Formalizing and Implementing Multi-Result Supercompilation

Ilya G. Klyuchnikov    Sergei A. Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

2012-07 / Meta 2012
Outline

1. Different types of supercompilation
   - SC: Deterministic/traditional SC (a function)
   - NDSC: Non-deterministic SC (a relation)
   - MRSC: Multi-result SC (a multi-valued function)

2. Nice features of multi-result supercompilation
   - Finiteness of trees of completed graphs
   - Decoupling whistle and generalization

3. The core of the MRSC Toolkit
   - Two representations for graphs of configurations
   - Operations on S-graphs

4. Conclusions
Outline

1 Different types of supercompilation
   - SC: Deterministic/traditional SC (a function)
   - NDSC: Non-deterministic SC (a relation)
   - MRSC: Multi-result SC (a multi-valued function)

2 Nice features of multi-result supercompilation
   - Finiteness of trees of completed graphs
   - Decoupling whistle and generalization

3 The core of the MRSC Toolkit
   - Two representations for graphs of configurations
   - Operations on S-graphs

4 Conclusions
1 Different types of supercompilation
   • SC: Deterministic/traditional SC (a function)
   • NDSC: Non-deterministic SC (a relation)
   • MRSC: Multi-result SC (a multi-valued function)

2 Nice features of multi-result supercompilation
   • Finiteness of trees of completed graphs
   • Decoupling whistle and generalization

3 The core of the MRSC Toolkit
   • Two representations for graphs of configurations
   • Operations on S-graphs

4 Conclusions
Outline

1. Different types of supercompilation
   - SC: Deterministic/traditional SC (a function)
   - NDSC: Non-deterministic SC (a relation)
   - MRSC: Multi-result SC (a multi-valued function)

2. Nice features of multi-result supercompilation
   - Finiteness of trees of completed graphs
   - Decoupling whistle and generalization

3. The core of the MRSC Toolkit
   - Two representations for graphs of configurations
   - Operations on S-graphs

4. Conclusions
1. Different types of supercompilation
   - SC: Deterministic/traditional SC (a function)
   - NDSC: Non-deterministic SC (a relation)
   - MRSC: Multi-result SC (a multi-valued function)

2. Nice features of multi-result supercompilation
   - Finiteness of trees of completed graphs
   - Decoupling whistle and generalization

3. The core of the MRSC Toolkit
   - Two representations for graphs of configurations
   - Operations on S-graphs

4. Conclusions
(Fold) \[ \exists \alpha : foldable(g, \beta, \alpha) \quad g \rightarrow fold(g, \beta, \alpha) \]

(Drive) \[ \forall \alpha : foldable(g, \beta, \alpha) \quad \neg dangerous(g, \beta) \quad cs = driveStep(c) \quad g \rightarrow addChildren(g, \beta, cs) \]

(Rebuild) \[ \forall \alpha : foldable(g, \beta, \alpha) \quad dangerous(g, \beta) \quad c' = rebuilding(g, c) \quad g \rightarrow rebuild(g, \beta, c') \]

\textit{dangerous}(g, \beta) \text{ appears in (Drive) and (Rebuild)!}

(Drive) and (Rebuild) are mutually exclusive.

(Rebuild) is used as a last resort if (Drive) is blocked by \textit{dangerous}(g, \beta)\ldots
A finite sequence of graphs:
NDSC: Non-deterministic SC (a relation)

(Fold) \[ \exists \alpha : foldable(g, \beta, \alpha) \quad g \rightarrow fold(g, \beta, \alpha) \]

(Drive) \[ \forall \alpha : foldable(g, \beta, \alpha) \quad cs = driveStep(c) \quad g \rightarrow addChildren(g, \beta, cs) \]

(Rebuild) \[ \forall \alpha : foldable(g, \beta, \alpha) \quad c' \in rebuildings(c) \quad g \rightarrow rebuild(g, \beta, c') \]

dangerous\((g, \beta)\) has disappeared from (Drive) and (Rebuild)!

(Drive) and (Rebuild) are not mutually exclusive.

(Drive) is always applicable.
NDSC: Non-deterministic SC (a relation)

A (possibly) infinite tree of graphs:
MRSC: Multi-result SC (a multi-valued function)

(Fold) \[ \exists \alpha : \text{foldable}(g, \beta, \alpha) \implies g \to \text{fold}(g, \beta, \alpha) \]

(Drive) \[ \forall \alpha : \text{foldable}(g, \beta, \alpha) \implies \neg \text{dangerous}(g, \beta) \quad \text{cs} = \text{driveStep}(c) \implies g \to \text{addChildren}(g, \beta, \text{cs}) \]

(Rebuild) \[ \forall \alpha : \text{foldable}(g, \beta, \alpha) \quad c' \in \text{rebuildings}(c) \implies g \to \text{rebuild}(g, \beta, c') \]

\textit{dangerous}(g, \beta) reappears in (Drive), but not in (Rebuild)!

(Drive) and (Rebuild) are not mutually exclusive.

(Drive) is not always applicable. \[ \neg \text{dangerous}(g, \beta) \] ensures termination...
A (desirably) finite tree of graphs:

MRSC: Multi-result SC (a multi-valued function)
Outline

1 Different types of supercompilation
   • SC: Deterministic/traditional SC (a function)
   • NDSC: Non-deterministic SC (a relation)
   • MRSC: Multi-result SC (a multi-valued function)

2 Nice features of multi-result supercompilation
   • Finiteness of trees of completed graphs
   • Decoupling whistle and generalization

3 The core of the MRSC Toolkit
   • Two representations for graphs of configurations
   • Operations on S-graphs

4 Conclusions
Theorem (Finiteness of sets of completed graphs)

If

1. any infinite branch in a graph of configurations is detected by the predicate dangerous,
2. for any configuration $c$ the set $\text{rebuildings}(c)$ is finite,
3. the number of successive rebuildings cannot be infinite (i.e. the chain $c_1, c_2, c_3, \ldots$, where $c_{k+1} \in \text{rebuildings}(c_k)$ is always finite),

then the application of the MRSC-rules produces a finite set of completed graphs of configurations.

Proof.

Collapse all successive rebuildings into one rebuilding. Everything else follows from König lemma (using arguments similar to those in the Sørensen’s proof.)
MRSC: Decoupling whistle and generalization

(Fold)  \[ \exists \alpha : \text{foldable}(g, \beta, \alpha) \]
\[ g \rightarrow \text{fold}(g, \beta, \alpha) \]

(Drive)  \[ \forall \alpha : \text{foldable}(g, \beta, \alpha) \quad \neg \text{dangerous}(g, \beta) \quad cs = \text{driveStep}(c) \]
\[ g \rightarrow \text{addChildren}(g, \beta, cs) \]

(Rebuild)  \[ \forall \alpha : \text{foldable}(g, \beta, \alpha) \quad c' \in \text{rebuildings}(c) \]
\[ g \rightarrow \text{rebuild}(g, \beta, c') \]

Observation

\text{dangerous}(g, \beta) \text{ does not appear in (Rebuild).}

Conclusion

The whistle does not have to know anything about rebuilding (generalization).
Outline

1. Different types of supercompilation
   - SC: Deterministic/traditional SC (a function)
   - NDSC: Non-deterministic SC (a relation)
   - MRSC: Multi-result SC (a multi-valued function)

2. Nice features of multi-result supercompilation
   - Finiteness of trees of completed graphs
   - Decoupling whistle and generalization

3. The core of the MRSC Toolkit
   - Two representations for graphs of configurations
   - Operations on S-graphs

4. Conclusions
A T-graph:

- Good for top-down traversal of graphs.
- Convenient when transforming a graph into a residual program.
- However, when making additions to a T-graph in two different ways, we have to do some copying.
- However, a deterministic supercompiler deals with a single graph! Hence, no copying is required...
Why T-representation is not good for MRSC?

A problem
- MRSC is able to produce millions of graphs of configurations.
- Huge memory consumption, a lot of copying...

A simple solution
- Sharing!

A sophisticated solution (Sergei Grechanik)
- Hypergraphs, hyperedges.
S-representation (= based on spaghetti-stacks)

An S-graph:

- Good, when making additions to an S-graph in **two different ways**, as no **copying** is required.
- Convenient for a multi-result supercompiler dealing with large collections of graph!
Reuse of nodes in S-graphs

S-graphs are immutable!

“Modifying” an S-graph in different ways we create new S-graphs.

The original S-graphs and derived S-graphs share common parts.
An implementation in Scala: T-graphs

type TPath = List[Int]

case class TNode[C, D](
    conf: C, outs: List[TEdge[C, D]],
    base: Option[TPath], tPath: TPath)

case class TEdge[C, D](
    node: TNode[C, D], driveInfo: D)

case class TGraph[C, D](
    root: TNode[C, D], leaves: List[TNode[C, D]])

- C is the type of configurations.
- D is the type of edge labels, produced by driving.
- TPath is the type of paths to nodes.
An implementation in Scala: S-graphs

type SPath = List[Int]

case class SNode[C, D](
    conf: C, in: SEdge[C, D],
    base: Option[SPath], sPath: SPath)

case class SEdge[C, D](
    node: SNode[C, D], driveInfo: D)

case class SGraph[C, D](
    incompleteLeaves: List[SNode[C, D]],
    completeLeaves: List[SNode[C, D]],
    completeNodes: List[SNode[C, D]]) {

    val isComplete = incompleteLeaves.isEmpty
    val current = if (isComplete) null else incompleteLeaves.head
}
Rewrite steps for S-graphs

sealed trait GraphRewriteStep[C, D]

case class CompleteCurrentNodeStep[C, D]
  extends GraphRewriteStep[C, D]

case class AddChildNodesStep[C, D](ns: List[(C, D)])
  extends GraphRewriteStep[C, D]

case class FoldStep[C, D](to: SPath)
  extends GraphRewriteStep[C, D]

case class RebuildStep[C, D](c: C)
  extends GraphRewriteStep[C, D]

These rewriting operations form a “basis” sufficient for building S-graphs during multi-result supercompilation. (Unlike deterministic supercompilation, there are no roll-backs!)
Rewrite steps for S-graphs: Complete

- **CompleteCurrentNodeStep** — marks the current leaf as a completed one.
  Used in driving.
Rewrite steps for S-graphs: Fold

FoldStep — performs a folding.
Rewrite steps for S-graphs: AddChildNodes

- AddChildNodesStep — adds child nodes to the current node. Used in driving.
Rewrite steps for S-graphs: Rebuild

- **RebuildStep** — performs a lower rebuilding of the graph (by replacing the configuration in the current node).
trait GraphRewriteRules[C, D] {
    type N = SNode[C, D]
    type G = SGraph[C, D]
    type S = GraphRewriteStep[C, D]
    def steps(g: G): List[S]
}

case class GraphGenerator[C, D] (rules: GraphRewriteRules[C, D], conf: C) extends Iterator[SGraph[C, D]] { ... }

- A concrete supercompiler is required to provide an implementation for the method `steps`.
- `steps` does not rewrite graphs: it only generates “commands” to be executed by the MRSC Toolkit.
- The main loop of supercompilation is implemented as an iterator that produces graphs in a lazy way, by demand.
The MRSC Toolkit: publications


The MRSC Toolkit: a public repository at GitHub

There one can find:

- The Core of the MRSC Toolkit.
- A domain-specific supercompiler for counter systems.
- The results of verification of a number of communication protocols.

...
Outline

1. Different types of supercompilation
   - SC: Deterministic/traditional SC (a function)
   - NDSC: Non-deterministic SC (a relation)
   - MRSC: Multi-result SC (a multi-valued function)

2. Nice features of multi-result supercompilation
   - Finiteness of trees of completed graphs
   - Decoupling whistle and generalization

3. The core of the MRSC Toolkit
   - Two representations for graphs of configurations
   - Operations on S-graphs

4. Conclusions
3 kinds of supercompilation (deterministic, non-deterministic and multi-result one) can be specified in a uniform way by graph rewriting rules.

Under certain conditions, a multi-result supercompiler produces a finite number of residual programs and terminates.

Conceptually, multi-result supercompilation is simpler than deterministic, single-result supercompilation, since the whistle and the generalization algorithm can be completely decoupled.

The use of immutable data-structures (S-graphs) and data sharing in the implementation of multi-result supercompilation, makes it possible to generate thousands of graphs, while still keeping memory consumption within reasonable limits.