ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 24 за 2012 г.

Климов А.В., Ключников И.Г., Романенко С.А.

Реализация предметно-ориентированного многорезультатного суперкомпилятора с помощью инструментария MRSC

Рекомендуемая форма библиографической ссылки: Климов А.В., Ключников И.Г., Романенко С.А. Реализация предметно-ориентированного многорезультатного суперкомпилятора с помощью инструментария MRSC // Препринты ИПМ им. М.В.Келдыша. 2012. № 24. 21 с. URL: http://library.keldysh.ru/preprint.asp?id=2012-24

ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
ИМЕНИ М.В.КЕЛДЫША
РОССИЙСКАЯ АКАДЕМИЯ НАУК

Анд.В. Климов, И.Г. Ключников, С.А. Романенко

Реализация предметно-ориентированного многорезультатного суперкомпилятора с помощью инструментария MRSC

Москва
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.

Содержание

1 Введение

Суперкомпиляция [22, 23] – это метод преобразования программ, первоначально разработанный В.Ф. Турчиным для языка программирования Рефал (функциональный язык первого порядка с вызовом по имени) [21], и первые суперкомпиляторы разрабатывались и реализовывались для языка Рефал [20, 24, 16, 15].

В дальнейшем, была дана более абстрактная переформулировка суперкомпиляции, что позволило выяснить, какие части первоначальной формулировки были обусловлены спецификой языка Рефал, а какие – применимы и к другим языкам программирования [18, 19, 5]. В частности, была показана применимость суперкомпиляции для нефункциональных (императивных и объектно-ориентированных) языков программирования [7].

Многорезультатная суперкомпиляция – это метод конструирования суперкомпиляторов, выдающих не одну, а несколько остаточных программ [14, 9].

Другое направление развития – предметно-ориентированная суперкомпиляция для предметно-ориентированных языков, которая, как было показано в работе [12], в некоторых случаях может иметь некоторые преимущества по сравнению с универсальной суперкомпиляцией.

- Задания на суперкомпиляцию могут формулироваться на предметно-ориентированном языке, в терминах конкретной предметной области.
- Можно использовать упрощенный алгоритм суперкомпиляции.
- За счет использования знаний о предметной области можно значительно уменьшить вычислительные ресурсы, потребляемые многорезультатной суперкомпиляцией.

Однако, основным аргументом против предметно-ориентированной суперкомпиляции является то, что (теоретически), в случае использования универсальной суперкомпиляции, один и тот же суперкомпилятор может использоваться снова и снова для решения различных задач, в то время как при использовании предметно-ориентированной суперкомпиляции требуется создавать много специализированных суперкомпиляторов, каждый из которых имеет ограниченную область применения.

Таким образом, использование проблемно-ориентированной суперкомпиляции выглядит оправданным только в том случае, если затраты труда на разработку и реализацию проблемно-ориентированного суперкомпилятора значительно меньше, чем в случае универсального суперкомпилятора.

В данной работе мы рассматриваем структуру специализированного многорезультатного суперкомпилятора [14, 12], предназначенного для анализа счетчиковых систем [2, 7, 8, 6, 9, 11, 10], и его реализацию на основе инструментария MRSC [13]. Входным языком этого суперкомпилятора является проблемно-ориентированный язык, предназначенный для описания недетерминированных моделей коммуникационных протоколов. Реализация этого языка основана на “встраивании” [17, 4] и широком использовании конструкций высшего порядка. Рассматривается 2 версии многорезультатного суперкомпилятора. В первой версии реализован “наивный” алгоритм, который оказывается весьма неэффективным. Во второй версии учитываются особенности предметной области, что позволяет значительно уменьшить количество порождаемых графов конфигураций и снизить потребление вычислительных ресурсов.

Объем этой реализации – несколько десятков строк кода, что достигается благодаря использованию набора компонент, предоставляемых инструментарием MRSC [13].

Таким образом, можно утверждать, что при использовании инструментария MRSC, становится возможным “массовое производство” проблемно-ориентированных многорезультатных суперкомпиляторов, превращающее их из “предметов роскоши” в “товар массового потребления”.

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

Обозначения:

g – текущий граф конфигураций
β – активный лист строящегося графа конфигураций
c – конфигурация, находящаяся в узле β

Рис. 1: Схема многорезультатной суперкомпиляции

2 Многорезультатная суперкомпиляция – общая структура

Как было показано в работе [13], различные разновидности суперкомпиляции (детерминированная, недетерминированная и многорезультатная) могут быть описаны наборами правил переписывания. Правила, соответствующие многорезультатной суперкомпиляции, показаны на Рис. 1. Если имеется (недостроенный) граф конфигураций g, правила указывают, какие новые графы конфигураций могут быть порождены из графа g с помощью одного шага суперкомпиляции.

Таким образом, правила переписывания описывают некоторое *отношение* на множестве графов конфигурации. Однако, при создании суперкомпилятора, требуется разработать *алгоритм*, который получает на входе начальную конфигурацию, а на выходе выдает множество завершенных графов, соответствующих начальной конфигурации.

В инструментарии MRSC используется итеративный алгоритм, в процессе работы которого поддерживается некоторая коллекция U, содержащая незавершенные графы конфигураций. В начале работы в U, помещается начальный граф конфигураций, который состоит из единственного узла, содержащего начальную конфигурацию.

На каждом шаге работы алгоритма из коллекции U выбирается какой-то незавершенный граф g. Пусть $D = \{g_1, \ldots, g_n\}$ – это множество графов, получающихся их g путём применения одного из правил (Fold, Drive или Rebuild). Суперкомпилятор удаляет g из U, строит множество D и проверяет, какие из графов в D являются завершенными. С завершенными графами делать больше ничего не надо, поскольку они являются конечным результатом работы суперкомпилятора. Те же графы в D, которые являются незавершенными,

добавляются к коллекции U.

Если оказывается, что коллекция U пуста, работа суперкомпилятора на этом завершается.

Есть несколько тонких моментов в работе этого алгоритма, отличающих его от алгоритма, использующегося в классических однорезультатных суперкомпиляторах.

- Суперкомпилятор работает с коллекцией незавершенных графов, а не с единственным графом. При этом на каждом шаге суперкомпиляции один из графов "рождает потомков", но сам при этом "вымирает". В случае же однорезультатной суперкомпиляции имеется единственный граф конфигураций, который достраивается и преобразуется до тех пор пока не станет завершенным.

- После того, как из коллекции незавершенных графов U выбран некоторый граф g, делается попытка применить к нему каждое из правил Fold, Drive и Rebuild. При этом не все правила могут оказаться применимыми. В частности, если не применимо ни одно из правил, g просто "вымирает", не оставив потомков. Благодаря этому размер коллекции U уменьшается, что. в конечном счете, и обеспечивает завершаемость процесса суперкомпиляции.

- Правила Fold, Drive и Rebuild применяются независимо друг от друга. В частности, прогонка и обобщение могут выполняться одновременно к одному и тому же графу. В случае же однорезультатной суперкомпиляции, прогонка и обобщение взаимно исключают друг-друга.

- Свисток управляет применимостью правил, но не влияет на результат работы тех правил, которые применимы. Благодаря этому, в отличие от однорезультатной суперкомпиляции, свисток не должен заботиться о том, где, когда и как должно выполняться обобщение.

- Обобщение выполняется только для листьев графа конфигураций. В отличие от однорезультатной суперкомпиляции, не требуется выполнять *перестройку сверху* (откат к α), которая заключается в том, что всё поддерево, растущее из α, удаляется, а конфигурация c в α заменяется на более общую конфигурацию c'.

Таким образом, хотя это и кажется на первый взгляд парадоксальным, многорезультатная суперкомпиляция, в некоторых отношениях, устроена проще, чем однорезультатная суперкомпиляция.

3 Суперкомпиляция для счетчиковых систем

Одной из областей применения многорезультатной суперкомпиляции является верификация протоколов (кеш-когерентности и коммуникационных), пу-

тем анализа счетчиковых систем [2], являющихся моделями протоколов. Какие именно механизмы из числа используемых в универсальных суперкомпиляторах являются существенными для анализа счетчиковых систем? Этот вопрос был рассмотрен Климовым, предложившим несколько специализированных алгоритмов суперкомпиляции и для которых была доказана их корректность, завершаемость, а также их способность заведомо решать проблему достижимости для счетчиковых систем из некоторого класса [7, 8, 6, 9, 11, 10].

Выяснилось, что в случае счетчиковых систем возможны следующие упрощения алгоритма суперкомпиляции.

- Достаточно работать с конфигурациями вида $(a_1, \ldots, a_n)$, каждая компонента которых a_i является либо натуральным числом N, либо символом ω.
- При прогонке, при переходах от одной конфигурации к другой достаточно работать с проверками вида $e = N$ и $e \geq N$, где N - натуральное число, а e – арифметическое выражение, которое может содержать только операции $+$, $-$, натуральные числа и символ ω. Операции над символами ω выполняются следующим образом: $N < \omega =$ `True` and $\omega + N = \omega - N = \omega + \omega = \omega$.
- Все обобщения конфигурации c получаются путем замены некоторых компонент-чисел в c на ω.
- Для обеспечения завершаемости суперкомпиляции можно применить очень простой свисток: если одна из компонент конфигурации является числом n, и $n \geq l$, где l – константа, задаваемая при вызове суперкомпилятора, конфигурация объявляется “опасной”. Легко видеть, что для каждого фиксированного l множество “неопасных” конфигураций – конечно.

В последующих разделах мы опишем реализацию предметно-ориентированного многорезультатного суперкомпилятора для счетчиковых систем с помощью которого были получены минимальные графы конфигураций для нескольких протоколов [12]. Эта реализация основана на использовании компонент предоставляемых инструментарием MRSC `[13]`. Исходные тексты инструментария MRSC и суперкомпилятора доступны по адресу `https://github.com/ilya-klyuchnikov/mrsc`.

4 Реализация проблемно-ориентированного входного языка

Мы рассматриваем предметно-ориентированный суперкомпилятор, который принимает на входе программу на предметно-ориентированном языке (DSL),

```
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
  }
}
```

Рис. 2: Модель протокола MSI в виде DSL-программы

```
package mrsc

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

Рис. 3: Package `mrsc.counters`: определения и неявные преобразования

предназначенный для описания моделей протоколов. На Рис. 2 показано, как на этом DSL записывается модель протокола MSI. Другие примеры моделей протоколов и объяснение смысла программ на этом DSL можно найти в работе [12].

Отметим, что внешний вид программ, написанных на данном DSL, – не идеален, поскольку некоторые конструкции выглядят громоздко. Например, каждое правило начинается с комбинации `case List(i, m, s) if`, которую можно было бы сократить, например, до `(i, m, s) |`. Можно было бы сделать программу еще короче, объявив список переменных `(i, m, s)` только один раз, и не повторять его в начале каждого правила. Однако, мы не стали заниматься этими усовершенствованиями именно потому, что при их реализации в рамках языка Scala [17] не возникает никаких проблем: достаточно применить несколько стандартных приемов [4]. Наша же задача в данный момент – разобраться, как реализуется специализированный суперкомпилятор с помощью инструментария MRSC. Поэтому, мы постарались минимизиро-

```
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
}
```

Рис. 4: Реализация DSL

вать те части суперкомпилятора, которые не относятся к суперкомпиляции как таковой.

В результате получилось, что реализация DSL сведена к абсолютному минимуму: состоит из нескольких строчек кода, показанных на Рис. 3. А именно, “конфигурация” – это список “выражений”, а “правило перехода” – это частичная функция, отображающая одну конфигурацию в другую. Почему

функция частична? Потому, что правило перехода может быть не применимо к некоторым конфигурациям, и считается, что в этом случае функция не определена. При этом, если правило применимо, то оно порождает ровно одну конфигурацию, в которую происходит переход. При этом, в данном суперкомпиляторе конфигурации не содержат переменных, поэтому, на дугах графа конфигураций не требуется записывать какие-либо условия на значения переменных.

Мы воспользовались тем, что в языке Scala, конструкция вида `case p if c => e`, где `p` – образец, `c` – условие, а `e` – выражение, является записью частичной функции и означает, что если аргумент функции сопоставим с образцом `p` и выполнено условие `c`, то результатом функции является результат вычисления выражения `e`. Если же аргумент не сопоставим с образцом `p`, или не выполнено условие `c`, то для данного значения аргумента функция не определена. Таким образом, правило перехода может быть записано как `case p if c => e`.

Полное задание на суперкомпиляцию состоит из следующих частей.

- `start` – начальная конфигурация.
- `rules` – список правил перехода .
- `unsafe` – предикат, который определяет, является ли конфигурация "ненадежной".

Предикат `unsafe` является предметно-ориентированной частью задания на суперкомпиляцию, и служит для проверки, содержит ли построенный граф конфигураций ненадежные конфигурации?

На Рис. 4 показано формальное определение понятия "модель протокола" в виде трейта `Protocol`. Каждая такая модель является заданием на суперкомпиляцию, но, с другой стороны, может рассматриваться и как программа на предметно-ориентированном языке (DSL).

Интересно отметить, что DSL-программа не является сущностью первого порядка (как это неявно предполагается в классической формулировке проекций Футамуры [3]), а представляет собой смесь из сущностей высшего порядка (функций) и сущностей первого порядка (списков, чисел). Этот подход близок к методу реализации предметно-ориентированных языков (DSL) известному под именем "встраивания" [4]. Благодаря тому, что встроенный DSL наследует большую часть своих конструкций от принимающего (host) языка Scala, его реализация и получается столь тривиальной (примерно в 10 строк текста).

Конечно, справедливости ради, следует отметить, что реализация DSL использует "выражения" (`Expr`), реализация которых будет рассмотрена в следующем разделе (и занимает ещё примерно 20 строк). Однако, выражения используются не только в DSL-программах, но и в качестве компонент конфигураций.

5 Операции над конфигурациями

Как было сказано в разделе 3, каждая конфигурация имеет вид $(a_1, \ldots, a_n)$, где каждая компонента которых a_i является либо натуральным числом N, либо символом ω. В реализации на языке Scala, такие конфигурации представлены списками "выражений", являющимися значениями типа `Expr` (см. Рис. 4).

Тип `Expr` определен как трейт, из которого выводится класс `Num`, представляющий натуральные числа и объект `Omega`, представляющий символ ω. Над значениями типа `Expr` можно выполнять операции + (сложение), - вычитание, >= (сравнение $\geq$) и === (сравнение на равенство $=$). Для сравнения на равенство используется знак операции ===, поскольку == и = используются в программах на языке Scala для других целей.

Заметим, что когда одним из операндов является символ ω (изображающий произвольное натуральное число), в качестве результата операции выдается некоторая "аппроксимация сверху". Например, считается, что $\omega \geq N$ истинно для любого натурального числа N. Действительно, проверки такого вида используются для того, чтобы определить, применимо ли некоторое правило? Поскольку ω изображает произвольное число N', а $N' \geq N$ истинно для некоторых N', в некоторых случаях правило является применимым. Поэтому мы и считаем, что $\omega \geq N$ истинно.

Таким образом, теперь мы имеем полное определение входного предметно-ориентированного языка для записи заданий на суперкомпиляцию (см. Рис. 3, 4). При этом, заодно определяется и язык, на котором записываются конфигурации, и операции над компонентами конфигураций, через которые реализуется прогонка.

Кроме прогонки, в процессе суперкомпиляции требуется выполнять над конфигурациями еще две операции: распознавание того, что конфигурация c_1 является частным случаем конфигурации c_2, и построение множества всех возможных обобщений для заданной конфигурации c. Реализация этих операций показана на Рис. 5.

Функция `instanceOf` проверяет, что конфигурация `c1` является частным случаем конфигурации `c2`.

Функция `genExpr` выдает множество всех возможных обобщений для заданного выражения (компонента конфигурации). При этом само исходное выражение включается в множество обобщений. Множество обобщений для символа ω состоит из символа ω, а множество обобщений для числа N содержит два элемента: N и ω.

Функция `gens` строит множество всех возможных обобщений для заданной конфигурации, при этом сама конфигурация в это множество не включается.

Функция `oneStepGens` строит для заданной конфигурации множество всех обобщений, которые получаются обобщением только одной из компонент конфигурации. Эта функция используется в оптимизированном варианте много-

```
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)
  }
}
```

Рис. 5: Операции над конфигурациями: распознавание частных случаев и построение обобщений

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

Рис. 6: “Строительный раствор” для построения суперкомпиляторов, предоставляемый инструментарием MRSC

результатного суперкомпилятора, который будет описан в разделе 7.

6 Построитель графов конфигураций

6.1 От правил переписывания к алгоритму

Технически, всякий суперкомпилятор, реализованный с помощью инструментария MRSC [13], использует две компоненты: `GraphRewriteRules` и `GraphGenerator` (см. Рис. 6).

Трейт `GraphRewriteRules` содержит объявление метода `steps`, который используется в главном цикле суперкомпиляции для получения всех графов, которые могут быть произведены из недостроенного графа g путем применения правил переписывания Fold, Drive и Rebuild, показанных на Рис. 1. А именно, `steps(`g`)` возвращает список возможных "шагов переписывания графа" [13]. После чего, генератор графов применяет каждый из этих "шагов" к графу g получая набор потомков графа g.

Каждый конкретный суперкомпилятор должен содержать свою реализацию метода `steps`. Класс `GraphGenerator`, напротив, представляет собой законченную компоненту, готовую для немедленного использования, и является составной частью любого суперкомпилятора, построенного на основе инструментария MRSC.

В случае суперкомпиляции для счетчиковых систем, метод `steps` реализуется довольно просто (см. Рис. 7).

Методы `fold`, `drive` и `rebuild` соответствуют правилам переписывания Fold, Drive и Rebuild (Рис. 1). Поскольку правила переписывания не зависят друг от друга, тело метода (steps) можно было бы реализовать следующим тривиальным способом:

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

Однако, мы немного оптимизировали реализацию метода, учитывая тот факт, что правила Drive и Rebuild применимы только в случаях, когда неприменимо правило Fold. Другая тонкость состоит в том, что правило Fold является недетерминированным, ибо текущая конфигурация может оказаться свертываемой сразу к нескольким другим конфигурациям в графе. Однако же, в случае счетчиковых систем, различия между возможными вариантами свертки несущественны. Поэтому, в реализации на Рис. 7 метод `fold` выдает только один из возможных вариантов свертки, в связи с чем типом его результата является не `List[S]`, а `Option[S]`. Попытка же применить правила Drive и Rebuild делается только в тех случаях, когда `fold` не выдает ни одного варианта свертки.

Реализации методов `fold` и `rebuild` – "прямолинейна".

Метод `dangerous` реализует вариант свистка, предложенный Климовым [6, 11, 10]: конфигурация считается "опасной", если она содержит число N, такое, что $N \geq l$, где l – константа, которая задается в качестве входного параметра при запуске суперкомпилятора.

Реализация метода `drive` использует вспомогательный метод `next`, который пытается применять все правила переходов к конфигурации c. Если

```
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: Реализация многорезультатного построителя графов конфигураций

какое-то правило применимо, оно выдает конфигурацию c', и в список, выдаваемый методом **next**, включается пара $(c', ())$. В общем случае, такая пара может иметь вид (c', d), где c' – новая конфигурация, а d – метка, которой должна быть помечена дуга, идущая к узлу, содержащему c'. Однако, в случае счетчиковых систем, дуги не несут никакой информации, и мы выдаем “пустышку” () в качестве второго элемента пары.

6.2 Функциональное программирование + ООП

Заметим, что инструментарий MRSC [13] обеспечивает инфраструктуру для написания реализаций суперкомпиляторов в “функциональном” стиле. Графы конфигураций никогда не “преобразуются”: недостроенный граф g производит потомков и “вымирает”. Однако, реализация устроена таким образом, что потомки графа g совместно используют общие части, унаследованные от графа g. Благодаря этому суперкомпилятор способен иметь дело с тысячами графов одновременно.

Таким образом, в случае многорезультатной суперкомпиляции, функциональный подход имеет определенные преимущества перед императивным подходом, основанным на языке SCPL, предложенным Турчиным [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: Реализация однорезультатного построителя графов конфигураций

Язык Scala предоставляет как объектно-ориентированные, так и функциональные изобразительные средства, и это используется в инструментарии MRSC, поскольку позволяет создавать суперкомпиляторы как повторно используя готовые компоненты, так и создавая специализированные компоненты для специфических областей. Кроме того, если требуется реализовать несколько модификаций какого-то суперкомпилятора, это легко сделать так, чтобы они совместно использовали их общие части. Например, на Рис. 8 показана реализация однорезультатного алгоритма суперкомпиляции, разработанного Климовым [6, 11, 10]. Эта реализация была получена с помощью определения подклассов для классов, реализующих многорезультатный суперкомпилятор, показанный на Рис. 7. Легко видеть, что основная часть кода является общей для двух суперкомпиляторов.

Другой полезной возможностью языка Scala являются трейты (traits). Вместо того, чтобы реализовывать суперкомпилятор как единственный монолитный класс, мы могли бы выделить реализации прогонки, перестройки и свистка в отдельные трейты. При этом появляется возможность реализо-

вать несколько вариантов свистка, перестройки и свистка в виде нескольких трейтов, а затем получать различные варианты суперкомпилятора, собирая вместе различные комбинации трейтов. Однако, мы не будем подробно останавливаться на этом вопросе в данной работе.

7 Сокращение перебора за счет учета специфики предметной области

Основным недостатком "наивного" многорезультатного суперкомпилятора, показанного на Рис. 7 является то, что в процессе верификации протокола он зачастую рассматривает тысячи (или даже – миллионы) графов, потребляя при этом значительные вычислительные ресурсы.

Однако перебор можно значительно сократить встроив некоторые знания о предметной области в предметно-ориентированный суперкомпилятор. В случае счетчиковых систем, это можно сделать реализовав следующие оптимизации [12].

- Можно фильтровать графы конфигураций, а не остаточные программы. Если суперкомпиляция используется для решения "проблемы достижимости" (доказательства того, что ненадежные состояния недостижимы), можно сразу отбрасывать те графы, которые содержат ненадежные конфигурации, не пытаясь преобразовывать их в остаточные программы.

- Можно отбрасывать незавершенные графы, в которых появилась хотя бы одна ненадежная конфигурация. Это сокращает пространство поиска, поскольку отбрасывается не только рассматриваемый граф, но и всё множество графов, которые могли бы быть из него порождены. Эта оптимизация основана на том, что предикат `unsafe` обязан обладать свойством *монотонности*: если `unsafe` c выполнено для некоторой конфигурации c, то для любой конфигурации c', являющейся обобщением c, тоже выполнено `unsafe` c'. При этом процесс многорезультатной суперкомпиляции устроен таким образом, что конфигурация c может исчезнуть из графа только в результате её замены на более общую конфигурацию c'. Таким образом, если в графе g появилась ненадежная конфигурация, достраивать g не имеет смысла, ибо во всех полученных из g графах будет содержаться хотя бы одна ненадежная конфигурация.

- Можно сократить количество рассматриваемых графов, если рассматривать для заданной конфигурации c не всё множество её возможных обобщений, а только "одношаговые" обобщения c', которые получаются заменой в c только одной компоненты-числа N на ω. Это возможно благодаря тому, что любое обобщение конфигурации может быть построено в результате выполнения последовательности одношаговых обобщений.

```
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
}
```

Рис. 9: Реализация оптимизированного многорезультатного построителя графов конфигураций

```
    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
      }
    }
```

Рис. 10: Реализация главного цикла оптимизированного многорезультатного суперкомпилятора

- При поиске завершенных графов минимального размера, можно отбрасывать незавершенные графы имеющие слишком большой размер. А именно, если уже был найден завершенный граф (не содержащий ненадежные конфигурации) размера `maxSize`, то можно отбрасывать все незавершенные графы, размер которых превышает `maxSize`. Эта оптимизация тоже основана на свойствах монотонности: при достраивании незавершенного графа g получаются графы, размер которых не меньше, чем размер графа g.

На Рис. 9 показан суперкомпилятор для счетчиковых систем, реализующий вышеописанные оптимизации. Этот суперкомпилятор был получен путем внесения изменений в "наивный" суперкомпилятор, показанный на Рис. 7. Технически, усовершенствованный суперкомпилятор реализован в виде класса `FastMRCountersRules`, являющегося подклассом класса `MRCountersRules`.

Основной цикл оптимизированного суперкомпилятора показан на Рис. 10. Завершенные графы порождаются итератором `graphs` по запросу. Поскольку цель суперкомпилятора – найти граф минимального размера, переменная `minGraph` содержит граф, который является минимальным среди уже построенных графов.

Теперь рассмотрим внутреннее устройство класса `FastMRCountersRules`.

Переменная `maxSize` содержит максимальный допустимый размер графов: если в процессе работы получается граф, размер которого превышает `maxSize`, этот граф отбрасывается (см. определение метода `steps`).

Метод `rebuild` переопределен таким образом, что рассматриваются не все возможные обобщения конфигурации (порождаемые функцией `gens`), а только одношаговые обобщения (порождаемые функцией `oneStepGens`).

Остальные изменения связаны с обнаружением ненадежных конфигураций: такие конфигурации обнаруживаются сразу же, в тот момент, когда они порождаются. Для этого предиката `unsafe` вызывается в нескольких местах и применяется к следующим конфигурациям.

- Проверяется надежность текущей конфигурации (метод `steps`). Эта проверка необходима, поскольку ненадежной может оказаться даже начальная конфигурация. Если текущая конфигурация ненадежна, граф отбрасывается.

- Проверяются все конфигурации, которые получаются в результате выполнения шага прогонки (метод `drive`). Если хотя бы одна из этих конфигураций оказывается ненадежной, прогонка к графу не применяется.

- Проверяются конфигурации, которые получаются в результате (одношагового) обобщения текущей конфигурации, и те конфигурации, которые оказываются ненадежными – отбрасываются.

Как было показано в [12], вышеупомянутые оптимизации могут давать ощутимый эффект. Например, при верификации протокола $ReaderWriter$

“наивная” версия суперкомпилятора выполняет 24963661 элементарных операций по построению графов, а оптимизированная версия – только 3213 шагов.

Еще раз отметим, что хотя реализация вышеописанных оптимизаций чрезвычайно проста, возможность выполнения этих оптимизаций (и их корректность) основаны не учете некоторых тонких свойств предметной области, а также некоторых свойств алгоритма многорезультатной суперкомпиляции (например того факта, что перестройке могут подвергаться только листья дерева).

8 Заключение

В данной работе был рассмотрен специализированный многорезультатный суперкомпилятор, предназначенный для анализа поведения счетчиковых систем. Реализация суперкомпилятора была выполнена на основе инструментария MRSC, и занимает всего несколько десятков строк кода.

Краткость достигается за счет упрощений алгоритма суперкомпиляции, основанных на учете особенностей предметной области, и за счет использования набора компонент, предоставляемого инструментарием MRSC.

Таким образом, было показано, что, при использовании инструментария MRSC, изготовление проблемно-ориентированных многорезультатных суперкомпиляторов становится делом малозатратным, а стало быть – вполне целесообразным с практической точки зрения.

Благодарности

Авторы выражают признательность участникам Рефал-семинаров, проводимых в ИПМ им. М.В. Келдыша, за ценные замечания и плодотворные обсуждения этой работы.

Список литературы

[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. URL: http://library.keldysh.ru/preprint.asp?id=2012-19.

[13] I. G. Klyuchnikov and S. A. Romanenko. MRSC: a toolkit for building multi-result supercompilers. Preprint 77, Keldysh Institute of Applied Mathematics, 2011. URL: http://library.keldysh.ru/preprint.asp?lg=e&id=2011-77.

[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.