Keldysh Institute • Publication search

Keldysh Institute preprints • Preprint No. 24, 2012

Klimov A.V., Klyuchnikov I.G., Romanenko S.A.

Implementing a domain-specific multi-result supercompiler by means of the MRSC toolkit

Recommended form of bibliographic references: Klimov A.V., Klyuchnikov I.G., Romanenko S.A. Implementing a domain-specific multi-result supercompiler by means of the MRSC toolkit. Keldysh Institute preprints, 2012, No. 24, 20 p. URL: http://library.keldysh.ru/preprint.asp?id=2012-24&lg=e

KELDYSH INSTITUTE OF APPLIED MATHEMATICS
Russian Academy of Sciences

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

Implementing a domain-specific multi-result supercompiler by means of the MRSC toolkit

Moscow
2012

Andrei V. Klimov, Ilya G. Klyuchnikov, Sergei A. Romanenko. Implementing a domain-specific multi-result supercompiler by means of the MRSC toolkit

The paper presents a simple domain-specific multi-result supercompiler for counter systems implemented by means of the MRSC toolkit. The input language of the supercompiler is a non-deterministic domain-specific language meant for specifying models of communication protocols. The implementation of this DSL is based on "embedding" and the heavy use of higher-order constructs. There are presented 2 versions of the multi-result supercompiler. The first one implements a naïve algorithm, which turns out to be rather inefficient. The second version exploits the specifics of the domain, thereby drastically reducing the number of generated graphs of configurations and the amount of resources consumed by supercompilation.

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.

Андрей В. Климов, Илья Г. Ключников, Сергей А. Романенко. Реализация предметно-ориентированного многорезультатного суперкомпилятора с помощью инструментария MRSC

В работе представлен простой предметно-ориентированный многорезультатный суперкомпилятор, предназначенный для анализа поведения счетчиковых систем и реализованный с помощью инструментария MRSC. Входным языком суперкомпилятора является недетерминированный предметно-ориентированный язык, предназначенный для описания моделей коммуникационных протоколов. Реализация этого языка основана на поверхностном встраивании и существенном использовании конструкций высшего порядка. Рассматривается две версии многорезультатного суперкомпилятора. В первой из них реализован "наивный" алгоритм, который оказывается низкоэффективным. Во второй версии, благодаря учету особенностей проблемной области, удается значительно уменьшить количество порождаемых графов конфигураций и снизить потребление вычислительных ресурсов.

Работа выполнена при поддержке гранта РФФИ № 12-01-00972-а и гранта Президента РФ для ведущих научных школ № НШ-4307.2012.9.

Contents

1 Introduction

Supercompilation [22, 23] is a program manipulation technique that was originally introduced by V. Turchin in terms of the programming language Refal (a first-order applicative functional language) [21], for which reason the first supercompilers were designed and developed for the language Refal [20, 24, 16, 15].

Further development of supercompilation led to a more abstract reformulation of supercompilation [18, 19, 5]. It particular, it was shown that supercompilation is as well applicable to non-functional programming languages (imperative and object-oriented ones) [7].

Multi-result supercompilation is a technique of constructing supercompilers that, given an input program, are able to produce a set of residual programs, rather than just a single one [14, 9].

Another line of development is domain-specific supercompilation for domain-specific languages, and, as has been shown in [12], there are some cases where domain-specific supercompilation has certain advantages over general-purpose supercompilation.

- The tasks for a domain-specific supercompiler can be written in a domain-specific language in terms of a specific problem domain.
- The machinery of supercompilation can be simplified.

- The amount of resources consumed by multi-result supercompilation can be significantly reduced by exploiting the knowledge about the specifics of the domain.

However, the main argument against domain-specific supercompilation is that, upon having designed and implemented a general-purpose supercompiler, we can apply it to various problems again and again (theoretically speaking, without any extra effort). While, in the case of domain-specific supercompilation, we have to develop a whole "zoo" of supercompilers tailored for different application areas.

Hence, the use of domain-specific supercompilation seems to be justified only in cases where the design and implementation of a domain-specific supercompiler would take a fraction of the effort needed for creating a general-purpose supercompiler.

The paper presents a simple domain-specific multi-result supercompiler [14, 12] for counter systems [2, 7, 8, 6, 9, 11, 10] implemented by means of the MRSC toolkit [13]. The input language of the supercompiler is a non-deterministic domain-specific language meant for specifying models of communication protocols. The implementation of this DSL is based on embedding [17, 4] and the heavy use of higher-order constructs. There are presented 2 versions of the multi-result supercompiler. The first one implements a naïve algorithm, which turns out to be rather inefficient. The second version exploits the specifics of the domain, thereby drastically reducing the number of generated graphs of configurations and the amount of resources consumed by supercompilation.

The implementation of the aforementioned supercompilers takes only a few dozen lines of code, which is achieved by using prefabricated components provided by the MRSC toolkit [13].

Thus it can be argued that the MRSC toolkit allows domain-specific multi-result supercompilers to be manufactured at low cost, making them a budget solution, rather than a luxury.

2 Multi-result supercompilation from a bird's-eye view

As was shown in [13] various kinds of supercompilation (deterministic, non-deterministic and multi-result supercompilation) can be described by sets of rewrite rules. The rules corresponding to multi-result supercompilation are shown in Fig. 1. Given an (incomplete) graph of configurations g, the rules specify which new graphs of configurations can be produced from g by a single step of supercompilation.

Thus, the rules specify a *relation* on the set of graphs of configurations. However, when implementing a supercompiler, we need an *algorithm* which, given an initial configuration, generates a collection of completed graphs corresponding to the initial configuration.

(Fold) $$\frac{\exists \alpha : foldable(g, \beta, \alpha)}{g \to fold(g, \beta, \alpha)}$$

(Drive) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad \neg dangerous(g, \beta) \quad cs = driveStep(c)}{g \to addChildren(g, \beta, cs)}$$

(Rebuild) $$\frac{\nexists \alpha : foldable(g, \beta, \alpha) \quad c' \in rebuildings(c)}{g \to rebuild(g, \beta, c')}$$

Notation:

g – a current graph of configurations
β – a current node in a graph of configurations
c – a configuration in a current node β

Figure 1: Multi-result supercompilation specified by rewrite rules

The core of the MRSC toolkit implements an iterative algorithm that maintains a collection U of incomplete graphs of configurations. At the start, U contains a single graph whose single node contains the initial configuration.

At every step, the algorithm selects an incomplete graph g in the collection. Let $D = \{g_1, \ldots, g_n\}$ be the collection of graphs that can be immediately derived from g by applying one of the rules (Fold, Drive or Rebuild). The supercompiler removes g from U, generates the collection D and inspects the graphs in D to see which ones are completed. The completed graphs are "final products" of the supercompiler and need not be further processed. As regards the incomplete graphs appearing in D, they are added to the collection U.

When the collection U becomes empty, the algorithm terminates.

There are a number of subtle points in which this algorithm differs from those used by classic single-result supercompilers.

- The algorithm deals with a collection of incomplete graphs, rather than with a single graph. At every step, one of the graphs produces a number of "descendants" and "dies out", while in the case of single-result supercompilation, there is a single graph that "grows" and "evolves" until it becomes completed.

- Upon selecting a graph g in the collection U, the algorithm tries to apply each of the rules Fold, Drive and Rebuild to g. However, some rules may turn out to be inapplicable. In the extreme case, when no rule is applicable, g just "dies out", without producing descendants. Thus the size of the collection U gets reduced, which, in the long run, leads to the termination of the algorithm.

- The rules Drive and Rebuild can be triggered simultaneously, while in the case of single-result supercompilation driving and rebuilding are mutually

exclusive.

- The whistle controls the applicability of the rules, but has no effect on the results they produce. In particular, in contrast to single-result supercompilation, the whistle need not to take care of how and where to generalize configurations.

- Generalization is only performed for the leaves of graphs of configurations. Thus, in contrast to single-result supercompilation, there never arises a need to perform the *upper rebuilding* of a graph of configurations (a rollback to α), which consists in the deletion of all successors of the node α, followed by the replacement of a configuration c in α with a configuration c'.

Hence, as paradoxical as it may seem, the anatomy of multi-result supercompilation, in some respects, is simpler than that of single-result supercompilation.

3 Supercompilation for counter systems

One of the applications of multi-result supercompilation is the verification of cache coherence and communication protocols modeled by counter systems [2]. Which of the techniques used by general-purpose supercompilers are really essential for the analysis of counter systems? This question was investigated by Klimov, who developed several specialized supercompilation algorithms, which were proven to be correct, always terminating, and able to solve reachability problems for a certain class of counter systems [7, 8, 6, 9, 11, 10].

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

- It is sufficient to deal with configuration of the form $(a_1, \ldots, a_n)$, whose each component a_i is either a natural number N, or the symbol ω.

- As regards driving, it is sufficient to deal with tests of the form either $e = N$, or $e \geq N$, where N is a natural number and e is an arithmetic expression that can only contain the operators $+$, $-$, natural numbers and ω. The operations on arguments with ω are performed in the following way: $\omega \geq N = \texttt{True}$ and $\omega + N = \omega - N = \omega + \omega = \omega$.

- All generalizations of a configuration c can be obtained by replacing some numeric components of c with ω.

- The termination of the supercompilation algorithm is ensured by means of a very simple whistle: if a component of a configuration is a natural number n, and $n \geq l$, where l is a constant given to the supercompiler as one of its input parameters, then the configuration is declared to be "dangerous". It can be easily seen that, given an l, the set of "non-dangerous" configurations is finite.

In the following sections we present an implementation of a domain-specific multi-result supercompiler for counter systems that has been used for producing minimal graphs of configuration for a number of protocols [12]. The implementation is based on the use of prefabricated components provided by the MRSC toolkit [13]. The source codes of the MRSC toolkit and of the supercompiler can be found at `https://github.com/ilya-klyuchnikov/mrsc`.

4 A domain-specific input language and its implementation

```
package mrsc.counters

case object MSI extends Protocol {
  val start: Conf = List(Omega, 0, 0)
  val rules: List[TransitionRule] = List(
    {case List(i, m, s) if i >= 1 => List(i + m + s - 1, 1, 0)},
    {case List(i, m, s) if s >= 1 => List(i + m + s - 1, 1, 0)},
    {case List(i, m, s) if i >= 1 => List(i - 1, 0, m + s + 1)}
  )

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

Figure 2: MSI protocol: a protocol model in form of a DSL program

```
package mrsc

package object counters {
  type Conf = List[Expr]
  type TransitionRule = PartialFunction[Conf, Conf]
  implicit def intToExpr(i: Int): Expr = Num(i)
}
```

Figure 3: Package `mrsc.counters`: declarations and implicit conversions

We consider a domain-specific supercompiler for counter systems, which takes as input a specification of a communication protocol written in a domain-specific language (DSL). Fig. 2 shows a model of the MSI protocol in form of a DSL

```
package mrsc.counters

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

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

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

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

Figure 4: DSL for specifying counter systems: its implementation in Scala

program. Additional examples of protocol models with an explanation of the meaning of DSL programs can be found in [12].

Note that this DSL is far from being ideal, as some constructs look somewhat awkward. For example, each rule starts with the combination `case List(i, m, s) if`, which, for example, could be shortened to `(i, m, s) |`. Moreover, the size of the program could be further reduced by specifying the list of variables `(i,`

`m, s)` once, at the beginning of the program, instead of repeating it in every rule. However, we have preferred not to implement these improvements just because they are easy to implement in the language Scala [17] by using a number of standard tricks [4]. Since, at the moment, our goal is to show how a domain-specific supercompiler can be implemented by means of MRSC, we have chosen to minimize the DSL-related parts of the supercompiler.

Thus the implementation of the DSL has been reduced to a bare minimum and takes only a few lines of code (see Fig. 3). A "configuration" is a list of "expressions", and a "transition rule" is a partial function taking a configuration to another configuration. Why has this function to be partial? Just because a transition rule may not be applicable to a configuration, in which case the function is not defined for this configuration. Otherwise, if a rule is applicable to a configuration, it produces a single configuration. Note that the supercompiler deals with configurations that does not contain variables, for which reason the edges in graphs of configurations do not have to be labeled with conditions on variables.

We exploit the fact that, in Scala, the construct `case p if c => e`, where `p` is a pattern, `c` a condition, and `e` an expression, defines an anonymous partial function. If the function's arguments matches `p` and satisfies p, then the function returns the value obtained by evaluating e. Otherwise, the function is undefined for this argument. Therefore, a transition rule can just be written as `case p if c => e`.

A task for our domain-specific supercompiler comprises the following parts.

- `start` – an initial configuration.
- `rules` – a list of transition rules.
- `unsafe` – a predicate that determines whether a configuration is "unsafe".

The predicate `unsafe` is a domain-specific part of a supercompilation task, and is used for determining whether a graph of configurations contains "unsafe" configurations, or not.

A formal definition of the notion of "a protocol model" is given in Fig. 4 in form of the trait `Protocol`. A protocol model is a task for the supercompiler, but, on the other hand, can be regarded as a program in a domain-specific language.

Note that a DSL program is not a first-order value (as is implicitly assumed in the classic formulation of the Futamura projections [3]), but rather is a mixture of first-order values (numbers, lists) and higher-order values (functions). This approach is close to the DSL implementation technique known as "embedding" [4]. Since the embedded DSL inherits the constructs of its host language Scala, its implementation is so trivial, taking about 10 lines of code.

To be fair, we should note that the DSL's implementation is based on the use of "expressions" (`Expr`), whose implementation additionally takes about 20 lines of code (and will be described in the next sections). However, expressions are used not only in DSL programs, but also as components of configurations.

5 Operations over configurations

As has been pointed out in Section 3, a configuration has the form $(a_1, \ldots, a_n)$, where each component a_i is either a natural number N, or the symbol ω. In the Scala implementation, a configuration is represented by a list of "expressions" of type `Expr` (see Fig. 4).

The type `Expr` is declared as a trait with 2 subtypes: the class `Num`, representing natural numbers, and the object `Omega`, representing the symbol ω. There are defined the following operations over values of the type `Expr`: `+` (addition), `-` (subtraction), `>=` (comparison $\geq$) and `===` (the test for equality $=$). The equality operation is denoted by `===`, because `==` and `=` are used in Scala programs for other purposes.

Note that, when an operand of an operation is the symbol ω (representing an arbitrary natural number), the result of the operation is an "upper approximation". For instance, $\omega \geq N$ is assumed to be true for any natural number N, since the conditions of that kind are used for deciding whether a rule can be triggered? Since ω is a wildcard representing an arbitrary natural number N', and $N' \geq N$ is true for some N', the rule may be applicable in some cases. Hence, to be on the safe side, $\omega \geq N$ is assumed to be true.

Thus, in Fig. 3 and 4, we now have a complete implementation of the DSL for formulating tasks for the supercompiler. At the same time we get an implementation of the language of configurations and of a number of operations over components of configurations, which will be used in the implementation of driving.

Besides driving, the supercompiler has to perform two more operations: testing whether a configuration c_1 is an instance of a configuration c_2, and enumerating all possible generalizations of a configuration c. An implementation of these operations is shown in Fig. 5.

The function `instanceOf` tests whether a configuration `c1` is an instance of a configuration `c2`.

The function `genExpr` generates all possible generalization of an expression (which is a component of a configuration). Note that the original expression is included into the set of generalization. The set of generalization of the symbol ω contains only the symbol ω, while the set of generalizations of a number N consists of two elements: N and ω.

The function `gens` generates the set of all possible generalizations of a configuration c. Note that c is not included into this set.

The function `oneStepGens` generates the set of all generalizations of a configurations c that can be produced by generalizing a single component of c. This function will be used in the optimized version of the supercompiler described in Section 7.

```
package mrsc.counters

object Conf {
  def instanceOf(c1: Conf, c2: Conf): Boolean =
    (c1, c2).zipped.forall((e1, e2) => e1 == e2 || e2 == Omega)

  def gens(c: Conf) =
    product(c map genExpr) - c

  def oneStepGens(c: Conf): List[Conf] =
    for (i <- List.range(0, c.size) if c(i) != Omega)
      yield c.updated(i, Omega)

  def product[T](zs: List[List[T]]): List[List[T]] = zs match {
    case Nil       => List(List())
    case x :: xs => for (y <- x; ys <- product(xs)) yield y :: ys
  }

  private def genExpr(c: Expr): List[Expr] = c match {
    case Omega            => List(Omega)
    case Num(i) if i >= 0 => List(Omega, Num(i))
    case v                => List(v)
  }
}
```

Figure 5: Operations over configurations: testing for instances and building generalizations

```
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]] { ... }
```

Figure 6: MRSC “middleware” for supercompiler construction

6 Graph builder

6.1 From rewrite rules to an algorithm

Technically, a supercompiler written using MRSC is based upon two components shown in Fig. 6: `GraphRewriteRules` and `GraphGenerator` [13].

The trait `GraphRewriteRules` declares the method `steps`, which is used in the main loop of supercompilation for obtaining all graphs that can be derived from a given incomplete graph g by applying the rewrite rules Fold, Drive and Rebuild shown in Fig. 1. Namely, `steps(`g`)` returns a list of "graph rewrite steps" [13]. Then the graph generator applies each of these "steps" to the graph g to produce the collection of the descendants of g.

A concrete supercompiler is required to provide an implementation for the method `steps`. The class `GraphGenerator`, by contrast, is a ready-to-use component: it is a constituent part of any supercompiler built on top of MRSC.

In the case of supercompilation for counter systems the method `steps` can be straightforwardly implemented as shown in Fig. 7.

The methods `fold`, `drive` and `rebuild` correspond to the rewrite rules Fold, Drive and Rebuild (Fig. 1). Since the rewrite rules are independent from each other, the body of the method (steps) could have been defined in the following trivial way:

```
fold(g) ++ rebuild(g) ++ drive(g)
```

However, we have preferred to slightly optimize the implementation by taking into account that the rule Fold is mutually exclusive with the rules Drive and Rebuild. Another subtle point is that, in general, the rule Fold is non-deterministic, because the current configuration may be foldable to several configurations in the graph. Thus, the rule Fold may be applicable in zero, one or more ways. However, in the case of counter systems, all variants of folding are equally good. For this reason, in the implementation in Fig. 7, the method `fold` returns no more than one variant of folding, the type of the results being `Option[S]`, rather than `List[S]`. And the rules Drive and Rebuild are only applied if `fold` returns zero results.

The implementations of the methods `fold` and `rebuild` are straightforward.

The method `dangerous` implements the whistle suggested by Klimov [6, 11, 10]: a configuration is considered as "dangerous" if it contains a number N, such that $N \geq l$, where l is a constant given to the supercompiler as one of its input parameters.

The implementation of the method `drive` uses an auxiliary method `next`, which tries to apply all transition rules to a configuration c. If a rule is applicable, it returns a configuration c', in which case the pair $(c', ())$ is included in the list returned by `next`. In general, this pair has the form c', d, where c' is the new configuration and d the label for the edge entering the node containing the configuration c'. But, in the case of counter systems, edges need not be labeled, for which reason we put the placeholder () in the second component of the pair.

```
package mrsc.counters

class MRCountersRules(protocol: Protocol, l: Int)
  extends GraphRewriteRules[Conf, Unit] {

  override def steps(g: G): List[S] =
    fold(g) match {
      case None    => rebuild(g) ++ drive(g)
      case Some(s) => List(s)
    }

  def fold(g: G): Option[S] = {
    val c = g.current.conf
    for (n <- g.completeNodes.find(n => instanceOf(c, n.conf)))
      yield FoldStep(n.sPath)
  }

  def drive(g: G): List[S] =
    if (dangerous(g)) List()
    else List(AddChildNodesStep(next(g.current.conf)))

  def rebuild(g: G): List[S] =
    for (c <- gens(g.current.conf))
      yield RebuildStep(c): S

  def dangerous(g: G): Boolean =
    g.current.conf exists
      { case Num(i) => i >= l; case Omega => false }

  def next(c: Conf): List[(Conf, Unit)] =
    for (Some(c) <- protocol.rules.map(_.lift(c)))
      yield (c, ())
}
```

Рис. 7: Graph rewrite rules: an implementation for counter systems

6.2 Functional programming meets OOP

Note that the MRSC toolkit [13] provides infrastructure for writing supercompilers in "functional" style. Graphs of configurations are never "transformed": an incomplete graph g produces descendants and "dies out". In the implementation, however, the descendants of a graph g share some parts of their parent g, which enables the supercompiler to deal with thousands of graphs.

Thus, in the case of multi-result supercompilation, this functional approach has certain advantages over the imperative-style approach based on the language SCPL suggested by Turchin [23].

```
package mrsc.counters

class SRCountersRules(protocol: Protocol, l: Int)
  extends MRCountersRules(protocol, l) {

  def genExpr(e: Expr): Expr =
    if (e >= l) Omega else e

  override def rebuild(g: G): List[S] =
    if (dangerous(g))
      List(RebuildStep(g.current.conf.map(genExpr)))
    else List()
}
```

Рис. 8: Graph rewrite rules: an implementation for single-result supercompilation

The MRSC toolkit exploits the fact that the language Scala integrates object-oriented and functional features. This enables supercompilers to be produced by reusing the components provided by the MRSC toolkit and to create specialized components for specific problem domains. Besides, if we have to implement several variations of a supercompiler their common parts can be easily shared. For example, Fig. 8 shows an implementation of a single-result supercompilation algorithm developed by Klimov [6, 11, 10]. This implementation has been produced by subclassing the multi-result supercompiler in Fig. 7, and as can be easily seen, most part of the code is shared between the two supercompilers.

Another useful feature of Scala are traits. Instead of implementing a supercompiler as a monolithic class, we could have separated the implementations of driving, rebuilding and the whistle into a number of trait. This approach gives an opportunity to define several variants of driving, rebuilding and the whistle, and then produce different variations of the supercompiler by trying various combinations of traits. But this topic is beyond the scope of the present work.

7 Reducing the search space by taking into account the specifics of the domain

The main drawback of the naïve multi-result supercompiler presented in Fig. 7 is that, when used for the verification of a protocol, it may consider thousands (or even millions) of graphs, thereby consuming considerable resources.

However the search space can be drastically reduced by taking into account the specifics of the problem domain. In the case of counter systems this is achieved by implementing the following optimizations [12].

- Filtering graphs of configurations, rather than residual programs. When

supercompilation is used for solving a "reachability problem" (e.g. proving the fact that unsafe states are unreachable), the graphs containing unsafe configurations can be discarded, without transforming them into residual programs.

- A graph containing an unsafe configuration can be immediately discarded, even if this graph is incomplete. Thereby the search space is reduced, because the supercompiler does not have to consider the graphs that would be derived from this graph. This optimization exploits the *monotonicity* of the predicate `unsafe`: if a configuration c is an instance of a configuration c', and if `unsafe c` is true, then `unsafe c'` is also true. In addition, during multi-result supercompilation, a configuration c, appearing in a graph, can be removed only by replacing c with a more general configuration c'. Hence, if a graph g contains an unsafe configuration, all graphs that could be derived from g would contain at least one unsafe configuration. Therefore, there is no point in trying to complete this graph.

- The number of graphs considered during supercompilation can be reduced by performing only "one-step" generalizations, which amount to replacing a single numeric component N in a configuration with the symbol ω. This is correct, because any generalization can be achieved by a sequence of one-step generalizations.

- If multi-result supercompilation is used for finding completed graphs of minimum size, there is no point in considering incomplete graphs that are too big. Namely, if the supercompiler have already found a complete graph (without unsafe configurations) whose size is `maxSize`, all incomplete graphs whose size is greater than `maxSize` can be discarded. This optimization exploits the *monotonicity* of multi-result supercompilation: the descendants of a graph g cannot be smaller in size than their parent g.

Fig. 9 shows the supercompiler for counter systems that has been produced from the supercompiler in Fig. 7 by implementing the aforementioned optimizations. Technically, the improved supercompiler is implemented as the class `FastMRCountersRules`, which is a subclass of `MRCountersRules`.

The main loop of the optimized supercompiler is shown in Fig. 10. Complete graphs are produced by the iterator `graphs` by demand. Since the goal is to find a graph of minimum size, the variable `minGraph` contains the smallest of the graphs that have been encountered.

Now let us consider the internals of the class `FastMRCountersRules`.

The variable `maxSize` holds the maximum size of graphs that are worth considering: if the supercompiler encounters a graph whose size exceeds `maxSize`, this graph is discarded (see the definition of the method `steps`).

The method `rebuild` is redefined: now, instead of considering all possible generalization (produced by the method `gens`), it only considers one-step generalizations (produced by the method `oneStepGens`).

```
package mrsc.counters

class FastMRCountersRules(protocol: Protocol, l: Int)
  extends MRCountersRules(protocol, l) {

  var maxSize: Int = Int.MaxValue

  override def drive(g: G): List[S] =
    for (AddChildNodesStep(ns) <- super.drive(g)
         if ns.forall(c =>!protocol.unsafe(c._1)))
      yield AddChildNodesStep(ns)

  override def rebuild(g: G): List[S] =
    for (c <- oneStepGens(g.current.conf) if !protocol.unsafe(c))
      yield RebuildStep(c): S

  override def steps(g: G): List[S] =
    if (protocol.unsafe(g.current.conf) || size(g) > maxSize)
      List()
    else
      super.steps(g)

  private def size(g: G) =
    g.completeNodes.size + g.incompleteLeaves.size
}
```

Figure 9: Graph rewrite rules: an optimized implementation for counter systems

```
    val rules = new FastMRCountersRules(protocol, l)
    val graphs = GraphGenerator(rules, protocol.start)

    var minGraph: SGraph[Conf, Unit] = null
    for (graph <- graphs) {
      val size = graphSize(graph)
      if (size < rules.maxSize) {
        minGraph = graph
        rules.maxSize = size
      }
    }
```

Figure 10: Optimized implementation of the main loop of multi-result supercompilation

All other modifications are related to detecting unsafe configurations: the goal is to detect unsafe configurations as soon as possible. This is achieved by applying the predicate `unsafe` to the following configurations.

- The current configuration (the method `steps`). This test is necessary, because, in principle, even the initial configuration may turn out to be unsafe. If the current configuration is unsafe, the graph is discarded.
- The configurations that are produced by a driving step (the method `drive`). If at least one of these configurations is unsafe, the driving step is not performed.
- The configurations that are produced by a one-step generalization of the current configuration (the method `rebuild`). The generalizations that lead to unsafe configurations are discarded.

As has been shown in [12], the above optimizations produce a considerable effect. For example, when verifying the `ReaderWriter` protocol, the naïve version of the supercompiler performs 24963661 graph building steps, while the optimized version – only 3213.

Note that, although the implementation of the above optimizations is rather trivial, the correctness of the optimizations is based on taking into account a number of specifics of the problem domain, and exploiting some subtle properties of the supercompilation algorithm (such as the fact that only the leaves of a graph can be rebuild).

8 Conclusions

We have considered a simple domain-specific multi-result supercompiler for counter systems implemented by means of the MRSC toolkit. The implementation of the aforementioned supercompilers takes only a few dozen lines of code, which is achieved by simplifying the supercompilation algorithms and by using prefabricated components provided by the MRSC toolkit.

Thus it can be argued that the MRSC toolkit allows domain-specific multi-result supercompilers to be manufactured at low cost, thereby making them tools of practical value.

Acknowledgements

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

References

[1] 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.

[2] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Form. Methods Syst. Des.*, 23:257–301, November 2003.

[3] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[4] D. Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[5] N. D. Jones. The essence of program transformation by partial evaluation and driving. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of Systems Informatics, Third International Andrei Ershov Memorial Conference, PSI 1999, Akademgorodok, Novosibirsk, Russia July 6-9, 1999*, volume 1755 of *Lecture Notes in Computer Science*, pages 62–79. Springer, 2000.

[6] A. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In *PSI 11*, 2011.

[7] A. V. Klimov. An approach to supercompilation for object-oriented languages: the Java Supercompiler case study. In A. P. Nemytykh, editor, *First International Workshop on Metacomputation in Russia, Pereslavl-Zalessky, Russia, July 2–5, 2008*, pages 43–53. Ailamazyan University of Pereslavl, Pereslavl-Zalessky, 2008.

[8] A. V. Klimov. A Java Supercompiler and its application to verification of cache-coherence protocols. volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2010.

[9] 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.

[10] A. V. Klimov. Yet another algorithm for solving coverability problem for monotonic counter systems. In V. Nepomnyaschy and V. Sokolov, editors, *Second Workshop "Program Semantics, Specification and Verification: Theory and Applications", PSSV 2011, St. Petersburg, Russia, June 12–13, 2011*, pages 59–67. Yaroslavl State University, 2011.

[11] A. V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In Clarke et al. [1], pages 193–209.

[12] A. V. Klimov, I. G. Klyuchnikov, and S. A. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation. Preprint 19, Keldysh Institute of Applied Mathematics, 2012.

[13] I. G. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute of Applied Mathematics, 2011.

[14] I. G. Klyuchnikov and S. A. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In Clarke et al. [1], pages 210–226.

[15] A. P. Nemytykh. The supercompiler SCP4: General structure. In M. Broy and A. V. Zamulin, editors, *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003. Revised Papers*, volume 2890 of *Lecture Notes in Computer Science*, pages 162–170. Springer, 2003.

[16] A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: Experiments with a supercompiler. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 249–260, 1996.

[17] M. Odersky et al. *Programming in Scala*. Artima, 2nd edition, 2010.

[18] 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.

[19] M. H. Sørensen, R. Glück, and N. D. Jones. A positive supercompiler. *Journal of Functional Programming*, 6(6):811–838, 1996.

[20] V. F. Turchin. A supercompiler system based on the language Refal. *ACM SIGPLAN Not.*, 14(2):46–54, 1979.

[21] 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.

[22] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.

[23] V. F. Turchin. Supercompilation: Techniques and results. In 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*, pages 227–248. Springer, 1996.

[24] 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.