Accepted for Ershov Informatics Conference 2011
Pre-proceedings version

Multi-Result Supercompilation as Branching Growth of the Penultimate Level in Metasystem Transitions*

Ilya Klyuchnikov and Sergei Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Abstract. The paper deals with some aspects of metasystem transitions in the context of supercompilation. We consider the manifestations of the law of *branching growth of the penultimate level* in the case of higher-level supercompilation and argue that this law provides some useful hints regarding the ways of constructing metasystems by combining supercompilers. In particular we show the usefulness of multi-result supercompilation for proving the equivalence of expressions and in two-level supercompilation.

1 Introduction

A *supercompiler* is a source-to-source program transformer SC based on *supercompilation* techniques [27,28,30], which, given an input program p, generates a *residual program* $p' = SC[\![p]\!]$. Supercompilation is often seen as a method of program optimization but also may be used for program analysis. Since our area of research is program analysis by supercompilation, we assume all the supercompilers we deal with to strictly preserve the semantics of programs.

The general concept of *metasystem transition* was put forward by V.F. Turchin in 1970-s [26, Chapter 3, Section "The Metasystem Transition"]:

> We shall call the system made up of control subsystem X and the many homogeneous subsystems A_1, A_2, A_3 ... controlled by it a metasystem in relation to systems A_1, A_2, A_3 ... Therefore we shall call the transition from one stage to the next the *metasystem transition.*

In particular, by treating supercompilation as an elementary operation, we can use supercompilers as building blocks for constructing more complex systems [29,30], which may be considered as an instance of metasystem transition.

There is no generally accepted term for systems built from supercompilers by metasystem transitions. Thus, for lack of something better, we use the term "higher-level supercompilation"[1].

* Supported by Russian Foundation for Basic Research project No. 09-01-00834-a.

[1] A possible alternative could be "meta-supercompilation" or "metacomputation" (suggested by V.F. Turchin [30]).

Higher-level supercompilation is a new area of research whose potential is not fully realized, some aspects being still poorly understood. The *law of branching growth of the penultimate level* is one of such aspects. Of course, laws are called "laws", because they manifest itself regardless of whether we are aware of them or not. However, if a law is well understood, its intentional use often enables us to find more elegant and (conceptually) simple solutions, as compared to those obtained by blind trial and error.

In the subsequent sections we consider how the law of branching growth of the penultimate level manifests itself in higher-level supercompilation and how it can be used when constructing metasystems composed of a number of supercompilers.

2 Diversity of Higher-Level Supercompilation

In the following, higher-level supercompilation is assumed to mean the construction of systems that use SC, the operation of supercompilation, as a primitive. Here are some examples of higher-level supercompilation:

- Self-application of a supercompiler [21], or, more generally, an application of supercompilers to supercompilers [3] (also known as Futamura projections [2]). In this case an instance of a supercompiler, essentially, controls the execution of another instance of the same supercompiler.
- Proving the equivalence of expressions by supercompilation [18,16]. In this case supercompilation is used as a means of normalizing programs for further check for syntactic isomorphism.
- Two-level supercompilation [17], in which the "upper" supercompiler uses improvement lemmas proved by the "lower" supercompiler.
- Distillation [4,6,5], which involves comparing recursive (supercompiled) representations of configurations, instead of configurations (although the separation of the ground level and the metalevel is not as clear as in the case of two-level supercompilation).

This list, which is certainly incomplete, shows that constructing metasystems by combining supercompilers may be done in a variety of ways. However, until recently, the idea of *self-application* (in the form of Futamura projections) has enjoyed the most popularity. The point is that this form is not the only possible form of higher-level supercompilation[2].

3 Branching Growth of the Penultimate Level in Metasystem Transitions

What is the concrete mechanics of a metasystem transition? Some light on this is shed in "The phenomenon of science" by V.F. Turchin [26, Chapter 3, Section "The Metasystem Transition"]:

[2] Note, that the emergence of various forms of higher-level supercompilation can also be seen as the growth of the penultimate level leading to a metasystem transition.

> In general we must note that the integration of subsystems is by no means the end of their evolutionary development. We must not imagine that systems A_1, A_2, A_3, ... are reproduced in large numbers after which the control device X suddenly arises "above them". On the contrary, the rudiments of the control system form when the number of subsystems A_i is still quite small. As we saw above, this is the only way the trial and error method can operate. But after control subsystem X has formed, there is a massive replication of subsystems A_i and during this process both A_i and X are refined. The appearance of the structure for control of subsystems A_i does not conclude rapid growth in the number of subsystems A_i; rather, it precedes and causes this growth because it makes multiplication of A_i useful to the organism. The carrier of a definite level of organization branches out only after the new, higher level begins to form. This characteristic can be called the law of *branching growth of the penultimate level*. In the phenomenological functional description, therefore, the metasystem transition does not appear immediately after the establishment of a new level; it appears somewhat later, after the penultimate level has branched out.

This can be summed up by the "formula":

$$\textit{Control} + \textit{Branching Growth} \Longrightarrow \textit{Metasystem Transition} \quad (1)$$

In some cases of higher-level supercompilation, such as self-application [2,21] and the simplest two-level supercompilation [16,17], there is a single (and fixed) supercompiler under control. However, this straightforward approach is unable to take advantage of the real potential of metasystem transition, because any real control implies the possibility of choice, which does not exist without some variety and/or multiplicity at the lower level.

A multiplicity of choice may exist *in time* (if there is a single unit at the lower level, whose state may be controlled or modified) or *in space* (if there is a number of controllable units). In general, there may be a combination of multiplicity in time and in space. However, in computer science, the division into space and time may be rather arbitrary. For example, we may run a program several times with different options, in which case the variants will be separated "in time". But, given a supercomputer, we may run the program with different options on multiple nodes, in which case the variants will be separated "in space".

The emergence of multiplicity of choice brings about the need for improvements and refinements in the control system, and often leads to a metasystem transition, which was formulated by V.F. Turchin as the law of branching growth of the penultimate level.

The next section considers some forms of branching growth in the context of supercompilation.

4 Multi-Result Supercompilation and Program Analysis

At a high level of abstraction, supercompilation can be seen as a *transformation relation SC* [8,11], such that, given two programs p and p', $p\ SC\ p'$ means that p' is a residual program with respect to the source program p.

From this viewpoint, there may exist several residual programs for a given input program, and we may construct a supercompiler producing a set of residual programs, rather than a single program. In such cases we will speak about *multi-result supercompilation*, to distinguish it from traditional (single-result) supercompilation.

Until recently, multi-result supercompilation was not paid sufficient attention. This situation was probably because supercompilation was primarily considered as a tool for program *optimization*. And, when seen as an optimizer, a supercompiler is usually expected to produce a single program $p' = SC[\![p]\!]$.

Indeed, the word "optimization" implies that we are interested in obtaining the best result[3]. Hence, an optimizing supercompiler is expected to return a single result, "the best" one, even if it could produce several residual programs. The suboptimal results are not exposed explicitly – there is some logic under the hood that chooses "the best" variant [4].

Optimizing supercompilers usually are implemented in the form of deterministic algorithms. If a deterministic supercompiler faces a choice, it only considers a single variant, which it believes to result into "the best" residual program.

In the case of optimization, the concepts of "better" and "worse" are relatively easy to formalize. A program p' is usually assumed to be better than p if it is faster and/or smaller. The execution speed and code size are measurable and can be expressed numerically. So the criteria used by optimizing supercompiler in order to choose "the best" (or at least not the worst) variant, when non-determinism appears during supercompilation, are formalizable.

But, if supercompilation is used for program analysis by transformation, things get more complicated. In this case a residual program p' is better than p'' if p' is easier to analyze than p''. This criteria is less formalizable than the notions of execution speed and code size. Besides, some choices that appeared to be obvious or natural in the case of an optimizing supercompiler may not be good if residual programs are meant for subsequent analysis, rather than execution. Moreover, the influence of the choices made during supercompilation on the "analyzability" of residual programs are rather difficult to foresee.

On the other hand, the results of transformation are much easier to estimate post factum. So multi-result supercompilation seems to be a natural solution for program analysis, since the results of choices made during supercompilation can be evaluated by examining the final results.

It is interesting that some choices which look unnatural during supercompilation may lead to interesting and useful final results (see Section 5.4).

[3] In Latin "optimus" means "the best".

[4] Note that multiplicity is internally used by some optimizers [25]

5 The Synergy of Two-Level and Multi-Result Supercompilation

In the context of supercompilation the formula from Section 3

Control + Branching Growth ⇒ Metasystem Transition

can be instantiated in the following way:

Two-Level Supercompilation + Multi-Result Supercompilation ⇒ Metasystem Transition

In other words: combining two-level supercompilation (control) with multi-result supercompilation (growth of the penultimate level) leads to a metasystem transition, thereby increasing the power of two-level supercompilation.

The evolution history of HOSC [9] – a supercompiler for a subset of Haskell – seems to be a good illustration for the quotation in Section 3. The initial goal of the project was to study the applicability of supercompilation to program analysis by transformation, the previous studies in the field of optimizing supercompilation serving as the starting point. The stages the project HOSC has passed through are considered in the following subsections.

5.1 Stage 0. Proving the Equivalence of Expressions: Rudiments of Control

The development of HOSC started with the simple motto: transformed programs are to be analyzed, rather than executed. So we made a few minor modifications, which could be unacceptable in the context of program optimization, but are quite natural in the case of program analysis:

1. A supercompiler is allowed to duplicate code.
2. A supercompiler is allowed to generate a non-modular flat program.

These modifications enabled the internals of the supercompiler to be considerably simplified. In addition, it was found that these simplifications enhanced the ability of the supercompiler to transform equivalent programs to the same syntactic form. That is, the supercompiler HOSC proved to be more powerful at "normalizing" programs, than optimizing compilers.

The mechanism of program normalization can be metaphorically explained in this way: it is easier to transform two programs to the same bad form, than to the same good form (where "goodness" is measured in terms of size and execution speed).

The details of this approach are described in [16]. HOSC turned out to be quite good at normalizing modular programs with heavy use of higher-order functions (especially, various forms of combinators).

Let us consider a quite spectacular example. A Haskell program defining fragments of Peano and Church arithmetics is shown in Fig. 1. A conjecture

```
data Peano = Z | S Peano;
type Church x = (x → x) → (x → x);

foldn :: (t → t) → t → Peano → t;
foldn = λs z n → case n of { Z → z; S n1 → s (foldn s z n1); };

add :: Peano → Peano → Peano;
add = λx y → foldn S y x;

mult :: Peano → Peano → Peano;
mult = λx y → foldn (add y) Z x;

mult' :: Church x → Church x → Church x;
mult' = λm n f → m (n f);

peano2church :: Peano → Church x;
peano2church = λp → foldn (λm f x → f (m f x)) (λf x → x) p;

church2peano :: Church Peano → Peano;
church2peano = λn → n S Z;
```

Fig. 1. Fragments of Peano and Church arithmetics: definitions in Haskell

```
∀x y. mult x y ≅ church2peano (mult' (peano2church x) (peano2church y))
```

Fig. 2. A conjecture about the equivalence of expressions

```
letrec f = λm n → case m of {
  Z → Z;
  S p → letrec g = λz → case z of { S v → S (g v); Z → f p n; } in g n;
} in f x y
```

Fig. 3. Proof by supercompilation: both parts of the conjecture are transformed to the same residual expression (shown in the Figure)

in Fig. 2 states the operational equivalence of two expressions with universally quantified variables. HOSC transforms both expressions to the same residual expression depicted in Fig. 3 and thus deduces that the conjecture holds.

Certainly, launching two instances of the same supercompiler, followed by comparing the results, can be regarded as a form of control, but as a very primitive one.

5.2 Stage 1. Two-Level Supercompilation: Shaping of Control

Simply put, supercompilation is based on the following simple operations on *configurations* (expressions with free variables):

```
data Bool = True | False;
data Peano = Z | S Peano;
even = λx → case x of { Z → True; S x1 → odd x1; };
odd = λx → case x of { Z → False; S x1 → even x1; };
or = λx y → case x of { True → True; False → y; };
```

Fig. 4. Numeric operations

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

Fig. 5. `or (even m) (odd m)`: the result of two-level supercompilation

1. Driving: rewriting a configuration into an *equivalent* one, in order to *simplify* it.
2. Case analysis: splitting a configuration into a finite number of subconfigurations.
3. Folding: comparing two configurations for syntactic isomorphism, in order to reduce the current configuration to a previously encountered one.
4. Generalization: replacing a configuration with a more general one, in order to guarantee termination.

Generalization often leads to redundancy in residual programs, so that supercompilers should try to avoid it.

The main idea of two-level supercompilation [17,12] consists in avoiding generalization as much as possible. When a two-level supercompiler encounters a configuration A that has to be generalized according to the rules of ordinary supercompilation, it tries to replace A with an equivalent configuration B that does not have to be generalized. The equivalence of A and B is proven by invoking two instances of a lower-level supercompiler.

For example, let us consider the expression `or (even m) (odd m)` in the context of the program shown in Fig. 4. The single-level HOSC is unable to transform this expression into a program that is certain not to return `False`. During supercompilation the following expressions are checked for syntactic isomorphism:

1. `case (even m) of { True → True; False → odd m; }`
2. `case (even n) of { True → True; False → odd (S (S n)); }`

Since these expressions are not syntactically isomorphic, the single-level HOSC has to perform a generalization. However, the two-level HOSC is able to prove (by calling the single-level HOSC twice) that the following configuration (3)

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

is equivalent to the configuration (2). Now the configuration (3) is syntactically isomorphic to the configuration (1). Hence, we can fold (3) to (1), thereby avoiding generalization. The corresponding residual program is shown in Fig. 5. This program is certain not to return `False` (just because `False` does not appear in the program).

5.3 Stage 2. Multi-Result Supercompilation: Branching Growth of the Penultimate Level

The two-level supercompiler described in [17] calls an instance of itself [5]. Moreover, there exists a "recipe" of turning *some* classical single-level supercompilers into two-level ones. This procedure can be schematically represented by the following formula (which may seem a bit obscure for now, but it will be explained in Section 6 in detail):

$$L2(Sc) = Sc_{mod}(Sc)$$

where Sc_{mod} is a modification of a classical supercompiler Sc. Thus a modified instance of a classical supercompiler uses an unmodified instance of the *same* supercompiler.

In the paper [12] this formula is generalized as follows:

$$L2'(Sc', Sc'') = Sc'_{mod}(Sc'')$$

The point is that a two-level supercompiler can be produced from two *different* supercompilers[6]. And $L2(Sc) = L2'(Sc, Sc)$ is just a special case, where $Sc' = Sc''$.

The fruitfulness of this generalization is illustrated in [12] by the following example. Let us express two BNF-grammars by means of combinators:

```
doubleA1 = ϵ | A doubleA1 A
doubleA2 = ϵ | A A doubleA2
```

Although these grammars are equivalent, the corresponding parsers are different, the complexity of the first parser being $O(n^2)$, while the complexity of the second one being $O(n)$. The paper [12] shows that there is no two-level supercompiler $L2(Sc_i)$ produced from supercompilers $\overline{Sc_i}$ from [10] that can transform the first grammar into the second one. The problem is that each time when the upper supercompiler makes a conjecture (about the equivalence of expressions), the lower supercompiler is unable to prove this conjecture.

However, we can combine supercompilers described in [10] by means of the formula $L2'(Sc_i, Sc_j)$, in which case it is possible to find two supercompilers Sc_i and Sc_j, such that $L2'(Sc_i, Sc_j)$ transforms the first parser into the second one[7].

[5] So, it can be regarded as a special case of self-application.

[6] In the context of Futamura projections, the idea of combining *different* versions of a partial evaluator was considered by R. Glück [3].

[7] As was shown by Sørensen [24], a classical single-level supercompiler for a lazy functional language can not improve the runtime complexity of a program.

```
data List a = Nil | Cons a (List a);

app = λxs ys →
  case xs of { Nil → ys; Cons z zs → Cons z (app zs ys); };
rev = λxs →
  case xs of { Nil → Nil; Cons y ys → app (rev ys) (Cons y Nil); };
```

Fig. 6. Naive list reversal

```
letrec f = λx y → case x of { Nil → y; Cons v w → f w (Cons v y); }
in f xs Nil
```

Fig. 7. `rev xs`: a result of multi-result two-level supercompilation

In a sense, $L2$ and $L2'$ can be regarded as devices for breeding and multiplying supercompilers, $L2'$ being more "productive". Indeed, for 8 different single-level supercompilers described in [10], the formula $L2$ may produce only 8 different two-level supercompilers, while $L2'$ may produce 64 different supercompilers.

Now suppose that Sc'' in $L2'$ is a multi-result supercompiler. Again, we get one single-result two-level supercompiler. Applying $L2'$ to 8 different single-level supercompilers from [10], we get 8 different single-result two-level supercompilers.

And finally, combining these two-level supercompilers we can get one multi-result two-level supercompiler.

As was shown in [12,14] the construction of a two-level supercompiler by combining two single-result supercompilers is unable to reveal many opportunities for two-level supercompilation. However, using multi-result supercompiler as the lower supercompiler produces new results.

So using a multi-result supercompiler as the lower supercompiler in a two-level supercompiler increases the potential of two-level supercompilation and may be regarded as branching growth of the penultimate level according to V.F. Turchin.

5.4 Stage 3. Multi-Generalization: Refinement of Control

Supercompilation can be described as a non-deterministic algorithm [11], which has a set of choices at almost every step. But a deterministic implementation of supercompilation has to choose a single variant at each step.

The previous section shows how to build a multi-result supercompiler by combining several single-result supercompilers (treated as black-boxes). Multi-result supercompilers thus produced will be said to be of the first kind.

Is it possible to derive a multi-result supercompiler from a single-result one? Yes, by turning a deterministic supercompiler into a non-deterministic one

and then turning it into a multi-result one. The subtle point is that a non-deterministic supercompiler, in principle, may produce an infinite number of residual programs or even not terminate, while a multi-result supercompiler, for practical reasons, should always terminate and produce a finite set of residual programs. Hence, we need some reasonable finite non-determinism.

Multi-generalization, presented in [14], is a technique that enables a single-result supercompiler to be turned into a multi-result supercompiler that always terminates and produces a finite number of residual programs.

The main idea of multi-generalization is that when a supercompiler has to generalize a configuration, it should consider *all* possible generalizations of the configuration, rather than a most specific generalization only. If the set of possible generalizations is always finite (which is true of the HOSC supercompilation relation), the set of possible residual programs is also finite. A multi-result supercompiler thus produced will be said to be of the second kind.

If Sc' is a multi-result single-level supercompiler of the second kind and Sc'' a multi-result single-level supercompiler of the first kind, $L2'(Sc', Sc'')$ turns out to be a powerful multi-result two-level supercompiler, which is capable of finding non-trivial generalizations by means of multi-generalization, thereby coming to non-trivial results.

Let us consider the configuration `rev xs` for the program in Fig. 6. During supercompilation the following configuration:

```
case case v5 of {Cons p q → app (rev q) (Cons p Nil); Nil → Nil;} of {
    Cons r s → Cons r (app s (Cons v4 Nil));
    Nil → Cons v4 Nil;
}
```

is compared with a bit complicated configuration:

```
case
case (case v2 of {Cons p q → app (rev q) (Cons p Nil); Nil → Nil;}) of {
    Cons t u → Cons t (app u (Cons v1 Nil));
    Nil → Cons v1 Nil;
} of {
    Cons r s → Cons r (app s (Cons v4 Nil));
    Nil → Cons v4 Nil;
}
```

After this comparison a classic single-level deterministic supercompiler would perform the following generalization of the first configuration:

```
let g = case v5 of {Cons p q → app (rev q) (Cons p Nil); Nil → Nil;}
in
case g of {
    Cons r s → Cons r (app s (Cons v4 Nil));
    Nil → Cons v4 Nil;
}
```

which would not produce non-trivial results. However, by using multi-generalization, it is possible to find the following generalization:

```
let g = Cons v4 Nil in
case case v5 of {Cons p q → app (rev q) (Cons p Nil); Nil → Nil;} of {
    Cons r v21 → Cons r (app s g);
    Nil → g;
}
```

This allows the upper supercompiler to make a conjecture that is provable by the lower supercompiler, which gives the residual program shown in Fig. 7.

A yet another interesting point is that, in the case of two-level supercompilation, given an input program, we can introduce a "measure of non-triviality" on the set of corresponding residual programs. Namely, the "non-triviality" can be defined as the number of lemmas used during supercompilation. Note that in the case of single-level supercompilation there seems to be no simple measure of "non-triviality".

So, we see that multi-result supercompilation provides numerous opportunities for refining control.

6 MRSC: a Multi-Result Supercompilation Framework

For the first time, the described examples were obtained and tested when working with an experimental version of the supercompiler HOSC[8] for the language Haskell. Later we created an experimental version of the supercompiler SPSC[9] dealing with a first-order functional language, and were able to reproduce similar examples in the first-order setting. These results led us to the idea of the framework MRSC.

Before going into the details of MRSC, let us classify the ways of implementing a supercompiler. Basically, there are two main approaches. The first one is based on constructing a graph of configurations and, after the graph is completed, transforming it into a residual program [28,23,15,9]. (Let us call it "the graph-based approach".) Obviously, this graph of configurations is an intermediate data structure that has to be constructed and deconstructed, which slows down the supercompiler. For this reason, in order to be fast, some optimizing supercompilers [1,20,7], avoid the construction of graphs of configurations. (Let us call it "the direct approach".)

We have found that multi-result supercompilers are easier to implement using the graph-based approach rather than the direct approach. Also, when a supercompiler is used as a prover, a graph of configurations can be used to extract a proof readable by humans. Moreover, some additional transformations, which cannot be easily performed in a direct-style supercompiler, can be applied to a completed graph. For example, some parts of the graph can be re-arranged in order to simplify it. (However, for the lack of space, we will not discuss such transformations here.)

[8] `http://code.google.com/p/hosc/`, [9].
[9] `http://code.google.com/p/spsc/`, [15].

```
data DriveStep e = DriveStep e
type Driver e = e → DriveStep e
type Whistle e = [e] → e → Maybe e
type Rebuilder e = e → e → e
type MRebuilder e = e → e → [e]

data SC e = SC {drive :: Driver e, whistle :: Whistle e,
                rebuild :: Rebuilder e}
data MSC e = MSC {mdrive :: Driver e, mwhistle :: Whistle e,
                  mrebuild :: MRebuilder e}

data SCGraph e = SCGraph e

runSC :: SC e → e → e
runMSC :: MSC e → e → [e]
```

Fig. 8. MRSC: base abstractions

MRSC[10] is a multi-result supercompilation framework that is agnostic to the object language it deals with. The base abstractions MRSC is based upon are sketched (in pseudo-Haskell) in Fig. 8. At the heart of MRSC is a mini-framework for manipulating graphs of configurations, the core concept being `SCGraph e`, representing a supercompilation graph, parameterized by `e`, the type of expressions used as configurations. The logic of a supercompiler `SC e` is represented in MRSC as a set of functions for driving, identifying dangerous configurations (that might cause nontermination), and rebuilding of configurations.

`drive` evaluates an expression with free variables, `whistle` checks an expression in the history for being dangerous (i.e. a possible cause of non-termination of the transformation), and `rebuild e1 e2` rebuilds a current expression[11] `e2` with respect to a dangerous expression `e1`[12]. If we encode a classical positive supercompiler [24,23] in terms of MRSC, then the whistle will be implemented as the homeomorphic embedding relation, and the rebuilder of configurations as a most specific generalization.

However, in MRSC a supercompiler `SC e` does not perform transformations: it just represents the logic of a supercompiler. All dirty work of constructing supercompilation graphs and transforming them into residual programs is done by `runSC`, "applying" a supercompiler to an expression.

[10] `http://github.com/ilya-klyuchnikov/mrsc`, [14].

[11] Historically, the are two approaches to rebuilding: rebuilding of the current expression and rebuilding of the dangerous expression. In the latter case we need to prune a subtree with a root labeled by a dangerous expression. Here we consider the rebuilding of the current expression only (for the sake of brevity and simplicity).

[12] In order to ensure the correctness of transformations, we require `e1` to be an improvement of `e2` [22].

The logic of a multi-result supercompiler `MSC e` differs from that of a single-result supercompiler in a single detail: it may rebuild a current expression in several different ways. As in the case of `SC e`, `MSC e` does not perform transformations: all transformations are done by `runMSC`. The peculiarity of `runMSC` is that when multiple rebuildings are encountered, `runMSC` applies all of them "in parallel" by multiplying the current graph of configuration and applying each rebuilding to the corresponding copy[13].

The main feature of MRSC is that, by design, `runMSC` always produces a finite set of residual expressions [14].

6.1 Constructing Two-Level Supercompilers

A supercompiler written in functional style usually is represented as a composition of functions [1,20,13]. The subtle problem with such representation is that it is almost impossible to extract the ingredients of this composition in order to modify and rearrange them in a new way. This is why a supercompiler in MRSC is represented as a decomposable structure: we can disassemble a supercompiler, modify some of its ingredients, and re-assemble the modified parts back.

Now let us consider some ways of constructing new supercompilers from existing ones by means of MRSC. A few recipes described in previous sections are, more formally, presented in Fig. 9.

First, we have to define what is a substitution `Subst e` for expressions of type `e` and to implement the operation `//`, applying a substitution to an expression, and the function `test`, discovering whether there is a correspondence via substitution between two expressions. Then we can use three constructors of two-level supercompilers provided by MRSC: `l2`, `l2'` and `l2''`.

`l2` replaces the rebuilder `rb` of the given supercompiler `sc` with a new rebuilder `rb'`: when there is a request to rebuild an expression `e2` with respect to a dangerous expression `e1`, the new rebuilder `rb'` tries to find a substitution between the supercompiled (by the unmodified supercompiler `sc`) expressions, and if there is any, it applies this substitution to the dangerous expression and returns the result. Otherwise, it delegates the rebuilding to the original `rb`. `l2'` is defined in a similar way, but tries to find a substitution by means of the second supercompiler `sc`.

The constructor `l2''` checks all combinations of the residual expressions produced by a lower multi-result supercompiler `msc`. In the next section we will see how to construct a multi-result supercompiler from a single-result one.

6.2 Working with Multi-Result Supercompilers

Although the framework MRSC allows multi-result supercompilers to be implemented "by bare hands", it also provides a few ready-to-use constructors for turning ordinary supercompilers into multi-result ones.

[13] No real copying is performed here: the pieces of the "old" graph are just shared by new graphs.

```
type Subst e = e → e
(//) :: e → Subst e → e
test :: e → e → Maybe (Subst e)

-- makes a two-level single-result supercompiler
-- from a single-result supercompiler
l2 :: SC e → SC e
l2 sc@(SC d w rb) = SC d w rb' where
    rb' e1 e2 =
      maybe (rb e1 e2) (e1 //) (test (runSC sc e1) (runSC sc e2))
-- makes a two-level single-result supercompiler by combining
-- two different single-result supercompilers
l2' :: SC e → SC e → SC e
l2' (SC d w rb) sc = SC d w rb' where
    rb' e1 e2 =
      maybe (rb e1 e2) (e1 //) (test (runSC sc e1) (runSC sc e2))
-- makes a two-level single-result supercompiler by combining
-- a single-result supercompiler and multi-result supercompiler
l2'' :: SC e → MSC e → SC e
l2'' (SC d w rb) msc = SC d w rb' where
    rb' e1 e2 = maybe (rb e1 e2) (e1 //) res where
        res = msum [test x y | x <- es1, y <- es2]
        (es1, es2) = (runMSC msc e1, runMSC msc e1)
```

Fig. 9. MRSC: recipes for constructing single-result two-level supercompilers

The constructor `multi`, shown in Fig. 10, is the simplest one. It just replaces the ordinary rebuilding of a current expression with respect to a dangerous expression by a multi-generalization of the current expression. A surprising fact is that `multi` builds a supercompiler that always produces a finite set of residual programs, regardless of how `runMSC` is implemented.

The constructor `multi'` combines a multi-result supercompiler with a single-result one to produce a new multi-result supercompiler. The main trick here is that the new rebuilder `rb'` does not throw away the old rebuildings, but merges them with the *single* rebuilding (if any) returned by the lower supercompiler `sc`. This trick is further strengthened in the constructor `multi''`, where the set of old rebuildings is merged with the *set* of new rebuildings.

6.3 The Current State and Directions of Future Work

MRSC is a work in progress and is under active development now. So far we have finalized and tuned the core part of the framework operating, in a generic way, with graphs of configurations. Also we have reimplemented SPSC, a classical supercompiler for a simple first-order functional language SLL [24,23,15], by means of MRSC, and then automatically transformed SPSC into its multi-result

```
type MGeneralizer e = e → [e]
-- makes a multi-result one-level supercompiler by combining
-- a single-result supercompiler and multi-generalization
multi :: SC e → MGeneralizer e → MSC e
multi (SC d w _) g = MSC d w rb where
    rb _ e2 = g e2
-- makes a multi-result two-level supercompiler by combining
-- an upper multi-result supercompiler and
-- a lower single-result supercompiler
multi' :: MSC e → SC e → MSC e
multi' (MSC d w rb) sc = (MSC d w rb') where
    rb' e1 e2 = ex ++ (rb e1 e2) where
        ex = map (e1 //) $ maybeToList (test (runSC sc e1) (runSC sc e2))
-- makes a multi-result two-level supercompiler by combining
-- two multi-result supercompiler
multi'' :: MSC e → MSC e → MSC e
multi'' (MSC d w rb) msc = (MSC d w rb') where
    rb' e1 e2 = extra ++ (rb e1 e2) where
        extra = map (e1 //) $ catMaybes $ [test x y | x <- es1, y <- es2]
        (es1, es2) = (runMSC msc e1, runMSC msc e1)
```

Fig. 10. MRSC: recipes for constructing multi-result supercompilers

and two-level versions (producing the same results as the corresponding hand-crafted versions).

We have carried out only "proof-of-concept" experiments with MRSC so far. Our plans are the following:

- To continue experiments with MRSC in the context of program analysis.
- To compare how various whistles affect the size and properties of the sets of residual programs generated by multi-result supercompilation. Until now, we have tried only whistles based on the homeomorphic embedding relation. However, in the context of optimizing supercompilation, there have appeared new approaches to constructing whistles, such as based on tag-bags [19,1].
- To compare how in the context of multi-result supercompilation rebuilding of the dangerous expression differs from rebuilding of the current expression.
- To reimplement HOSC, a higher-order supercompiler, in terms of MRSC.

7 Conclusion

When supercompilation is used for the purposes of program optimization, the usual practice is to consider only deterministic algorithms (involving some heuristics), which, given an input program, produce a single residual program.

However, if we reformulate supercompilation in more abstract terms, in form of a *transformation relation* [8,11], we naturally come to the idea of *multi-result*

supercompilation. Namely, given an input program p, a multi-result supercompiler may produce several output programs p', such that $p\ SC\ p'$, where SC is a supercompilation relation.

Note that we differentiate the terms *non-deterministic* supercompilation and *multi-result* supercompilation. A non-deterministic supercompiler, in principle, may produce an infinite number of residual programs, or even not terminate, while a multi-result supercompiler, for practical reasons, should always terminate and produce a finite set of residual programs.

We have demonstrated that, when used for program analysis, multi-result supercompilation produces more nontrivial and stable results, as compared to single-result supercompilation.

The fact that multi-result supercompilation naturally arises in the context of two-level supercompilation, can be regarded as a manifestation of the general law of branching growth of the penultimate level in a metasystem transition [26]. It would be interesting to consider other manifestations of this law in the field of computer science.

Acknowledgements

The authors express their gratitude to all participants of Refal seminar at Keldysh Institute for useful comments and fruitful discussions of this work. Extra special thanks are to Sergei Abramov for his attention and support of this work.

References

1. M. Bolingbroke and S. L. Peyton Jones. Improving supercompilation: tag-bags, rollback, speculation, normalisation, and generalisation, 2011. Submitted to ICFP 2011.
2. Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.
3. R. Glück. Is there a fourth Futamura projection? In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 51–60, New York, NY, USA, 2009. ACM.
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. G. W. Hamilton and M. H. Kabir. Constructing programs from metasystem transition proofs. In *Proceedings of the First International Workshop on Metacomputation in Russia*, 2008.
7. P. Jonsson. *Positive supercompilation for a higher-order call-by-value language*. Luleå University of Technology, 2008.
8. A. Klimov. A program specialization relation based on supercompilation and its properties. In *Proceedings of the First International Workshop on Metacomputation in Russia*, pages 54–78. Ailamazyan University of Pereslavl, 2008.

9. I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.
10. I. Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting. Preprint 62, Keldysh Institute of Applied Mathematics, 2010.
11. I. Klyuchnikov. Supercompiler HOSC: proof of correctness. Preprint 31, Keldysh Institute of Applied Mathematics, Moscow, 2010.
12. I. Klyuchnikov. Towards effective two-level supercompilation. Preprint 81, Keldysh Institute of Applied Mathematics, 2010.
13. I. Klyuchnikov. The ideas and methods of supercompilation. *Practice of Functional Programming*, 7, 2011. In Russian.
14. I. Klyuchnikov. MRSC: a framework for multi-result supercompilation. Preprint, Keldysh Institute of Applied Mathematics, 2011. To appear.
15. I. Klyuchnikov and S. Romanenko. SPSC: a simple supercompiler in Scala. In *PU'09 (International Workshop on Program Understanding)*, 2009.
16. 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.
17. I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.
18. 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.
19. N. Mitchell. Rethinking supercompilation. In *ICFP 2010*, 2010.
20. 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.
21. A. P. Nemytykh, V. A. Pinchuk, and V. F. Turchin. A self-applicable supercompiler. In *Selected Papers from the Internaltional Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 322–337, 1996.
22. D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.
23. M. Sørensen, R. Glück, and N. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1993.
24. M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Københavns Universitet, Datalogisk Institut, 1994.
25. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. *SIGPLAN Not.*, 44:264–276, January 2009.
26. V. Turchin. *The phenomenon of science. A cybernetic approach to human evolution.* Columbia University Press, New York, 1977.
27. V. Turchin. *The Language Refal: The Theory of Compilation and Metasystem Analysis.* Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 1980.
28. V. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.
29. V. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(03):283–313, 1993.
30. V. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In *Partial Evaluation*, volume 1110 of *LNCS*, pages 481–509. Springer, 1996.