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:

$$\frac{precondition}{g \rightarrow g'}$$

Let g denote a current graph and g' a graph produced by a single step of rewriting. The rewriting step $g \rightarrow g'$ 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:

$$f(x)|_{\{x<5\}} \to let\ y = x|_{\{x<5\}}\ in\ f(y)|_{\{y<6\}} \to let\ z = y|_{\{y<6\}}\ in\ f(z)|_{\{z<7\}} \to \ldots$$

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

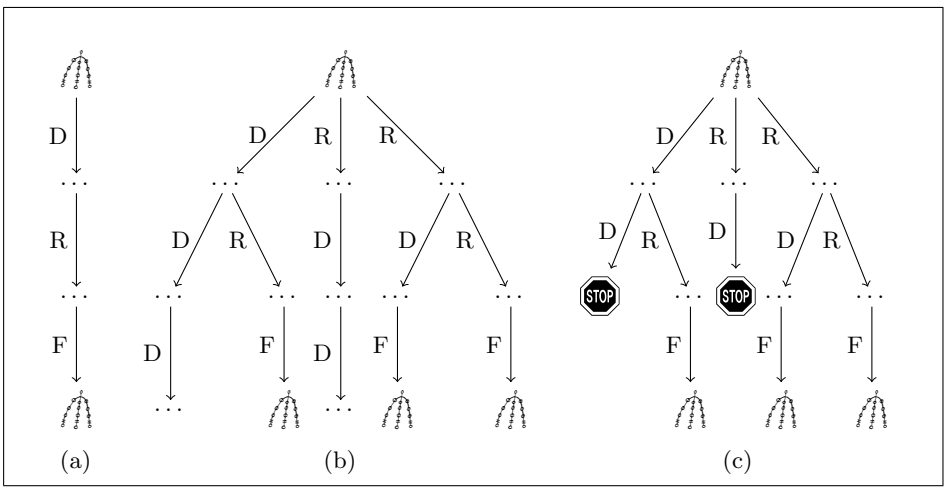

Fig. 3: Trees of graphs. (a) Deterministic algorithm, (b) Transformation relation, (c) Multi-result supercompilation.

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.

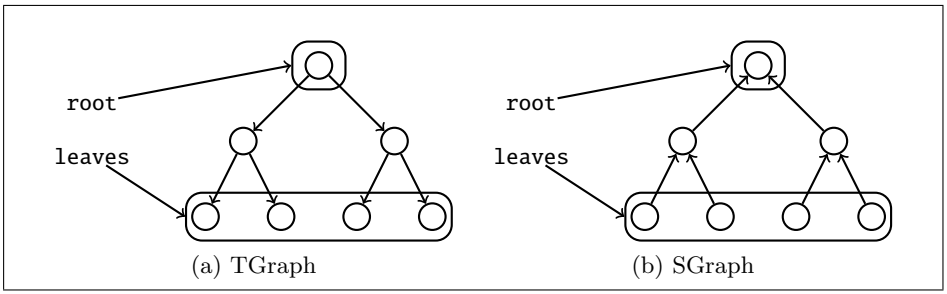

Fig. 5: MRSC data structures

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.

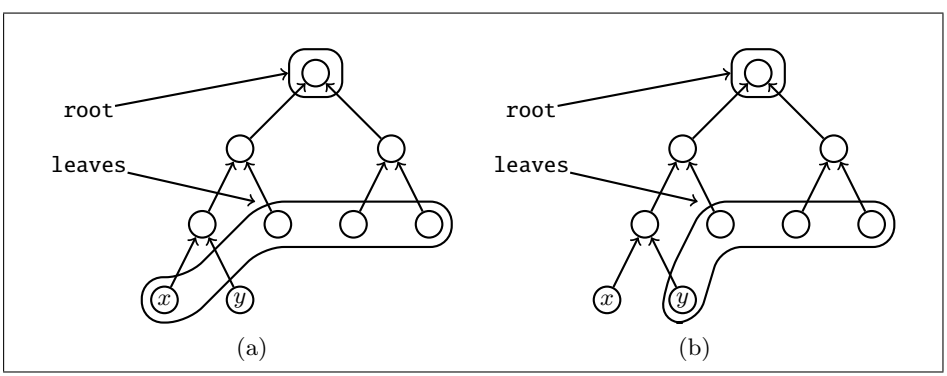

Fig. 6: Reuse of nodes in S-graphs

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.

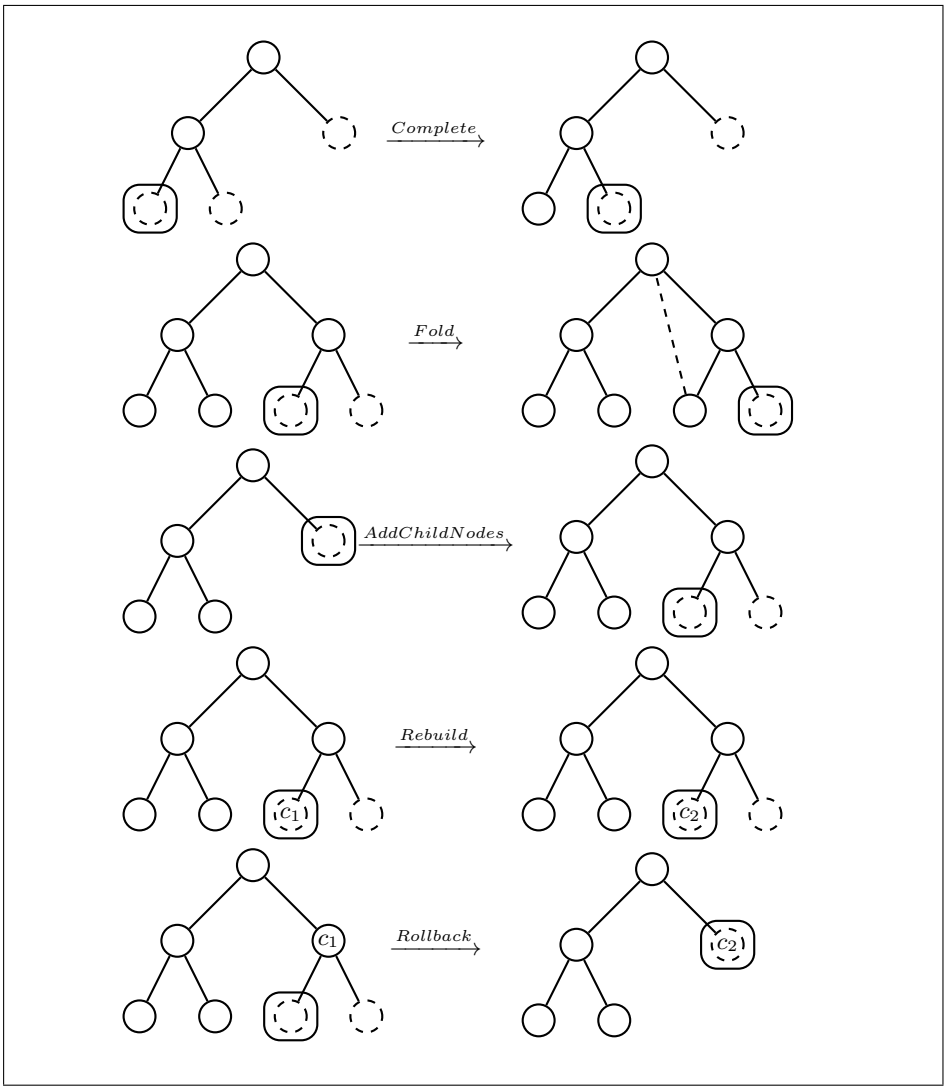

Fig. 7: Basis of operations on graphs schematically

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.