Staged Multi-Result Supercompilation: Filtering by Transformation*

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

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

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

1 Introduction

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

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

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

2 Filtering before Producing... How?

2.1 Multi-Result Supercompilation and Filtering

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

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

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

```
solver = filter ∘ mrsc
```

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

2.2 Fusion of Supercompilation and Filtering

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

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

```
solver′ = fuse filter mrsc
```

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

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

2.3 Staged Supercompilation: Multiple Results Seen as a Residual Program

Here we put forward an alternative approach that:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4 Typical Cleaners

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

2.5 What are the Advantages?

We get the following scheme:

$$\begin{aligned}&\texttt{filter} \circ \texttt{naive-mrsc} \doteq \\ &\quad \texttt{filter} \circ \langle\!\langle\_\rangle\!\rangle \circ \texttt{lazy-mrsc} \doteq \langle\!\langle\_\rangle\!\rangle \circ \texttt{clean} \circ \texttt{lazy-mrsc}\end{aligned}$$

We can see that:

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

2.6 Codata and Corecursion: Decomposing `lazy-mrsc`

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

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

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

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

2.7 Cleaning before Whistling

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

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

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

then

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

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

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

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

```
⟪ clean (lazy-mrsc c) ⟫
```

3 A Model of Big-Step Multi-Result Supercompilation

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

3.1 Graphs of Configurations

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

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

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

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

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

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

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

3.2 "Worlds" of Supercompilation

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

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

---

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

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

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

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

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

```
record ScWorld : Set₁ where

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

  open BarWhistle whistle public

  History : Set
  History = List Conf

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

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

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

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

[3] Record declarations in Agda are analogous to "abstract classes" in functional languages and "structures" in Standard ML, abstract members being declared as "fields".
[4] The construct `open BarWhistle whistle public` brings the members of `whistle` into scope, so that they become accessible directly.

3.3 Graphs with labeled edges

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

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

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

There is defined (in BigStepSc.agda) a function

```
injectLabelsInScWorld : ScWorldWithLabels → ScWorld
```

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

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

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

```
infix 4 _⊢NDSC_↪_ _⊢NDSC*_↪_

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

which are defined with respect to a world of supercompilation.

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

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

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

⊢NDSC_↪_ is defined by two rules

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

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

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

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

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

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

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

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

```
infix 4 _⊢MRSC_↪_ _⊢MRSC*_↪_

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

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

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

`_⊢MRSC_↪_` is defined by two rules

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

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

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

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

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

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

3.6 Bar Whistles

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

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

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

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

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

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

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

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

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

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

    bar[] : Bar ↯ []
```

where

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

3.7 A Function for Computing Cartesian Products

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  `naive-mrsc′ (c :: h) (bs c) c′`

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

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

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

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

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

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

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

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

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

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

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

4 Staging Big-Step Multi-Result Supercompilation

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

```
naive-mrsc ≗ ⟪_⟫ ∘ lazy-mrsc
```

At the first stage, `lazy-mrsc` generates a “lazy graph”, which, essentially, is a “program” to be “executed” by `⟪_⟫`.

4.1 Lazy Graphs of Configurations

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

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

A lazy graph is a tree whose nodes are "commands" to be executed by the interpreter ⟪_⟫.

The exact semantics of lazy graphs is given by the function ⟪_⟫, which calls auxiliary functions ⟪_⟫* and ⟪_⟫⇉ (see `Graphs.agda`).

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

Here is the definition of the main function ⟪_⟫:

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

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

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

The function ⟪_⟫* considers each lazy graph in a list of lazy graphs `ls`, and turns it into a list of graphs:

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

The function ⟪_⟫⇉ considers all possible decompositions of a configuration, and for each decomposition computes all possible combinations of subgraphs:

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

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

4.2 A Functional Specification of Lazy Multi-Result Supercompilation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5 Cleaning Lazy Graphs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Cleaner `cl-empty`

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

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

`cl-bad-conf` is correct with respect to ⟪_⟫:

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

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

5.3 Cleaner `cl-min-size`

The function `cl-min-size`

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

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

More formally,

- `⟪ l′ ⟫ = [ g′ ]`.
- `graph-size g′ = k`

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

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

– `k` $\leq$ `graph-size g` for all `g` $\in$ ⟪ `l` ⟫.

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

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

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

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

6 Codata and Corecursion: Cleaning before Whistling

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

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

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

6.1 Lazy Cographs of Configurations

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

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

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

6.2 Building Lazy Cographs

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

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

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

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

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

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

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

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

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

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

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

6.3 Pruning Lazy Cographs

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

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

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

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

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

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

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

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

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

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

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

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

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

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

6.4 Promoting some Cleaners over the Whistle

Suppose `clean∞` is a cograph cleaner such that

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

then

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

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

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

may be less time and space consuming than evaluating

```
⟪ clean (lazy-mrsc c) ⟫
```

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

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

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

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

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

7 Related Work

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

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

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

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

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

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

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

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

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

8 Conclusions

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

We have suggested the following solution.

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

Acknowledgements

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

References

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

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

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

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