SPP(v)$ such that $p(u)$. For example, in Dropping Bricks Puzzle, if we would like to compute value $G(H)$, then $spp(H) = [0..H]$ and the unique argument value of this kind is 0, and, hence, the unique function value that should be saved is $G(0)$; one can save this value as element $G[0]$ of integer array $G[0..H]$.
2. Expand the set of saved values of the objective function by values that can be immediately computed on base of the set of saved values: for every $u \in SPP(v)$, if $G(u)$ is not computed yet, but for every $w \in bas(u)$ value $G(w)$ has been computed and saved already, then compute and save $G(u) = g(u, (G(t_i(u)), i \in [1..n]))$. For example, in Dropping Bricks Puzzle, if values $G(0)$, ... $G(K)$ have been saved in array $G[0..H]$ in elements $G[0]$, ... $G[K]$ (where $0 \leq K < H$), one can compute value $G(K+1) = 1 + \min_{1 \leq k leq K} \max\{(k-1), G(H-k)\}$ and save it $G[K+1]$.
3. Repeat step 2 until the moment, when you save the value of the objective function for the argument value v. For example, Dropping Bricks Puzzle, step 2 should be executed H times and terminate after saving $G[H]$ in $G[0..H]$.

Let us observe that the ascending Dynamic Programming has not a recursive form but the iterative one.

3 Computing the Least Fix-Point

Let us formalize iterative ascending Dynamic Programming by means of imperative pseudo-code annotated by precondition and postcondition [6,3], i.e. by triples in the following form $\{B\}A\{C\}$, where A is an algorithm in pseudo-code, B — is a logical precondition, and C — is a logical postcondition. A triple $\{B\}A\{C\}$ is said to be valid (or that the algorithm A is partially correct with respect to precondition B and postcondition C), if every terminating exercise of A for input data that satisfy B, the output data satisfy C.

Formalization of the ascending Dynamic Programming follows.

\\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}
\\Template:
var Z:= S, Z1 : subsets of D;
repeat Z1:= Z ; Z:= F(Z) until (ρ(P,Z) or Z=Z1)
\\Postcondition:
{$\rho(P,Z) \Leftrightarrow \rho(P,T)$,
where T is the least fix-point of the mapping $\lambda Q.(S \cup F(Q))$}

We would like to refer this formalization as *ascending Dynamic Programming template*, since (as we will see in the section 4) many particular instances of ascending Dynamic Programming algorithms can be generated from this template by specialization of the domain D, sets S and P, and function F. (Like many instances of backtracking and branch-and-bound algorithms can be generated from the unified template that is presented and verified in [12].)

Partial correctness of the formalized ascending Dynamic Programming template follows from Knaster-Tarski fix-point theorem [9]. We would not like to present the exact formulation of the theorem, but would like to present the following proposition that is a trivial corollary from the theorem.

Proposition 2. *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: $R_0 = \emptyset$ 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 $R_0 \subseteq R_1 \subseteq R2 \subseteq \ldots R_k \subseteq R_{k+1} \subseteq \ldots \subseteq T$.*

The following proposition is a trivial consequences of the above proposition.

Proposition 3. *Dynamic Programming template is partially correct, i.e. for any input data that meets the precondition, the algorithm instantiated from the template either loops or halts in such a way that the postcondition holds upon the termination. Assuming that for some input data the precondition of the Dynamic Programming template is valid, and the domain D is finite, then the algorithm instantiated from the template terminates on these data after (at most) $|D|$ iterations of the loop* `repeat-until`.

Proof. Let us assume that a particular instance of the template terminates for some input data that meets the precondition. According to the above proposition 2, the following function $G = \lambda Q.(S \cup F(Q)) : \ 2^D \to 2^D$ (that maps every $Q \subseteq D$ to $S \cup F(Q)$) has the least fix-point. Let $R_0 = \emptyset$ and $R_{k+1} = G(R_k)$ for every $k \geq 0$; then for every $k > 0$ values of set variables `Z` and `Z1` immediately after k iterations of the loop are R_{k+1}, and R_k respectively, and (according to proposition 2) $R_k \subseteq T$, where is the least fix-point of the mapping G. Hence, if the repeat-loop terminates due to condition $\rho(P, \mathrm{Z})$, then $\rho(P, T)$ due to monotonicity of G; if this loop terminates, but not due to the condition $\rho(P, \mathrm{Z})$ (i.e. this condition is not valid), then it terminates due to another condition `Z=Z1`, that implies that the final value of `X` is equal to the least fix-point T, and hence $\rho(P, T)$ is not valid also. ■

4 Examples of the Template Specialization

In this section we illustrate how the ascending Dynamic Programming template works, i.e. how concrete algorithms can be generated from it by specialization (i.e. by instantiating concrete functions and predicates).

4.1 Computing Dynamic Programming

Let us start with Dropping Bricks Puzzle and adopt

- D to be an "initial segment" of the graph of the function G, i.e. the set of all integer pairs $(m, G(m))$, where m represents a level (in $[1..H]$);
- S to be a singleton set $\{(0, 0)\}$ that consists of the unique trivial pair, and P to be another singleton set $\{(H, G(H))\}$;
- F to be a function that maps any $Q \subseteq D$ to $\{(m, n) \in D \mid$ there exist integers $n_0, \ldots n_{m-1}$ such that $(0, n_0), \ldots (m-1, n_{m-1}) \in Q$ and $n = 1 + \min_{1 \leq k \leq m} \max\{(k-1), n_{m-k}\}\}$;
- $\rho(P, Q)$ to be $\exists n : (H, n) \in (P \cap Q)$.

This specialization meets the precondition of the template of the ascending Dynamic Programming, and D is the least fix-point of F. Hence (according to proposition 3), the resulting algorithm terminates after H iterations of the repeat-loop (since $|D| = H$), and (upon the termination) $(H, G(H)) \in \mathtt{Z}$ (since $\exists n : (H, n) \in (P \cap \mathtt{Z}) \Leftrightarrow \exists n : (H, n) \in (P \cap T)$ where T is D, the fix-point of F), but there is no any other $n \neq G(H)$ such that $(H, n) \in \mathtt{Z}$ (since P is a singleton).

The above example can be generalized as follows.

Proposition 4. *Let us consider a function $G : X \to Y$ that is defined by scheme (1) of recursive Dynamic Programming. Assume that $SPP : X \to 2^X$ is some upper approximation of the support function for G. Let $v \in X$ be any value. If to adopt*

- *the graph of G restricted on $SPP(v)$ as D,*
- *a set $\{(u, f(u)) | p(u)$ & $u \in SPP(v)\}$ as S,*

- *a singleton* $\{(v, G(v))\}$ *as* P,
- *a mapping* $Q \mapsto \{(u, w) \in D \mid \exists w_1, \ldots w_n : (t_1(u), w_1), \ldots (t_n(u), w_n) \in Q \ \&\ w = g(u, w_1, \ldots w_n)\}$ *as* $F : 2^D \to 2^D$,
- $\exists w : (v, w) \in (R \cap Q)$ *as* $\rho(R, Q) : 2^D \times 2^D \to Bool$,

then the algorithm that results from the template of the ascending Dynamic Programming computes $G(v)$ *in the following sense: it terminates after iterating repeat-loop* $|SPP(v)|$ *times at most, upon the termination* $(v, G(v)) \in \mathtt{Z}$ *and there is no any* $w \in Y$ *(other than* $G(v)$*) such that* $(v, w) \in \mathtt{Z}$.

Proof. The described specialization meets the precondition of the template of the ascending Dynamic Programming, and D is the least fix-point of F. Hence (according to proposition 3), the resulting algorithm terminates after at most $|SPP(v)|$ iterations of the repeat-loop (since $|D| \leq |SPP(v)|$), and (upon the termination) $(v, G(v)) \in \mathtt{Z}$ (since $\exists w : (v, w) \in (P \cap \mathtt{Z}) \Leftrightarrow \exists w : (v, w) \in (P \cap T)$ where T is D, the fix-point of F), but there is no any other $w \neq G(v)$ such that $(v, w) \in \mathtt{Z}$ (since P is a singleton). ■

4.2 Context-Free Parsing

Parsing theory for context-free (C-F) languages is well established and developed technology [1,2]. The first sound and efficient algorithm for parsing C-F languages was developed independently by J. Cocke, D.H. Younger and T. Kasami in period from 1965 to 1970. More efficient and practical parsing algorithms have appeared since these times, nevertheless Cocke - Younger - Kasami algorithm (CYK algorithm) still has educational importance nowadays[2]. A context-free grammar (C-F grammar) is a tuple $G = (N, E, P, S)$, where

- N and E are disjoint finite alphabets of *non-terminals* and *terminals*,
- $P \subseteq N \times (N \cup E)^*$ is a set of *productions* of the following form $n \to w$, $n \in N$, $w \in (N \cup E)^*$,
- $s \in N$ is the *initial* non-terminal.

A C-F grammar is in the Chomsky Normal Form (CNF) if the initial symbol does not occur in the right-hand side of any production, and every production has the form $n \to n'n''$ or $n \to e$, where $n, n', n'' \in N$ and $e \in E$. *Derivation* in a C-F grammar G is a finite sequence of words $w_0, \ldots w_k, w_{k+1}, \ldots w_m \in (N \cup E)^*$, $(m \geq 0)$, such that every word w_{k+1} within this sequence results from the previous one w_k by applying a production (in this grammar). For every words $w', w'' \in (N \cup E)^*$ let us write $w' \Rightarrow w''$ if there exists a derivation that starts with w' and finishes with w''. Language $L(G)$ generated by the grammar G is defined as follows: $L(G) = \{w \in E^* \mid s \Rightarrow w\}$.

Two C-F grammars are said to be equivalent if they generate equal languages. It is well-known fact that every C-F grammar that does not generate the empty word is equivalent to some CNF grammar [1].

[2] Recently M. Lange and H.F. Leiß suggested a generalized CYK algorithm for educational purposes [10].

Definition 4. *Assume that $G = (N, E, P, s)$ is a given C-F grammar.* Parsing problem *for $L(G)$ can be formulated as follows: for input word $w \in E^*$ construct the set of all pairs (n, u), $n \in N$ and $u \in E^*$ is a (non-empty) subword of w, such that $n \Rightarrow u$.*

In the sequel we discuss parsing problem for CNF grammars only. Let $G = (N, E, P, s)$ be a CNF grammar, $w \in E^*$ be the input word, $L = L(G)$ be the corresponding language, D be the set of all pairs (n, u), where $n \in N$ and $u \in E^*$ is a (non-empty) subword of w, and $T = \{(n, u) \in D \mid n \Rightarrow u\}$. It is easy to see that T is the least fix-point of the following monotone function $F : 2^D \to 2^D$ that maps every $Q \subseteq D$ to $F(Q) = \{(n, e) \in D \mid e \in E,\ (n \to e) \in P\} \cup \{(n, u) \in D \mid \exists (n^p rime, u^p rime), (n^p rime^p rime, u^p rime^p rime) \in Q : u \equiv u^p rime u^p rime^p rime$ and $(n \to n^p rime n^p rime^p rime) \in P\}$. Hence we can adopt $\{(n, e) \in D \mid e \in E,\ (n \to e) \in P\}$ as S, $\{(s, w)\}$ as P in the ascending Dynamic Programming template, and predicate $FALSE$ as ρ. After this specialization the template becomes CYK algorithm that solves the parsing problem for $L(G)$ by iterating repeat-loop at most $|N| \times |w|$ times.

5 Inverting Descending Dynamic Programming

Definition 5. *Let $G : X \to Y$ be a function. A function $G^- : Y \to X$ is said to be* inverse *of G if the following properties hold:*

- *for every $w \in Y$, if $w \in G(X)$ then $G^-(w)$ is defined and $G(G^-(w)) = w$;*
- *for every $w \in Y$, if $w \notin G(X)$ then $G^-(w)$ is undefined.*

Let us remark, that if a function $G : X \to Y$ is not injective, then G^- is not unique.

Let us assume that some total function $G : X \to Y$ is defined by recursive scheme of Dynamic Programming 1. Let us assume also that X is countable (with some fixed enumeration $cnt : N \to X$), that we have the following abstract data type $SubSet$ which values are subsets of X (i.e. all subsets, not just finite), that has standard set-theoretic operations *union* and *intersection* (applicable when *at least one* argument is finite) and another choice operation $fir : SubSet \to X$ that computes for every set T the element of T with *the smallest number* (according to *cnt*).

Assume that we want to design an algorithm that computes *some* inverse of G. The simplest way to compute $G^-(w)$ for a given $y \in Y$ is to proceed one by one according to count as follows.

```
\\ Precondition:
{G : X → Y is a total computable function,
X is a countable set,
Fir : SubSet → X is a choice function, y ∈ Y}
\\ Algorithm:
var x: X; var z: Y; var R:=X: SubSet;
repeat x:= Fir(R); z:= G(x); R:= R\{x} until (z=y or R=∅);
```

```
if y≠z then loop
\\ Postcondition:
```
$\{G(x) = y\}$.

Partial correctness of this algorithm is straightforward, but without memoization this algorithm is extremely inefficient. More efficient way to compute the inverse function is presented below as the following *Inverse Dynamic Programming algorithm.*

```
\\ Precondition:
```
$\{G : X \to Y$ `is a computable function`
`defined by scheme of recursive Dynamic Programming (1),`
$SPP : X \to 2^X$ `is an upper support approximation for` G`,`
X `is a countable set,` $Fir : SubSet \to X$ `is a choice function,`
$y \in Y\}$

```
\\ Algorithm:
var x: X; var R:=X, T: SubSet;
var D:=∅: 2^{X×Y};
var k : integer;
repeat x:= Fir(R); T:= SPP(x); R:= R\T;
     D:= D ∪ {(u, f(u)) | p(u) & u ∈ T};
     exercise k∈ [1..|T|] times: D:= D ∪ {(u, w) ∉ D |
                                  ∃w_1, ... w_n : (t_1(u), w_1), ... (t_n(u), w_n) ∈ D,
                                  t_1(u), ... t_n(u) ∈ T, & w = g(u, w_1, ... w_n)}
until (∃u : (u, y) ∈ D or R=∅);
if ∃u : (u, y) ∈ D then x:= (u such that (u, y) ∈ D) else loop
\\ Postcondition:
```
$\{G(x) = y\}$. (Parameter k may be any in the specified range and, maybe, it can be determined by supercompilation [13,14].)

Proposition 5. *Inverse Dynamic Programming algorithm is partially correct.*

Proof. One can proceed according to Floyd - Hoare method [6,3] and use the following (one and the same) invariant to both loops (i.e. for the external repeat-loop and for the internal exercise-loop): D is a subset of graph of G. ■

Proposition 6. *Assume that for some input data the precondition of the Inverse Dynamic Programming algorithm is valid and that the input value y belongs to $G(X)$. Then the algorithm eventually terminates.*

Proof. A standard way to prove algorithm (and program) termination is via *potential* (or bound) function [6,3], i.e. a function that maps states of the algorithm to natural numbers so that every legal loop execution reduces value of the function. Let $n \in N$ be an integer such that $y = G(cnt(n))$, let $m = \sum_{0 \le i \le n} |SPP(cnt(i))|$ and let $\pi(D) = m - |D|$. Then every legal iteration of any loop of our algorithm reduces the value of this function $\pi(D)$ at least by one. ■

As follows from propositions 5 and 6, the inverse Dynamic Programming really computes an inverse function for a function defined by recursive scheme for descending Dynamic Programming.

Let us present an example. It does not make sense to invert function G that solves Dropping Bricks Puzzle, since this function is not injective. So let us consider a simpler injection function $F : N \to N$

$$F(n) \;=\; if\ (n = 0\ or\ n = 1)\ then\ 1\ else\ F(n-1) + F(n-2)$$

that computes Fibonacci numbers. Let us assume that *cnt* is enumeration in the standard order. Then our Inverse Dynamic Programming algorithm gets the following form:

```
var x: N; var R:=N, T: 2^N;
var D:=∅ : 2^{N×N};
var k : integer;
repeat x:= Fir(R); T:= [0..x]; R:= R\[0..x];
     D:= D ∪ {(0,1) | 0 ∈[0..x]} ∪ {(1,1) | 1 ∈[0..x]};
     exercise k∈[1..x] times: D:= D∪ {(u,w) ∉ D |
                                        ∃w1,w2 : (u−1,w1),(u−2,w2) ∈ D,
                                   (u−1),(u−2) ∈[0..x], & w = w1 + w2}
until ∃u : (u,y) ∈ D;
if ∃u : (u,y) ∈ D then x:= (u such that (u,y) ∈ D) else loop.
```

After some simplification one can get the following algorithm:

```
var x: N; var T: 2^N; var D:=∅: 2^{N×N}; var k : integer;
x:=0; D:= {(0,1),(1,1)};
repeat x:=x+1; D:= D ∪ {(x,w1 + w2) |
                                        ∃w1,w2 : (x−1,w1),(x−2,w2) ∈ D};
until ∃u : (u,y) ∈ D;
if ∃u : (u,y) ∈ D then x:= (u such that (u,y) ∈ D) else loop
```

that just computes and saves Fibonacci sequence in the "array" `D` and checks whether y is in the array already.

6 Concluding Remarks

Author would not like to forth everyone to think about Dynamic Programming in terms of fix-point computations, but believe that ascending Dynamic Programming template presented in the paper will help to teach and (maybe) automatize Algorithm Design. This approach to teaching Dynamic Programming is in use in Master program at Information Technology Department of Novosibirsk State University since 2003. A possible application of the unified template is data-flow parallel implementation of the Dynamic Programming, but this topic need more research.

Acknowledgments. The research is supported by joint Russian-Korea project RFBR-12-07-91701-NIF-a.

References

1. Aho, A.V., Ullman, J.D.: The Theory of Parsing, Translation, and Compiling, Vol. 1, Parsing. Prentice Hall (1972)

2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd Edition. Addison-Wesley (2007)
3. Apt, K.R., de Boer, F.S., Olderog, E.-R.: Verifcation of Sequential and Concurrent Programs. Third edition, Springer (2009)
4. Astapov, D.: Recursion + Memoization = Dynamic Programming. (In Russian.) Practice of Functional Programming n.3,17–33, available at `http://fprog.ru/2009/issue3/` (2009)
5. Bellman, R.: The theory of dynamic programming. Bulletin of the American Mathematical Society 60, 503–516 (1954)
6. Gries, D.: The Science of Programming. Springer (1987)
7. Greibach, S.A.: Theory of Program Structures: Schemes, Semantics, Verification. Lecture Notes in Computer Science 36, Springer, Heidelberg (1975)
8. Kotov V.E., Sabelfeld V.K.: Theory of Program Schemata. (Teoria Skhem Programm.) Science (Nauka), Moscow (1991) (in Russian)
9. Knaster, B., Tarski, A.: Un theoreme sur les fonctions d'ensembles. Ann. Soc. Polon. Math., 6, 133–134 (1928)
10. Lange, M., Leiß, H.: To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm. Informatica Didactica 8, available at `http://www.informatica-didactica.de/cmsmadesimple/index.php?page=LangeLeiss2009_en` (2009).
11. Shilov, N.V.: A note on three Programming Paradigms. In: 2nd International Valentin Turchin Memorial Workshop on Metacomputation in Russia, pp.173–184. Ailamazyan Program Systems Institute, Pereslavl-Zalessky, Russia (2010)
12. Shilov, N.V.: Algorithm Design Template base on Temporal ADT. Proceedings of 18th International Symposium on Temporal Representation and Reasoning, IEEE Computer Society, 157–162 (2011)
13. Turchin, V.F.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems, 8(3), 292–325 (1986)
14. Turchin, V.F.: Supercompilation: the approach and results. (Superkompilyatsya: metody i rezultaty.) In: Current trends in architecture, design and implementation of program systems. (Problemy arkhitektury, analiza i razrabotki programmnyh system.). System Informatics (Sistemtaya Informatika) 6, Science (Nauka), Novosibirsk, 64–89 (1998) (in Russian)

Lightweight Polytypic Staging of DSLs in Scala*

Alexander V. Slesarenko

Keldysh Institute of Applied Mathematics, Moscow, Russia,
avslesarenko@gmail.com,
WWW home page: http://pat.keldysh.ru/~slesarenko/

Abstract. This paper describes *Lightweight Polytypic Staging*, - a new approach to the implementation of *deep embedding* of DSLs. We use the notion of *polytypic DSL*, – the DSL which is designed and implemented by means of polytypic (data-generic) programming techniques.
We show how to combine various *lightweight* techniques available in the Scala language (techniques based on expressive type system of the language). In particular, we use polytypic (data-generic) programming, polymorphic embedding, Lightweight Modular Staging (LMS) and language virtualization.
The combination of polytypic programming and staging gives us new opportunities for optimizations by transformation. It is traditional in polytypic programming to implement a user-defined data type by first, providing an isomorphic representation of the type built out of *sums of products* and second, by defining semantics of domain primitives only for sums of products. In polytypic staging context we introduce an *isomorphism lifting*, – a transformation that automatically *lifts* isomorphisms out of the domain code and separates the domain semantics from the user-defined views.
The implementation is based on the Scala-Virtualized compiler (an extension to facilitate deep DSL embedding) which makes the staging almost transparent to the DSL user (non-staged and staged code looks literally the same). We show how to apply polytypic staging to a particular domain by describing an implementation of the corresponding DSL. The domain is nested data parallelism and the DSL is the nested data parallel language embedded in Scala. The paper is organized around the specific DSL, but our implementation pattern should be applicable to any polytypic DSL in general.

Keywords: Generic programming, Polytypic programming, Polytypic staging, Nested Data Parallelism, Multi-stage programming, Domain-specific languages, Language Virtualization

1 Introduction

A long-standing trend in software development for parallel computing is the reduction of complexity, namely the development of easy-to-use languages and

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

libraries [8,9,24], encapsulation of complexity in an implementation of system software [21], creation of interactive working environments [1].

In particular, it was shown [28,26] that a combination of a DSL approach and program staging is a promising direction of work where sufficient performance optimizations of staged code were achieved by exploiting domain-specific semantics. So staging is a key point of DSL optimizations.

But what if the DSL for our domain can be naturaly implemeted by using polytypic (generic) programming techniques? How can we stage generic code? Interestingly, this is the case when we consider nested data parallelism (NDP) as the domain. In our previous work[27] we developed an embedded *polytypic DSL* for expressing nested data parallel algorithms in Scala language by using *generic programming* [12] (*polytypic programming* [17]) techniques.

In this paper, we describe an attempt to stage our polytypic DSL, hence we term this as *polytypic staging*. The implementation is *lightweight* in a sence that it is based on an expressive type system of the Scala language. In section 5 we briefly compare the lightweight staging approach with a traditional multi-stage programming [29].

The idea behind our approach that is proposed here is based on a combination between Lightweight Modular Staging (LMS) [25] and polytypic (datatype-generic) programming. The idea is that by writting programs using a polymorphic embedding style [14], programs can be interpreted in two modes: simulation and code generation. In the simulation mode programs are given an unoptimized (and slow), but straightforward implementation which is good for testing. In code generation mode, a fast, optimized implementation is generated at runtime. Datatype-generic programming techniques are then applied to allow the library to be specialized with user-specific datatypes (built out of arrays, sums and products) by providing isomorphic views types [15]. Term rewriting techniques can be applied on the staging (code generation) phase to perform generic and domain specific optimizations.

For domain specific foundations we rely on a series of publications [19,4,20] on the *nested data parallelism model*. The model of NDP was first formulated in the early 90's [3], but still is not widely used in practice, although there is a series of publications and a publicly available implementation [5]. On the other hand, many techniques and technologies [2,7,14,23,25], which we use as a foundation of our approach, appeared only in recent years so it is an interesting research question to restate the problem and implement the model in a new environment.

We propose our implementation of NDP as a DSL embedded in the Scala-Virtualized as the host language and packaged as a library. We compare it with Parser Combinators library which also has limited expressiveness and inherent composability, while still having a wide range of applications in different problem domains.

From the DSL point of view, we regard our previous implementation as *shallow embedding* as oppose to *deep embedding* that is described in this paper and which is consistent with our previous work.

In summary, this paper makes the following main contributions:

1. We extend our previously published work [27] by introducing type-directed *Lightweight Polytypic Staging* technique (LPS).
2. We describe how to extend the Lightweight Modular Staging (LMS) framework by making it polytypic (datatype-generic) over a family of type constructors: sum, product and array.
3. We show how our framework is able to support user-specific data types by providing isomorphic representations.
4. We show how the combination of polytypic programming and staging techniques gives us new opportunities for optimizations by transformation by introducing *isomorphism lifting*, – a transformation that automatically *lifts* isomorphisms out of domain code and separates domain semantics from user-defined views.
5. We show how to apply Lightweight Polytypic Staging to a special problem domain of nested data parallelism.

In this paper we also describe some aspects of the design and implementation of the Scalan library. [1]

1.1 The DSL

We start with some examples of the DSL to illustrate the basic ideas of the NDP domain from user's perspective.[2]

Consider the definition of `sparseVectorMul` in Fig. 1. We represent a sparse vector as an array of pairs where the integer value of the pair represents the index of an element in the vector and the float value of the pair represents the value of the element (compressed row format). Having this representation, we can define a dot-product of sparse and dense vectors as a function over arrays.

```
trait PArray[A]
type VectorElem = (Int,Float)
type SparseVector = PArray[VectorElem]
type Vector = PArray[Float] // dense vector
type Matrix = PArray[SparseVector] // sparse matrix
def sparseVectorMul(sv: SparseVector, v: Vector): Float =
      sum(sv map { case Pair(i,value) ⇒ v(i) * value })
def matrixVectorMul(matr: Matrix, vec: Vector): Vector =
      for (row <- matr) yield sparseVectorMul(row, vec)
```

Fig. 1. Sparse Matrix Vector Multiplication

[1] The complete code is available at http://github.org/scalan to supplement the paper.

[2] We extensively use Scala listings in the paper and assume familiarity with the language [22]. We only show parts of the code relevant to our discussion and refer to our previous paper [27] for details of the library design and more samples.

Instead of using the ordinary `Array[T]` type we use an abstract `PArray[T]` trait and by doing that, first, make the code abstract, and second, express our intent for a parallel evaluation.

When it comes to multiplying a sparse matrix with a dense vector, we can reuse our previously defined parallel function `sparseVectorMul` to define a new parallel function `matrixVectorMul`. This is the essence of nested data parallelism, on the one hand, we are able to nest one parallel map inside another parallel map and, on the other hand, it supports flattening that makes it possible to automatically transform any nested code into flat form which is good for execution. And that is the reason why we need staging in this domain in a first place, to be able to perform transformations.

We are free up to a family of product, sum and PArray type constructors (see Fig. 2) to define data types and in fact it is our responsibility as a programmer to define them properly. It is our choice here to represent sparse matrix as a parallel array of sparse vectors and not dense ones (as they can have considerably different memory usage characteristics). But what the polytypic DSL gives us is that for any data type we define it provides us with the specialized underlying data structure that is built in a generic way from the type definition (see section 2.4).

```
A,B = Unit | Int | Float | Boolean // base types
| (A,B) // product (pair of types)
| (A|B) // sum type where (A|B) = Either[A,B]
| PArray[A] // nested array
```

Fig. 2. Family of element types

We can also use a parallel function inside its own definition i.e. recursively. Fig. 3 shows how the QuickSort recursive algorithm can be expressed in the NDP model.

The DSL is purely functional, sequential and deterministic. The program can be thought of as being executed by the vector virtual machine where each array primitive is executed as one data-parallel step. We express parallelism (what we want to be executed in parallel and what we don't) by using types of an input data (`PArray` in this case), intermediate data (i.e. `subs` which has type `PArray[PArray[Int]]`) and also by using combinators over parallel data types (map, partition).

Note how `partition` increases the nesting level so that we can express the idea that both partitions should be executed in parallel using `map`. And then results are combined back in a flat array by `concat` which has the following type

```
def concat[A:Elem](a: PA[PA[A]]): PA[A]
```

```
trait PArray[T] {
  def partition(flags:PA[Boolean]):PA[PA[T]]
}
type PA[A] = PArray[A]
def qsort(xs: PA[Int]): PA[Int] = {
  val len = xs.length
  if (len <= 1) xs
  else {
    val pivot = xs(len / 2)
    val less = xs map { x ⇒ x < pivot }
    val subs = xs.partition(less)
    val sorted = subs map { sub ⇒ qsort(sub) }
    concat(sorted)
  }
}
```

Fig. 3. Parallel QuickSort

The point is that `concat` is a constant-time operation, and that is possible because the representation of the type `PA[PA[A]]` is specially choosen to support this. You can look at the Fig. 7 and probably guess how `concat` is implemented.

The implicit annotation `A:Elem` expresses a requirement that the type parameter A should be an instance of the type-class `Elem[A]` [7], which means, as we will see later, that A is either built by using products, sum, and PArray constructors, or it is a user-specific data type isomorphic to some B:Elem. It is *not just any* user defined Scala type but any Scala type can be made into type-class `Elem` by providing an isomorphism.

We systematically use the techniques described in [7] to implement polytypism in our DSL. In particular, in the section 2 we will see how we can define generic functions once and for all instances of the type-class `Elem`.

1.2 Adding More Types

If we limit the typing capabilities of the DSL to just the types shown in Fig. 2 we still be possible to cover many practical cases. It is limitied approach though, since we cannot define recursive data types in this way due to the limitations imposed by the Scala language itself. And it is not convenient for the user.

To both overcome this limitation and increase typing capabilities of the DSL we make it possible to extend the family of types shown in Fig. 2 with any user-specific type defined in Scala. The key point is to be able to make any such type `U` an instance of type-class `Elem`. The idea is to define a *canonical*[3] isomorphism (iso for short) between `U` and some existing instance `A:Elem`. This finally ensures

[3] Canonical isos are special because they are uniquely determined by the types involved, that is, there is at most one canonical iso between two polymorphic type schemes.

that every user-specific type is represented by an isomorphic view type [15]. It suffices to define a function on view types (and primitive or abstract types such as Int and Boolean) in order to obtain a function that can be applied to values of arbitrary data types.

Consider as an example the definition of the `Point` type shown in Fig. 4. Given a user-specific type (`Point` in this case) all we need to do is to define an instance of `Iso[A,B]` type-class (see `IsoPoint`) witnessing that `Point` is canonically representable in terms of already defined instances of the `Elem` type-class.

```
case class Point(x: Int, y: Int)
implicit object IsoPoint extends Iso[(Int, Int), Point] {
  def to = (p: (Int, Int)) ⇒ Point(p._1, p._2)
  def from = (p: Point) ⇒ (p.x, p.y)
}
def distance(p1: Point, p2: Point): Float = {
  val dx = p2.x - p1.x
  val dy = p2.y - p1.y
  sqrt(dx * dx + dy * dy)
}
def minDistance(ps: PArray[Point]): Float =
  min(for (p <- ps) yield distance(Point(0,0), p))

case class Circle(loc: Point, r: Int)
implicit object IsoCircle extends Iso[(Point, Int), Circle] {
  def from = (c: Circle) ⇒ (c.loc, c.r)
  def to = (c: (Point, Int)) ⇒ Circle(c._1, c._2)
}
```

Fig. 4. User-specific data type

Once the `Point` type is made an instance of the `Elem` type-class via isomorphism it can in turn be used to both define other user-specific types and participate in the isomorphims definitions for those types as it is shown in Fig. 4. We describe the design of these features in section 4.

In our polytypic staging framework we are able to give both evaluation and staging interpretation of all the examples discussed so far. This is described in sections 3 and 4.

1.3 Outline

This paper is organized as follows. Section 2 briefly introduces the theoretical foundations and techniques we use in our impelemtation. Section 3 shows the design of the polytypic staging framework. Section 4 describes our handling of user-specific data types by extending the polytypic staging framework with the generic views on data types. Related work and conclusions are given in Section 5.

2 Foundations of our approach

2.1 Polymorphic Embedding of DSLs

It is well known that a domain specific language (DSL) can be embedded in an appropriate host language [16]. When embedding a DSL in a rich host language, the embedded DSL (EDSL) can reuse the syntax of the host language, its module system, typechecking(inference), existing libraries, its tool chain, and so on.

In *pure embedding* (or *shallow embedding*) the domain types are directly implemented as host language types, and domain operations as host language functions on these types. This approach is similar to the development of a traditional library, but the DSL approach emphasizes the domain semantics: concepts and operations of the domain in the design and implementation of the library.

Because the domain operations are defined in terms of the domain semantics, rather than the syntax of the DSL, this approach automatically yields compositional semantics with its well-known advantages, such as easier and modular reasoning about programs and improved composability. However, the pure embedding approach cannot utilize domain semantics for optimization purposes because of tight coupling of the host language and the embedded one.

Recently, *polymorphic embedding* - a generalization of Hudaks approach - was proposed [14] to support multiple interpretations by complementing the functional abstraction mechanism with an object-oriented one. This approach introduces the main advantage of an external DSL, while maintaining the strengths of the embedded approach: compositionality and integration with the existing language. In this framework, optimizations and analyses are just special interpretations of the DSL program.

Considering advantages of the polymorphic embedding approach we employ it in our design. For details we refer to [14]. Conside the following example

```
type Rep[A]
trait PArray[A]
type SparseVector = PArray[(Int,Float)]
type Vector = PArray[Float]
def sparseVectorMul(sv: Rep[SparseVector], v: Rep[Vector]) =
  sum(sv map { case Pair(i,value) ⇒ v(i) * value })
```

On the DSL level we use product, sum and PArray type constructors and express domain types as Scala's abstract types (see `SparseVector`). Moreover, we lift all the functions over abstract type constructor Rep. This is important because later we can provide concrete definitions yielding specific implementations.

Our sequential implementation (we call it *simulation*) is implemented by defining `Rep` as

```
type Rep[A] = A
```

And in our staged implementation (we call it *code generation*) is implemented by defining Rep as

```
type Rep[A] = Exp[A]
```

where `Exp` is a representation of terms evaluating to values of the type A. Later we will see how it is defined in LMS framework.

The ultimate goal is to develop a polymorphically embedded DSL in the Scala language in such a way that the same code could have two different implementations with equivalent semantics. And thus we would benefit from both simulation (evaluation for debugging) and code generation (for actual data processing).

2.2 Generic programming

In addition to the polymorphic embedding technique, we also need a couple of others that were recently developed in the area of generic programming. We shall briefly overview them here starting with the notion of Phantom Types [6,11].[4]

Consider the definition of a data type (in a Haskell-like notation) shown in Fig. 5.

```
data Type τ =
  RInt with τ = Int
| RChar with τ = Char
| RPair (Type α) (Type β) with τ = (α, β)
| RList (Type α) with τ = [α]
```

Fig. 5. Type descriptors as phantom types

Types defined this way have some interesting properties:

- `Type` is not a container type: an element of `Type Int` is a runtime representation of type `Int`; it is not a data structure that contains integers.
- We cannot define a mapping function $(\alpha \to \beta) \to (\texttt{Type}\ \alpha \to \texttt{Type}\ \beta)$ as for many other data types.
- The type `Type` β might not even be inhabited: there are, for instance, no type descriptors of type `Type String`

It has been shown [11] that phantom types appear naturally when we need to represent types as data at runtime. In our DSL we make use of phantom types to represent types of array elements as runtime data (see Fig. 10) and staged values (see section 3).

Runtime type representations have been proposed as a lightweight foundation of generic programming techniques [10]. The idea is to define a data type whose elements (instances) represent types of data that we want to work with. A *Generic Function* is one that employs runtime type representations and is defined by induction on the structure of types.

[4] We could have used a more general notion of GADT [18] but we stick with phantom types as they are simplier and well enought for our presentation.

Consider again the definition of the data type `Type` in Fig. 5. An element `rt` of type `Type` τ is a runtime representation of τ. For example, following is a representation of type `String`.

```
rString :: Type String
rString = RList RChar
```

A generic function pattern matches on the type representation and then takes the appropriate action.

```
data Bit = 0|1
compress :: forall τ.Type τ → τ → [Bit]
compress (RInt) i = compressInt i
compress (RChar) c = compressChar c
compress (RList ra) [ ] = 0:[]
compress (RList ra) (a : as) = 1 : compress ra a ++ compress (RList ra) as
compress (RPair ra rb) (a, b) = compress ra a ++ (compress rb b)
```

We assume here that two functions are given

```
compressInt :: Int → [Bit]
compressChar :: Char → [Bit]
```

2.3 Generic programming in Scala

Generic functions can be encoded in Scala using an approach suggested in [23]. Fig. 6 shows the encodings in Scala for the above function `compress`.[5]

```
trait Rep[A]
implicit object RInt extends Rep[Int]
implicit object RChar extends Rep[Int]
case class RPair[A,B](ra:Rep[A], rb:Rep[B]) extends Rep[(A,B)]
implicit def RepPair[A,B](implicit ra:Rep[A], rb: Rep[B]) = RPair(ra,rb)
def compress[A](x:A)(implicit r:Rep[A]):List[Bit] = r match {
  case RInt ⇒ compressInt (x)
  case RChar ⇒ compressChar (x)
  case RPair(a, b) ⇒ compress(x._1)(a) ++ compress(x._2)(b)
}
```

Fig. 6. Generic function in Scala

Traditionally, generic (polytypic) functions are defined for a family of types built out of sums and products. We add `PArray` to the family of representation types. Definition of a generic function should be given for each representation

[5] The definition of `compress` for the case `RList` is straightforward and we leave it as an exercise.

type as shown in Fig. 6. For all other types it is usually required to give an isomorphic representation of the type in terms of the above fixed set of constructors. We give an account of isomorphic representations in section 4.1.

In the impelementation of the DSL we use similar techniques and type representations to implement array combinators as generic functions. But because parallel arrays that we discuss here are all implemented using type-indexed data types (also known as non-parametric representations) we follow a different pattern to introduce generic functions in our library.

2.4 Type-indexed data types

A *type-indexed data type* is a data type that is constructed in a generic way from an argument data type. It is a generic technique and we briefly introduce it here adapted for our needs. For a more thorough treatment the reader is referred to [13].

In our example, in the case of parallel arrays, we have to define an array type by induction on the structure of the type of an array element.

Suppose we have a trait `PArray[T]` (to represent parallel arrays) and convenience type synonym `PA[T]` defined as

```
trait PArray[A] // PArray stands for Parallel Array
type PA[A] = PArray[A]
```

For this abstract trait we want to define concrete representations depending on the underlying structure of the type `A` of the array elements. As shown in Fig. 2 we consider a family of types constructed by the limted set of type constructors.

Thus, considering each case in the definition above, we can define a representation transformation function RT (see Fig. 7) that works on types. It was shown [4] how such array representations enable nested parallelism to be implemented in a systematic way.

RT: * → *
RT⟦`PArray[Unit]`⟧ = `UnitArray(len:Int)`
RT⟦`PArray[T]`⟧ = `BaseArray(arr:Array[T])`
`  where T = Int | Float | Boolean`
RT⟦`PArray[(A,B)]`⟧ = `PairArray(a:` RT⟦`PArray[A]`⟧`,b:`RT⟦`PArray[B]`⟧)
RT⟦`PArray[(A|B)]`⟧ = `SumArray(flags:` RT⟦`PArray[Int]`⟧`,`
`  a:` RT⟦`PArray[A]`⟧`,`
`  b:` RT⟦`PArray[B]`⟧`)`
RT⟦`PArray[PArray[A]]`⟧ = `NArray(values:` RT⟦`PArray[A]`⟧`,`
`  segments:` RT⟦`PArray[(Int,Int)]`⟧`)`

Fig. 7. Representation Transformation

Below we show how to use Scala's case classes to represent structure nodes of a concrete representation (`UnitArray`, `BaseArray`, etc.) and how to keep the

data values (data nodes) unboxed in Scala arrays (`Array[A]`). A graphical illustration of these representations is shown in Fig. 8. For details related to these representations we refer to [4].

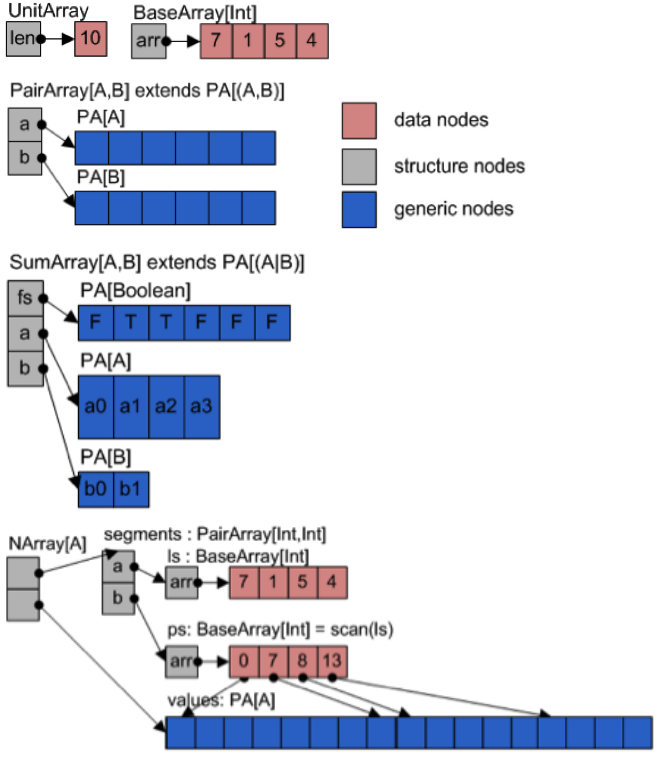


Fig. 8. Type-indexed representations of PArray

Consider as an example a representation of a sparse matrix rendered by applying *RT* function to `Matrix` type. It is shown graphically in Fig. 9.

2.5 Type-indexed arrays in the DSL's implementation

To employ the above techniques in the design of our DSL lets first represent the type structure of an array element type by using the Scala encodings of generic functions described above (see [27] for details).

Note, that in Scala we can equip type representations with generic functions (`replicate` in this sample) by using inheritance. Moreover, we can use a concrete

```
type VectorElem = (Int,Float)
type SparseVector = PArray[VectorElem]
type Matrix = PArray[SparseVector]
```

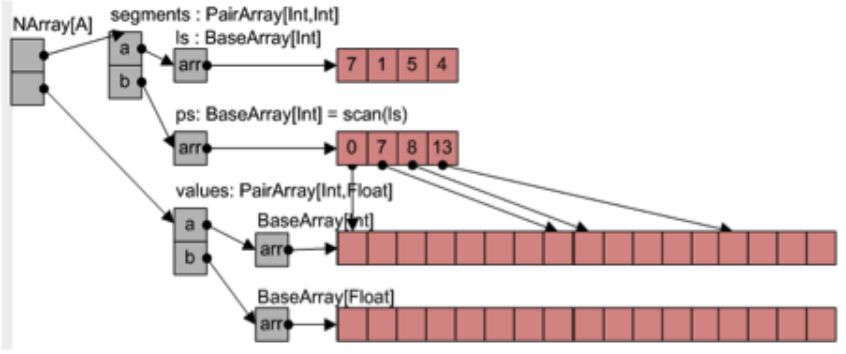


Fig. 9. Sparse matrix representation

array representation (`PairArray`) in the implementation for a particular type case (`pairElement`). All these lead to a fully generic while still statically typed code.

To define generic (polytypic) functions over our arrays we first declare them in the `PArray` trait

```
trait PArray[A] {
  def length: Int
  def map[R:Elem](f: A ⇒ R): PA[R]
  /* and other methods */
}
```

And then we implement these abstract methods in concrete array classes shown in Fig. 11. Note how the implementation changes depending on the type of an array element. Each method declared in the `PArray` trait is a *type indexed function* and each implementation in a concrete array class is an implementation of the function for the particulay type case.

2.6 Lightweight Modular Staging (LMS)

So far, given a type `A` of an array element we know how to build a type-indexed representation of the array using RT function thus yielding $RT[\![\text{PA[A]}]\!]$ type. Next, we have seen how to encode in our DSL these array representations together with polytypic operations over them. These techniques are used in our unstaged implementation of nested data parallelism (as described in [27]).

As it was mentioned before, the unstaged implementation is not intended to be efficient, rather, it should be simple and straightforward, as it is supposed to be used for debugging (in the aforementioned simulation mode). To enable

```
type Elem[A] = Element[A] // type synonim
trait Element[A] { // type descriptor for type A
  def replicate(count: Int, v: A): PA[A]
  def fromArray(arr: Array[A]): PA[A]
}
class BaseElement[T] extends Element[T] {
  def fromArray(arr:Array[T]) = BaseArray(arr)
  def replicate(len:Int, v:T) = BaseArray(Array.fill(len)(v))
}
implicit val unitElem: Elem[Unit] = new UnitElement
implicit val intElem: Elem[Int] = new BaseElement[Int]
implicit val floatElem: Elem[Float] = new BaseElement[Float]
implicit def pairElem[A,B](implicit ea:Elem[A], eb:Elem[B]) =
  new Element[(A,B)] {
    def replicate(count: Int, v: (A,B)) =
        PairArray(ea.replicate(count, v._1),
                  eb.replicate(count, v._2))
  }
```

Fig. 10. Representation of the types of array elements

a parallel and efficient implementation, we employ a *deep* polymorphic embedding technique, namely a particular instance of it known as *Lightweight Modular Staging (LMS)* [25].

In the name, *Lightweight* means that it uses just Scala's type system. *Modular* means that we can choose how to represent intermediate representation (IR) nodes, what optimizations to apply, and which code generators to use at runtime. And *Staging* means that a program instead of executing a value, first, produces other (optimized) program (in a form of a program graph) and then executes that new program to produce the final result.

Consider the method `matrixVectorMul` in Fig. 1 and types `Matrix`, `Vector` that were used in the declaration. In the LMS framework, in order to express staging, we are required to *lift* some types using the type constructor `Rep[_]` and use `Rep[Matrix]`, `Rep[Vector]`, etc. In fact, `sparseVectorMul` should have been declared like this to enable polymorphic embedding

```
def sparseVectorMul(sv: Rep[SparseVector], v: Rep[Vector]): Rep[Float] =
  sum(sv map { case Pair(i,value) ⇒ v(i) * value })
```

In the case of unstaged interpretation we define `Rep` as

```
type Rep[A] = A
```

which yields a unstaged implementation of the method above with the usual evaluation semantics of the host language (i.e. Scala). On the other hand, LMS is a staging framework and we want to build IR instead of just evaluating the method. To achieve this, LMS defines `Rep` as shown in Fig. 12.

This, in effect, enables lifting of the method bodies too, so that its evaluation yields a program graph. Lifting of expressions is performed when the code is

```
type PA[A] = PArray[A] // convenience type synonim
trait PArray[A]
case class UnitArray(len: Int) extends PArray[Unit]{
  def length = len
  def map[R:Elem](f: Unit⇒ R) = element[R].replicate(len, f(()))
}
case class BaseArray[A:Elem](arr: Array[A]) extends PArray[A] {
  def length = arr.length
  def map[R:Elem](f: A ⇒ R) =
    element[R].tabulate(arr.length)(i ⇒ f(arr(i)))
}
case class PairArray[A:Elem,B:Elem](a:PA[A],b:PA[B]) extends PArray[(A,B)]{
  def length = a.length
  def map[R:Elem](f: ((A,B)) ⇒ R) =
    element[R].tabulate(length)(i ⇒ f(a(i),b(i)))
}
case class NArray[A:Elem](values: PA[A], segs: PA[(Int,Int)])
   extends PArray[PArray[A]] {
  def length = segs.length
  def map[R:Elem](f: PA[A] ⇒ R): PA[R] =
    element[R].tabulate(length)(i ⇒ {
      val (p,l)= segs(i); f(values.slice(p,l))
    })
}
```

Fig. 11. Polytypic PArray methods

compiled using the Scala-Virtualized compiler [2]. For example, consider the following lines of code:

```
val x: Rep[Int] = 1
val y = x + 1
```

There is no method `+` defined for `Rep[Int]`, but we can define it on DSL level without providing any concrete implementation as follows

```
trait IntOps extends Base {
  def infix_+(x: Rep[Int], y: Rep[Int]): Rep[Int]
}
```

When such a declaration is in the scope of `x+1` then `+` is replaced by Scala compiler with `infix_+(x, toExp(1))`. In a staging context `infix_+` is defined so that it generates an IR node of the operation

```
trait IntOpsExp extends BaseExp with IntOps {
  case class IntPlus(x:Exp[Int],y:Exp[Int]) extends Def[Int]
  def infix_+(x: Exp[Int], y: Exp[Int]) = IntPlus(x,y)
}
```

```
trait BaseExp extends Base with Expressions {
  type Rep[T] = Exp[T]
}
trait Expressions {
  abstract class Exp[T]
  case class Const[T](x: T) extends Exp[T]
  case class Sym[T](n: Int) extends Exp[T]
  abstract class Def[T] // operations (defined in subtraits)

  class TP[T](val sym: Sym[T], val rhs: Def[T])
  var globalDefs: List[TP[_]] = Nil
  def findDefinition[T](d: Def[T]): TP[T] =
        globalDefs.find(_.rhs == d)
  def findOrCreateDefinition[T](d: Def[T]): TP[T] =
        findDefinition(d).getOrElse{
          createDefinition(fresh[T],d)
        }
  implicit def toExp[T](x: T): Exp[T] = Const(x)
  implicit def toExp[T](d: Def[T]): Exp[T] =
      findOrCreateDefinition(d).sym
}
```

Fig. 12. How Rep[T] is defined in LMS

Here `IntPlus` is an IR node that represents `+` in the program graph. Note that `infix_+` should return `Rep[Int]` while `IntPlus` extends `Def[Int]`, so implicit conversion

```
implicit def toExp[T](d: Def[T]): Exp[T] = findOrCreateDefinition(d).sym
```

which is defined in `Expressions` trait is called here thus providing graph building machinery. We refer to [25] for detailed explanation of how the LMS works.

3 Polytypic Staging

We have shown that for each type A of array element we use the type representation function *RT* to build type-indexed representation of `PArray[A]` type. We also showed how we define PArray's methods using polytypic techniques so that once defined they work for all types in the family. Thus, emphasizing the domain-specific nature of our library and considering its polytypic design we can think of it as a *polytypic DSL*.

If we want to deeply embed our polytypic DSL in Scala by applying polymorphic embedding techniques in general and the LMS framework in particular we need to answer the question: *How are we going to lift the type-indexed types along with the polytypic functions in the Rep world?*. In this section we describe the Polytypic Staging, our approach to a *deep embedding* of polytypic DSLs. By design, our framework:

1. is an extension of the LMS framework
2. respects the type-indexed representations described before
3. adds an additional dimension of flexibility to the LMS framework by making it polytypic
4. behaves as core LMS in the non-polytypic case

3.1 Staged Values

To be consistent with the LMS framework, we do not change the original definition of `Rep`, but we need to make some extensions to account for a polytypic case, they are shown on the following figure in italicized bold.

```
type Rep[T] = Exp[T]
abstract class Exp[+T] {
  def Type: Manifest[T] = manifest[T] // in LMS
  def Elem : Elem[T] // added in Scalan
}
case class Sym[T: Elem](val id: Int) extends Exp[T] {
  override def Elem = element[T]
}
case class Const[+T:Manifest](x: T) extends Def[T]
def element[T] = implicitly[Element[T]]
```

These additions ensure that each staged value has a runtime type descriptor that we use to implement polytypism. Whenever we construct a symbol we have to provide implicitly or explisitly its type descriptor. We also treat constants as definitions (more precisely as operations of arity 0), and we can do it without a loss of generality since given a symbol we can always extract its right-hand-side definition by using the `Def` extractor [22] defined in the core LMS framework.

```
object Def {
  def unapply[T](e: Exp[T]): Option[Def[T]] = e match {
    case s@Sym(_) ⇒ findDefinition(s).map(_.rhs)
    case _ ⇒ None
  }
}
```

Treating constants as definitions in our implementation of LMS means that any lifted value of the type `Rep[T]` is always an instance of `Sym[T]` which simplifies our implementation.

3.2 Staged Type Descriptors

In the staged context the descriptors of types of array elements shown in Fig. 10 remain unchanged. This means that we can keep our type representation schema with one adaptation: we need to *lift* all the methods of the `Element[T]` trait.

Note that even after the lifting of the methods their bodies remain literally the same. This is achieved first, by a systematic use of the `Rep[T]` type constructor in signatures of classes and methods and second, by using the Scala

```
type Elem[A] = Element[A]
trait Element[A] {
  def replicate(count: Rep[Int], v: Rep[A]): PA[A]
  def fromArray(arr: Rep[Array[A]]): PA[A]
}
class BaseElem[T] extends Element[T] {
  def fromArray(arr: Rep[Array[A]]) = BaseArray(arr)
  def replicate(len: Rep[Int], v: Rep[A]) =
          BaseArray(ArrayFill(len, v))
}
implicit val unitElem: Elem[Unit] = new UnitElem
implicit val intElem: Elem[Int] = new BaseElem[Int]
implicit val floatElem:Elem[Float]= new BaseElem[Float]
implicit def pairElem[A,B](implicit ea: Elem[A], eb: Elem[B]) =
  new Element[(A,B)] {
    def replicate(count:Rep[Int], v:Rep[(A,B)]): PA[(A,B)] =
      PairArray(ea.replicate(count, v._1), eb.replicate(count, v._2))
  }
```

Fig. 13. Staged type representations

idiom known as "pimp my library" to add methods that work with values lifted over `Rep[T]`. For example, consider expressions `v._1` and `v._2` in Fig. 13, whose implementation is shown in Fig. 14.

```
def unzipPair[A,B](p: Rep[(A,B)]): (Rep[A],Rep[B]) = p match {
  case Def(Tup(a, b)) ⇒ (a, b)
  case _ ⇒ (First(p), Second(p))
}
class PairOps[A:Elem,B:Elem](p: Rep[(A,B)]) {
  def _1: Rep[A] = { val (a, _) = unzipPair(p); a }
  def _2: Rep[B] = { val (_, b) = unzipPair(p); b }
}
implicit def pimpPair[A:Elem,B:Elem](p: Rep[(A,B)]) = new PairOps(p)
case class Tup[A,B](a: Exp[A], b: Exp[B]) extends Def[(A,B)]
case class First[A,B](pair: Exp[(A,B)]) extends Def[A]
case class Second[A,B](pair: Exp[(A,B)]) extends Def[B]
```

Fig. 14. Staging methods using 'Pimp My Library'

We use the core LMS's `Def` extractor to implement the staging logic. Given a lifted pair (`p: Rep[(A,B)]`) we either successfully extract a `Tup(a,b)` constructor and return the original constituents of the pair, or we emit the new IR nodes thus deferring the tuple deconstruction until later stages. Figures above show

how we implement our polytypic staging framework on top of the core LMS, but as we will see in the next section, to lift type-indexed data type representations of `PArray[A]` over `Rep[_]` and to stage type-indexed (polytypic) array methods we still need to introduce some extensions above the core LMS.

3.3 Staged Type-Indexed Data Types

Polytypism in our DSL is focused around the `PArray[A]` trait (which on the DSL level represents parallel arrays) and every value of the `PArray[A]` type has a type-indexed representation that is built by induction on the structure of A. We also extensively use a convenience type synonym PA defined as follows

```
trait PArray[A]
type PA[A] = Rep[PArray[A]]
```

Thus, in a staged context, `PA` is no longer a synonym of `PArray` and now it is a synonym of a lifted `PArray`. In other words `PA[T]` is a lifted value of array with elements of type T. It is not a key point in our implementation but the introduction of `PA[A]` simplifies our presentation (and in fact greatly simplifies the code of the library).

Let us use the code in Fig. 13 to describe how values of the type `PArray` are staged (or lifted) in our polytypic staging framework. First, notice that the `replicate` method of `pairElem` produces a value of the `PA[(A,B)]` type which is a synonym of `Rep[PArray[(A,B)]]` and so it is a lifted `PArray[(A,B)]` and in LMS such values are represented by symbols of type `Sym[PArray[(A,B)]]`. Thus having a value of type `PA[(A,B)]` we can think of it as a value of some symbol of type `Sym[PArray[(A,B)]]`. Next, recall that in LMS we get lifted values of the type `Rep[T]` by the following implicit conversion (recall also that `Rep[T] = Exp[T]`)

```
implicit def toExp[T](d: Def[T]): Exp[T] = findOrCreateDefinition(d).sym
```

The conversion is automatically inserted by the compiler, it converts any definition to a symbol and builds a program graph as its side effect. We employ this design by deriving all classes that represent parallel arrays from `Def[T]` with appropriate T so that they can be first, converted to symbols and second, added to the graph as array construction nodes. As an example see Fig. 13 where `PairArray` is returned by the method `replicate`. The definitions to represent arrays are shown in Fig. 15.[6]

Compare these classes with those shown in Fig. 11. and note how class signatures became lifted either explicitly by using the `Rep[T]` constructor or implicitly by redefining the `PA[T]` synonym as `Rep[PArray[A]]`. Moreover, the type representation transformation function TR shown in Fig. 7 also remains almost the same, but works with lifted types (see Fig. 16). This similarity is due to the polymorphic embedding design of our approach where we want to give different implementations to the same code.

Note, how we mix-in the `PArray[A]` trait into every graph node of the type `PADef[A]`. In this way, when we stage (or lift over `Rep`) a type-indexed representation of `PArray[T]` we both create the data structure using our concrete array

[6] Please, refer to the source code for the case of SumArray.

```
abstract class PADef[A] extends Def[PArray[A]] with PArray[A]
case class UnitArray(len: Rep[Int]) extends PADef[Unit] {
  def map[R:Elem](f: UnitRep ⇒ Rep[R]): PA[R] =
    element[R].replicate(len, f(toRep(())))
}
case class BaseArray[A:Elem](arr: Rep[Array[T]]) extends PADef[A] {
  def map[B:Elem](f: Rep[A] ⇒ Rep[B]) =
    element[B].tabulate(arr.length)(i ⇒ f(arr(i)))
}
case class PairArray[A:Elem,B:Elem](a:PA[A],b:PA[B]) extends PADef[(A,B)]{
  def map[R:Elem](f: Rep[(A,B)]⇒ Rep[R]): PA[R] = {
    element[R].tabulate(length)(i ⇒ f(a(i),b(i)))
  }
}
case class NArray[A:Elem](arr: PA[A], segments:PA[(Int,Int)])
   extends PADef[PArray[A]] {
  def map[R:Elem](f: PA[A] ⇒ R): PA[R] =
    element[R].tabulate(length)(i ⇒ {
      val Pair(p,l) = segments(i); f(arr.slice(p,l))
    })
}
```

Fig. 15. Array classes as graph nodes (Defs)

classes and at the same time we build nodes of the program graph. This is another key difference from the LMS framework. In the LPS design some nodes of the graph can have a behavior.

The staged representation transformation (SRT) is shown in Fig. 16. The function L is a mapping of types of concrete arrays to the types of staged values.

A graphical illustration of these representations in a form of a program graph is shown in Fig. 17 where we use the following methods that allow us to construct new arrays:

```
def fromArray[T:Elem](x: Rep[Array[T]]): PA[T] =
  element[T].fromArray(x)
def replicate[T:Elem](count: Rep[Int], v: Rep[T]):PA[T]=
  element[T].replicate(count, v)
```

By a staged context (when **type** `Rep[A] = Exp[A]`) it is possible to achieve an effect of constant propagation and a limited form of partial evaluation by applying domain-specific rewritings (see Section 4). Our experiments show that if all the input data of the function is known at staging time, our rewriting method, while simple enough, is able to fully evaluate the function. It is illustrated in Fig. 17 where the array building expressions are evaluated to a type-indexed representation of the resulting arrays and that representation only contains data arrays in Const nodes and concrete array nodes form Fig. 15 that represent `PArray[A]` values.

```
L, SRT: * → *
L⟦UnitArray(len: Rep[Int])⟧ = Rep[PArray[Unit]]
L⟦BaseArray(
    arr:Rep[Array[T]])⟧ = Rep[PArray[T]]
                          where T=Int|Float|Boolean
L⟦PairArray(a:PA[A], b:PA[B])⟧ = Rep[PArray[(A,B)]]
L⟦SumArray(flags:PA[Boolean],
          a:PA[A], b: PA[B])⟧ = Rep[PArray[(A|B)]]
L⟦NArray(
     values:PA[A],
     segs:PA[(Int,Int)])⟧ = Rep[PArray[PArray[A]]]

SRT⟦PArray[Unit]⟧ = UnitArray(len:Rep[Int])
SRT⟦PArrya[T]⟧ = BaseArray(arr:Rep[Array[T]])
                 where T = Int|Float|Boolean
SRT⟦PArray[(A,B)]⟧ = PairArray(a:L⟦SRT⟦PArray[A]⟧⟧,
                               b:L⟦SRT⟦PArray[B]⟧⟧)
SRT⟦PArray[(A|B)]⟧ = SumArray(
          flags: L⟦SRT⟦PArray[Int]⟧⟧,
          a: L⟦SRT⟦PArray[A]⟧⟧,
          b: L⟦SRT⟦PArray[B]⟧⟧)
SRT⟦PArray[PArray[A]]⟧ = NArray(
       values : L⟦SRT⟦PArray[A]⟧⟧,
       segments: L⟦SRT⟦PArray[(Int,Int)]⟧⟧)
```

Fig. 16. Staged Representation Transformation

3.4 Staged Polytypic Functions

The same way as we lift the methods in the type descriptors (types derived from `Element[T]` and shown in Fig. 13) we can lift the methods in the concrete array classes (those derived from `PArray[T]` and shown in Fig. 15).

Compare this code with the non-staged version in Fig. 11 and note how the signatures are all lifted over `Rep` and the bodies of the methods remain literally unchanged. It is interesting that polymorphic embedding allows to share the same code for unstaged and staged implementation even in the library itself which makes the design very flexible.

As a not very trivial example of staging, we show in Fig. 18 a program graph that we get by staging of the function `sparseVectorMul`. The `Lambda(x,exp)` is a representation in the graph of a lambda abstraction where `x` is a symbol that represents the variable and `exp` is a symbol that represent the body of the lambda term.

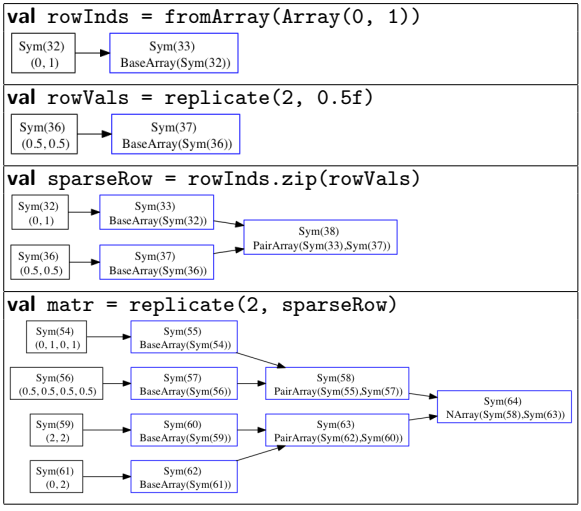


Fig. 17. Array constructors and the resulting graph

4 User-Specific Data Types

4.1 Isomorphic Representations

In this section we describe how to add any user-specific data type to our framework. The key point is to be able to make any such type `U` an instance of typeclass `Elem`. The idea is to define isomorphism between U and some existing instance A:Elem. We extend our family of array element types as it is shown in Fig. 19.

In other words, type U can be regarded as belonging to the type-class `Elem` if there is an isomorphism between U and some `A:Elem`. Type `Iso[A,B]` is defined like this

```
trait Iso[A,U] {
  def eA: Elem[A] // type descriptor for A
  def eU: Elem[U] // type descriptor that is built with this Iso
  def from: U ⇒ A // unstaged morphisms
  def to: A ⇒ U
  def fromStaged: Rep[U] ⇒ Rep[A] // staged morphisms
  def toStaged: Rep[A] ⇒ Rep[U]
}
```

The reason we have separate versions for unstaged and staged isomorphism is that in a staged context we need to have an unstaged version of isos too.

Next, we need to extend the representation transformation for both unstaged (defined in Fig. 7) and staged (defined in Fig. 16) versions. Corresponding extensions are shown in Fig. 20.

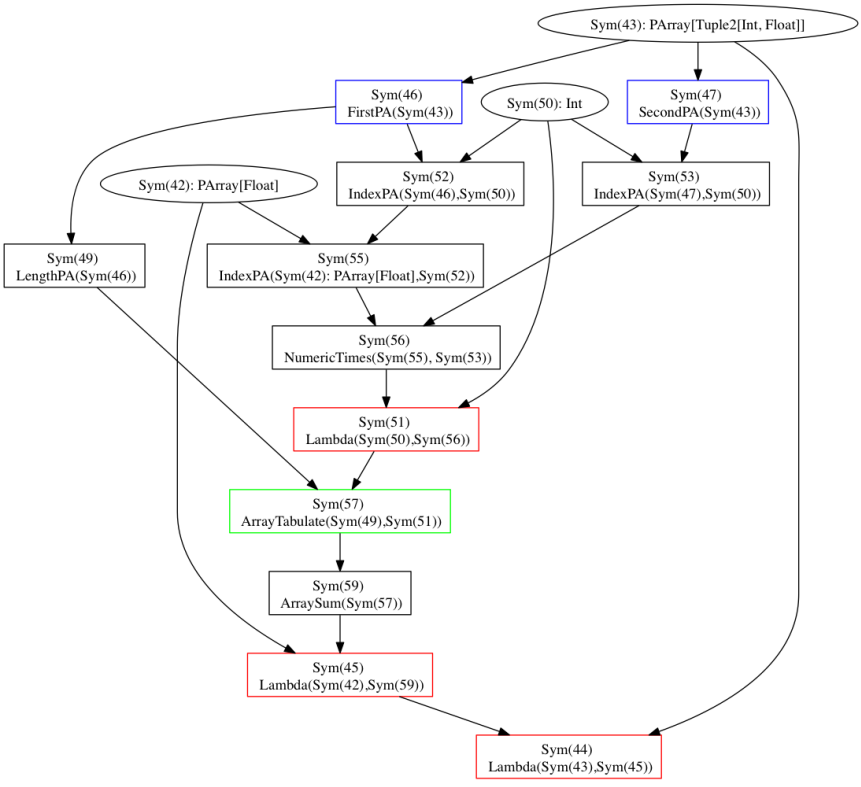


Fig. 18. Program graph for sparceVectorMul

Remember, that for every type A we need a runtime type descriptor `Elem[A]` to be able to create arrays of type `PArray[A]`. For the case of user-specific data type `U` the type descriptor is shown below

```
implicit def viewElement[A,U](implicit iso: Iso[A,U]): Elem[U] =
  new Element[U] {
    def replicate(count: Rep[Int], v:Rep[U]): PA[U] =
      ViewArray(iso.eA.replicate(count, iso.fromStaged(v)), iso)
  }
```

We use the type descriptor of an representation type `iso.eA` to build an actual array and wrap it with `ViewArray` to get type-indexed representation for `PArray[U]` (see Fig. 20). If the descriptor `iso.eA` itself or in some part is a result of `viewElement` (so it is built from user-specific type) then we have nested structure of `ViewArray` wrappers. Isomorphism lifting transformation (see 4.3) is able to

```
A,B = Unit | Int | Float | Boolean // base types
| (A,B) // product (pair of types)
| (A|B) // sum type where (A|B) = Either[A,B]
| PArray[A] // nested array
| U if exist Iso[A,U] for some A : Elem
```

Fig. 19. User-specific type as array element type

RT, L, SRT: $* \to *$
$RT[\![$`PArray[U]`$]\!]$ = `ViewArray(arr:` $RT[\![$`PArray[A]`$]\!]$`, iso: Iso[A,U])`
`if exists unique Iso[A,U] for some A:Elem`

$L[\![$`ViewArray(arr: PA[A], iso:Iso[A,U])`$]\!]$ = `Rep[PArray[U]]`

$SRT[\![$`PArray[U]`$]\!]$ = `ViewArray(arr:` $L[SRT[\![$`PArray[A]`$]\!]]$`, iso: Iso[A,U])`
`if exists unique Iso[A,U] for some A:Elem`

Fig. 20. Representation transformation for user-specific types

eliminate this nesting, by combining corresponding morphisms (`fromStaged` in this case).[7]

To complete our presentation of user-specific types we show an implementation of function `map` below. Notice the usage of the isomorphism in the body of the function.

```
case class ViewArray[A,U](arr: PA[A], iso: Iso[A,U]) extends PArray[U] {
  def map[R:Elem](f: Rep[U] ⇒ Rep[R]): PA[R] = {
    val len = length
    element[R].tabulate(len)(i ⇒ f(iso.toStaged(arr(i))))
  }
}
```

4.2 Samples

Let us see how it works on a simple example. Consider user-specific data types along with their isomorphic representations shown in Fig. 21. Given that definitions we can build for example an array of circles by applying one of the constructor functions. (see Fig. 22)

4.3 Isomorphism lifting transformation

Given function $f : U_1 \to U_2$ between two user-specific types, the isomorphims lifting is a rewrite-based transformation that, when applied at graph generation

[7] We claim, but have not yet proved this.

```
case class ExpPoint(x: Rep[Int], y: Rep[Int]) extends Def[Point]
object ExpPoint {
  class IsoExpPoint extends Point.IsoPoint {
    override def fromStaged = (p: Rep[Point]) ⇒ (p.x, p.y)
    override def toStaged = (p: Rep[(Int, Int)]) ⇒ ExpPoint(p._1, p._2)
  }
}
case class ExpCircle(loc: Rep[Point], rad:Rep[Int]) extends Def[Circle]
object ExpCircle {
  class IsoExpCircle extends Circle.IsoCircle {
    override def fromStaged = (x: Rep[Circle]) ⇒ (x.loc, x.rad)
    override def toStaged = (x: Rep[(Point, Int)]) ⇒
ExpCircle(x._1, x._2)
  }
}
```

Fig. 21. Sample user-specific data types

```
val circles = replicate(2, Circle(Point(10, 20), 30))
```

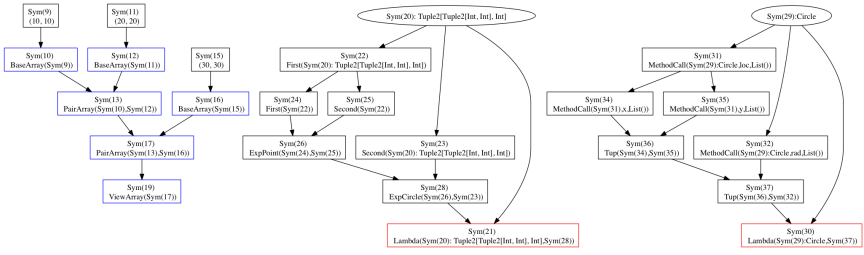


Fig. 22. Isomorphisms lifted out from domain code

stage, transforms the function f in the composition $to_{U2} \circ f^0 \circ from_{U1}$, where $f^0 : U_1^0 \to U_2^0$ and U_1^0, U_2^0 - canonical isomorphic representations of the types U_1 and U_2 respectively.

To perform this transformation we need to combine isos in different ways. Below we show a series of functions that build isos from isos.

For each instance `A:Elem` we have identity isomorphism

```
def identityIso[A:Elem]: Iso[A, A] = new Iso[A,A] {
  def from = (x: A) ⇒ x
  def to = (x: A) ⇒ x
  def fromStaged = (x: Rep[A]) ⇒ x
  def toStaged = (x: Rep[A]) ⇒ x
}
```

For each iso we can build its nested version

```
def nestIso[A,B](iso: Iso[A,B]) = new Iso[PArray[A], PArray[B]] {
```

```
  def from = (bs: PArray[B]) ⇒ bs map iso.from
  def to = (as: PArray[A]) ⇒ as map iso.to
  def fromStaged = (bs: Rep[PArray[B]]) ⇒ bs map iso.fromStaged
  def toStaged = (as: Rep[PArray[A]]) ⇒ as map iso.toStaged
}
```

Given a pair of isomorphisms we can build their product

```
def pairIso[A1,B1,A2,B2](iso1: Iso[A1,B1], iso2: Iso[A2,B2]) =
  new Iso[(A1, A2), (B1,B2)] {
    def from = (b: (B1,B2)) ⇒ (iso1.from(b._1), iso2.from(b._2))
    def to = (a: (A1, A2)) ⇒ (iso1.to(a._1), iso2.to(a._2))
    def fromStaged = (b: Rep[(B1,B2)]) ⇒
      (iso1.fromStaged(b._1), iso2.fromStaged(b._2))
    def toStaged = (a: Rep[(A1, A2)]) ⇒
      (iso1.toStaged(a._1), iso2.toStaged(a._2))
  }
```

And we also can compose

```
def composeIso[A,B,C](iso2:Iso[B,C], iso1:Iso[A,B]) = new StagedIso[A,C]{
  def from = (c: C) ⇒ iso1.from(iso2.from(c))
  def to = (a: A) ⇒ iso2.to(iso1.to(a))
  def fromStaged = (c: Rep[C]) ⇒ iso1.fromStaged(iso2.fromStaged(c))
  def toStaged = (a: Rep[A]) ⇒ iso2.toStaged(iso1.toStaged(a))
}
```

Note that our staging framework is flexible enough so that we can build fully generic isomorphism combinators on top of the existing polytypic framework both for the unstaged and staged versions.

Given iso combinators we can use them to perform isomorphism lifting in our polytypic staging framework. Since our staging framework is based on LMS we can use its simple but powerfull enough rewriting method to implement required transformations on the fly.

One of the benefits that we can get out of deep embedding is the ability to perform domain-specific optimizations. For instance we can use the staging time rewrites. Our method of rewriting is very simple and is based on the one proposed in [25].

The method is based on the fact that every staged operation, which is represented by a graph node, in terms of the Scala language is represented by descendants of the `Def` class. Every time a new definition is created it is converted to the corresponding `Exp` by the special function shown in Fig. 23

The rewriting works using the following algorithm. If we can find the definition in the graph, we just return its symbol. Otherwise, we try to rewrite the Def. If the result of `rewrite` is not defined then there is no rules that can be applied so the definition is added to the graph. If the `rewrite` comes back with a new symbol then we extract its definition from the graph (by using `Def`) and go recursively with the new definition.

```
implicit def toExp[T:Elem](d: Def[T]): Exp[T] = findDefinition(d) match {
  case Some(TP(s, _)) ⇒ s
  case None ⇒
    var ns = rewrite(d)
    ns match {
      case null ⇒
        val TP(res, _) = createDefinition(fresh[T], d)
        res
      case _ ⇒ ns match {
        case Var(_) ⇒ ns
        case Def(newD) ⇒ toExp(newD)
      }
    }
}
```

Fig. 23. Graph building and rewriting algorithm

Rewriting rules that perform isomorphism lifting are shown in Fig. 24.[8]

5 Conclusions and Related Work

In a traditional multi-stage programming the original program should be rewritten in a special *quotation* syntax to get a staged version where computation and code generation is mixed and expressed explicitly by the programmer. In this approach the compiler is able to statically ensure that the generated code is type-safe. Moreover, the staged program is equivalent to original even though it is partially evaluated at compile time. This compile time guarantees are the most noticable advantages of multi-staged programming when compared with our technique. On the other hand, the requirement to rewrite the original program in the quotation syntax can be difficult to a non-experienced programmer.

In the lightweight staging approach, which is based on polymorphic embedding, the staging itself is regarded as just a special interpretation of the domain semantics in addition to the usual interpretation as evaluation. The same code is interpreted in two different ways, so there is no need for rewriting to get a staged version. The syntactic overhead of staging is minimal is this case. What can be considered as disadvantage of the lightweight approach is that there is no guarantees of correctness that come from the staging framework itself. It is the responsibility of the author of the DSL to provide such a guarantees. It is an interesting direction of further research to give a general characterisation of correctness in lightweight staging context.

At the same time, lightweight staging based on polymorphic embedding by its definition allows us to implement *debugging by simulation*. Given two equivalent

[8] We only show the rules that demonstrate the usage of the iso combinators. Other rules can be found in source code.

```
override def rewrite[T:Elem](d: Def[T]) = d match {
  case ViewArray(Def(ViewArray(arr, iso1)), iso2) ⇒ {
    val compIso = composeIso(iso2, iso1); ViewArray(arr, compIso)
  }
  case NArray(Def(view@ViewArray(a, iso)), segs) ⇒ {
    val nested = NArray(a, segs)
    ViewArray(nested, nestIso(iso))
  }
  case PairArray(Def(v1@ViewArray(arr1, iso1)),
                 Def(v2@ViewArray(arr2, iso2))) ⇒ {
    val pIso = pairIso(iso1, iso2)
    val arr = PairArray(arr1, arr2)
    ViewArray(arr, pIso)
  }
  case PairArray(Def(v1@ViewArray(arr1, iso1)), arr2) ⇒ {
    val iso2 = identityIso
    val pIso = pairIso(iso1, iso2)
    val arr = PairArray(arr1, arr2)
    ViewArray(arr, pIso)
  }
  case PairArray(arr2, Def(v1@ViewArray(arr1, iso1))) ⇒ {
    val iso2 = identityIso
    val pIso = pairIso(iso2, iso1)
    ViewArray(arr, pIso)
  }
  case _ ⇒ super.rewrite(d)
}
```

Fig. 24. Isomorphism lifting rules

interpretations of the DSL, one - evaluation (which is simple), and another - staged code generation (which can be quite involved), we can debug the program in *simulation mode* using evaluation and then by applying staged interpretation to the same code we can produce executable code to run with real data.

Our experience with embedding of the DSL for nested data parallelism (which is a polytypic DSL) shows that our approach is

1. practical - allows for creation of high level expressive DSLs where staging is almost transparent to the user
2. flexible - can be extended in various ways using power of the host language Scala and user-specific data types
3. lightweight for the user - based on library approach rather than on host language extension

Isomorphic representations or *view types* were proposed for Generic Haskell. Our lifting transformation corresponds (in spirit) to bimap function described in [15]. We have not proved it yet but there are reasons to believe that we will be able to fully implement the lifting transformation as a set or rewrite rules.

Clear separation of a domain code and isomorphisms in an intermediate representation (IR graph) can be usefull for analisys and transformation as they belong to the different domains with different algebraic properties.

In this paper we have described a new staging technique that can be used to develop embedded DSLs for different *polytypic* domains, - the domains that admit specification and formalization in terms of generic (polytypic) programming. To our best knowledge this the first attempt to state this problem explicitly.

Staging approach as it is described here is a front-end of the compiler toolchain. In polytypic context it opens up many questions both for research and software engineering. That is also true for rewriting rules. Our experiments with rewritings in NDP domain show that even simple rewriting strategy combined with domain knowledge can exhibit radical optimizations not possible in the context of general purpose language. We regard this questions as directions of the future research.

Acknowledgments
The author expresses his gratitude to Sergei Romanenko, Andrei Klimov and other participants of Refal seminar at Keldysh Institute for numerous useful comments and fruitful discussions of this work.

References

1. Eclipse. `http://eclipse.org/`.
2. Philipp Haller Adriaan Moors, Tiark Rompf and Martin Odersky. Tool Demo: Scala-Virtualized, 2011.
3. Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, 1990.
4. Manuel M. T. Chakravarty and Gabriele Keller. An Approach to Fast Arrays in Haskell, 2002.
5. Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: a status report. In *In DAMP 2007: Workshop on Declarative Aspects of Multicore Programming.* ACM Press, 2007.
6. James Cheney and Ralf Hinze. Phantom types, 2003.
7. Bruno C. d. S. Oliveira, Adriaan Moors, and Martin Odersky. Type Classes as Objects and Implicits. In *n Proceedings of the 25th ACM International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH/OOPSLA)*, October 2010.
8. Jeffrey Dean, Sanjay Ghemawat, and Google Inc. MapReduce: simplified data processing on large clusters. In *In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation.* USENIX Association, 2004.
9. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. MPI: The Complete Reference (Vol. 2). Technical report, The MIT Press, 1998.
10. Ralf Hinze. A new approach to generic functional programming. In *In The 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132. ACM Press, 1999.

11. Ralf Hinze. Fun with phantom types, 2003.
12. Ralf Hinze. Generics for the masses. *SIGPLAN Not.*, 39:236–243, September 2004.
13. Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In *SCIENCE OF COMPUTER PROGRAMMING*, pages 148–174, 2004.
14. Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 137–148, New York, NY, USA, 2008. ACM.
15. Stefan Holdermans, Johan Jeuring, Andres Löh, and Alexey Rodriguez. Generic views on data types. In *In Tarmo Uustalu, editor, Proceedings 8th International Conference on Mathematics of Program Construction, MPC'06, volume 4014 of LNCS*, pages 209–234. Springer-Verlag, 2006.
16. Paul Hudak. Building domain-specific embedded languages. *ACM COMPUTING SURVEYS*, 28, 1996.
17. Patrik Jansson. Polytypic programming. In *2nd Int. School on Advanced Functional Programming*, pages 68–114. Springer-Verlag, 1996.
18. Simon L. Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In *ICFP*, pages 50–61, 2006.
19. Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell, 2008.
20. Gabriele Keller and Manuel M.T. Chakravarty. Flattening Trees, 1998.
21. NVIDIA. NVIDIA CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`, 2011.
22. Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, Second Edition*. Artima, 2010.
23. Bruno C.d.S. Oliveira and Jeremy Gibbons. Scala for generic programmers. In *Proceedings of the ACM SIGPLAN workshop on Generic programming*, WGP '08, pages 25–36, New York, NY, USA, 2008. ACM.
24. Aleksandar Prokopec, Tiark Rompf, Phil Bagwell, and Martin Odersky. A generic parallel collection framework, 2010.
25. Tiark Rompf and Martin Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.
26. Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-blocks for performance oriented dsls. In *DSL*, pages 93–117, 2011.
27. Alexander Slesarenko. Scalan: polytypic library for nested parallelism in Scala. Preprint 22, Keldysh Institute of Applied Mathematics, 2011.
28. Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In Lise Getoor and Tobias Scheffer, editors, *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, pages 609–616, New York, NY, USA, June 2011. ACM.
29. Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50, 2003.

Author Index

Научное издание

Труды конференции

Сборник трудов Третьего международного семинара по метавычислениям имени В. Ф. Турчина, г. Переславль-Залесский, 5–9 июля 2012 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. **16,2**. Усл. печ. л. **13,3**. Подписано к печати **15.06.2012**. Ответственный за выпуск: *С. М. Абрамов*.

Издательство Университет города Переславля

Отпечатано в ООО ≪Регион≫. Печать цифровая. Бумага офсетная. Тираж 150 экз. Заказ 14.
152025 Ярославская область, г. Переславль-Залесский, ул. Строителей, д. 41