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

Препринты ИПМ • Препринт № 19 за 2012 г.

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

Автоматизированная верификация счетчиковых систем посредством предметно-ориентированной многорезультатной суперкомпиляции

Рекомендуемая форма библиографической ссылки: Климов А.В., Ключников И.Г., Романенко С.А. Автоматизированная верификация счетчиковых систем посредством предметно-ориентированной многорезультатной суперкомпиляции // Препринты ИПМ им. М.В.Келдыша. 2012. № 19. 42 с. URL: http://library.keldysh.ru/preprint.asp?id=2012-19

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

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

Автоматизированная верификация счетчиковых систем посредством предметно-ориентированной многорезультатной суперкомпиляции

Москва
2012

Andrei V. Klimov, Ilya G. Klyuchnikov, Sergei A. Romanenko. Automatic verification of counter systems via domain-specific multi-result supercompilation

We consider an application of supercompilation to the analysis of counter systems. Multi-result supercompilation enables us to find the best versions of the analysis by generating a set of possible results that are then filtered according to some criteria. Unfortunately, the search space may be rather large. However, the search can be drastically reduced by taking into account the specifics of the domain. Thus, we argue that a combination of domain-specific and multi-result supercompilation may produce a synergetic effect. Low-cost implementations of domain-specific supercompilers can be produced by using prefabricated components provided by the MRSC toolkit.

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. Автоматизированная верификация счетчиковых систем посредством предметно-ориентированной многорезультатной суперкомпиляции

Рассматривается применение суперкомпиляции к анализу поведения счетчиковых систем. Многорезультатная суперкомпиляция позволяет обнаруживать наилучшие варианты анализа, благодаря тому, что порождается множество возможных результатов анализа, которое затем фильтруется в соответствии с некоторыми критериями. К сожалению, пространство поиска при этом может получаться весьма обширным. Однако, можно значительно уменьшить объем поиска за счет учета особенностей предметной области. Таким образом, сочетание предметно-ориентированной и многорезультатной суперкомпиляции может давать синергетический эффект. Затраты на реализацию предметно-ориентированных многорезультатных суперкомпиляторов могут быть невелики, если использовать компоненты, предоставляемые инструментарием MRSC.

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 – это метод преобразования программ, первоначально разработанный В.Ф. Турчиным для языка программирования Рефал (функциональный язык первого порядка с вызовом по имени) [40], и первые суперкомпиляторы разрабатывались и реализовывались для языка Рефал [38, 42, 31].

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 [35, 36, 5]. В частности, была показана применимость суперкомпиляции для нефункциональных (императивных и объектно-ориентированных) языков программирования [8].

Также следует отметить, что хотя изначально суперкомпиляция рассматривалась В.Ф. Турчиным и как средство оптимизации, и как средство анализа программ [39], работы в области суперкомпиляции долгое время были,

главным образом, посвящены применениям суперкомпиляции для оптимизации программ. В последнее время, однако, наблюдается возвращение интереса к применению суперкомпиляции в качестве средства выявления и доказательства свойств программ [28, 14, 13].

Многорезультатная суперкомпиляция – это метод конструирования суперкомпиляторов, выдающих не одну, а несколько остаточных программ [16, 11].

Цель настоящей работы – показать на конкретном примере, что многорезультатная, проблемно-ориентированная суперкомпиляция не является теоретическим курьезом, а представляет практический интерес, и, в силу следующих причин, может иметь определенные преимущества по сравнению с универсальной однорезультатной (детерминированной) суперкомпиляцией.

- Входным языком универсального суперкомпилятора должен быть некоторый универсальный язык, не "заточенный" под конкретную область применения. Вследствие чего, для каких-то областей применения он может быть избыточно сложен, но в то же время чего-то в нём может и не хватать.

- В случае использования суперкомпиляции для задач анализа и верификации, остро встаёт проблема надёжности и корректности самого суперкомпилятора. Насколько можно доверять сложному и громоздкому универсальному суперкомпилятору?

- В случае проблемно-ориентированной суперкомпиляции, входном языке суперкомпилятора могут предоставляться операции и управляющие конструкции относящиеся к рассматриваемой предметной области, и обладающие заранее известными математическими свойствами. Знание об этих свойствах может быть "встроено" в специализированный суперкомпилятор, давая возможность подвергать программы более глубокому анализу и преобразованию, чем в случае использования "чистой" суперкомпиляции.

- Ограниченность возможностей специализированного входного языка часто позволяет резко упростить некоторые части суперкомпилятора. Например, для многих приложений не нужно работать с конфигурациями, содержащими вложенные вызовы функций. А упрощение суперкомпилятора повышает его надёжность, и облегчает применение формальных методов для проверки корректности суперкомпилятора (как было показано Крустевым [17]).

- Если реализовывать предметно-ориентированные суперкомпиляторы на основе набора заранее заготовленных компонент (например, используя инструментарий MRSC [16, 15]), затраты труда на изготовление специализированного суперкомпилятора могут быть на порядок меньше, чем в случае реализации универсального суперкомпилятора.

По адресу `https://github.com/ilya-klyuchnikov/mrsc` можно найти исходные тексты инструментария MRSC и суперкомпиляторов, рассматриваемых в данной работе.

2 Анализ поведения систем с помощью суперкомпиляции

Одним из способов анализа поведения систем является создание их моделей в виде программ. В результате, задача исследования поведения исходной системы сводится к задаче анализа свойств некоторой программы.

При этом, для анализа программ может применяться *трансформационный подход*, при котором исходная программа p преобразуется в другую программу p' (эквивалентную исходной программе p). В результате, некоторые скрытые свойства исходной программы p могут стать очевидными в преобразованной программе p'.

Например, предположим, что исходная программа p имеет сложную структуру и содержит предложения вида `return False`. Тогда ответ на вопрос, может ли она выдавать `False` – неочевиден. Предположим, что в результате преобразования программы p получилась тривиальная программа p', тело которой состоит из одного предложения `return True`. Тогда можно сделать вывод, что p' никогда не выдает `False`. А поскольку p' эквивалентна p, из этого следует, что и p никогда не выдает `False`.

Начальные состояния:

$$(i, 0, 0, 0)$$

Переходы:

$$\begin{array}{lcl}
(i, e, s, m) \mid i \geq 1 & \longrightarrow & (i - 1, 0, s + e + m + 1, 0) \\
(i, e, s, m) \mid e \geq 1 & \longrightarrow & (i, e - 1, s, m + 1) \\
(i, e, s, m) \mid s \geq 1 & \longrightarrow & (i + e + s + m - 1, 1, 0, 0) \\
(i, e, s, m) \mid i \geq 1 & \longrightarrow & (i + e + s + m - 1, 1, 0, 0)
\end{array}$$

Ненадежные состояния:

$$\begin{array}{l}
(i, e, s, m) \mid m \geq 2 \\
(i, e, s, m) \mid s \geq 1 \wedge m \geq 1
\end{array}$$

Рис. 1: Протокол MESI: модель протокола в виде счетчиковой системы

Одной из областей применения трансформационного подхода является анализ счетчиковых систем [3], используемых в качестве моделей коммуникационных протоколов. Рассмотрим модель протокола MESI в виде счетчиковой системы, неформальное описание которой дано на Рис. 1.

Состояния системы изображаются четверками натуральных чисел. Спецификация системы состоит из описания множества начальных состояний, и набора правил перехода из состояния в состояние, имеющих вид

$$(i, e, s, m) \,|\, p \longrightarrow (i', e', s', m')$$

где i, e, s, m – переменные, p – условие на значения переменных, при выполнении которого возможен переход, а i', e', s', m' – выражения, в которых могут использоваться переменные i, e, s, m.

Несколько правил могут быть применимы одновременно, т.е. система является недетерминированной.

Спецификация модели протокола также содержит описание множества ненадежных (unsafe) состояний. Целью анализа модели протокола является решение *проблемы достижимости*: требуется доказать, что множество состояний, достижимых из начальных состояний, не содержит ненадежные состояния.

Как показали Лёйшель и Леман [23, 20, 21, 18], задачи достижимости для такого рода систем могут решаться с помощью специализации программ. При этом анализируемая система может описываться программой на специализированном языке (DSL) [23], а затем, с помощью первой проекции Футамуры [4], преобразовываться в программу на Прологе посредством применения классического частичного вычислителя LOGEN [7, 19]. После этого программа на Прологе анализируется с помощью специализатора ECCE [24, 22], близкого по своему внутреннему устройству к суперкомпиляторам.

Лисица и Немытых [27, 28, 29], осуществили верификацию нескольких коммуникационных протоколов используя суперкомпилятор SCP4 [31, 30, 25]. Входным языком суперкомпилятора SCP4 является Рефал, функциональный язык первого порядка, разработанный Турчиным [39]. При этом SCP4 является развитием более ранних суперкомпиляторов для языка Рефал [38, 39, 42, 40, 41].

Позднее эти же протоколы были верифицированы и с помощью суперкомпилятора JScp, обрабатывающего программы на языке Java [9, 10], а также с помощью некоторых других суперкомпиляторов.

Согласно подходу Лисицы и Немытых, модели протоколов должны формализовываться в виде Рефал-программ. Например, протокол MESI [3, 25, 26] кодируется в виде Рефал-программы, показанной на Рис. 2. Программа написана таким образом, что в случае достижения ненадежного состояния, она прекращает работу и выдает символ `False`.

После обработки этой программы суперкомпилятором SCP4 получается остаточная программа, показанная на Рис. 3, которая не содержит ни одного вхождения символа `False`. Это наводит на мысль, что эта программа не может выдавать `False`. Однако, это рассуждение было бы безупречным только в случае статически типизированного языка программирования. В случае же языка с динамической типизацией (каковыми являются языки Лисп, Scheme

```
*$MST_FROM_ENTRY;
*$STRATEGY Applicative;
*$LENGTH 0;

$ENTRY Go {e.A (e.I) =
  <Loop (e.A) (Invalid e.I)(Modified )(Shared )(Exclusive ) >;}

Loop {
  () (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    <Result (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4)>;
  (s.A e.A) (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
    <Loop (e.A)
    <RandomAction s.A
    (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4)>>;
}

RandomAction {
* rh Trivial
* rm
A (Invalid s.1 e.1) (Modified e.2) (Shared e.3) (Exclusive e.4) =
   (Invalid e.1)  (Modified )  (Shared s.1 e.2 e.3 e.4 ) (Exclusive );
* wh1 Trivial
*wh2
B (Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive s.4 e.4) =
  (Invalid e.1)(Modified s.4  e.2)(Shared e.3)(Exclusive e.4);
* wh3
C (Invalid e.1)(Modified e.2)(Shared s.3 e.3)(Exclusive e.4) =
  (Invalid e.4 e.3 e.2 e.1)(Modified  )(Shared )(Exclusive s.3);
* wm
D (Invalid s.1 e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) =
  (Invalid e.4 e.3 e.2 e.1)(Modified  )(Shared )(Exclusive s.1);
}

Result{
(Invalid e.1)(Modified s.2 e.2)(Shared s.3 e.3)(Exclusive e.4) = False;
(Invalid e.1)(Modified s.21 s.22 e.2)(Shared e.3)(Exclusive e.4) = False;

(Invalid e.1)(Modified e.2)(Shared e.3)(Exclusive e.4) = True;
}
```

Рис. 2: Протокол MESI: модель протокола в виде Рефал-программы

и Рефал), программа, в принципе, может выдать `False` даже если он не содержится в тексте программы, поскольку `False` может быть получен программой через входные данные и затем передан на выход. В действительности, такого рода “инженерные решения” популярны среди хакеров, как средство нападе-

```
* InputFormat: <Go e.41 >
$ENTRY Go {
 (e.101 )  = True ;
 A e.41 (s.103 e.101 )  = <F24 (e.41 ) (e.101 ) s.103 > ;
 D e.41 (s.104 e.101 )  = <F35 (e.41 ) (e.101 ) s.104 > ;
}

* InputFormat: <F35 (e.109 ) (e.110 ) s.111 e.112 >
F35 {
 () (e.110 ) s.111 e.112  = True ;
 (A e.109 ) (e.110 ) s.111 s.118 e.112  =
    <F24 (e.109 ) (e.112 e.110 ) s.118 s.111 > ;
 (A e.109 ) (s.119 e.110 ) s.111  = <F24 (e.109 ) (e.110 ) s.119 s.111 >;
 (B ) (e.110 ) s.111 e.112  = True ;
 (B A e.109 ) (e.110 ) s.111 s.125 e.112  =
    <F24 (e.109 ) (e.112 e.110 ) s.125 s.111 > ;
 (B A e.109 ) (s.126 e.110 ) s.111  =
    <F24 (e.109 ) (e.110 ) s.126 s.111> ;
 (B D e.109 ) (e.110 ) s.111 s.127 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.127 > ;
 (B D e.109 ) (s.128 e.110 ) s.111  =
    <F35 (e.109 ) (s.111 e.110 ) s.128> ;
 (D e.109 ) (e.110 ) s.111 s.120 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.120 > ;
 (D e.109 ) (s.121 e.110 ) s.111  = <F35 (e.109 ) (s.111 e.110 ) s.121 >;
}

* InputFormat: <F24 (e.109 ) (e.110 ) s.111 e.112 >
F24 {
 () (e.110 ) s.111 e.112  = True ;
 (A e.109 ) (s.114 e.110 ) s.111 e.112  =
    <F24 (e.109 ) (e.110 ) s.114 s.111 e.112 > ;
 (C e.109 ) (e.110 ) s.111 e.112  =
    <F35 (e.109 ) (e.110 ) s.111 e.112 >;
 (D e.109 ) (s.115 e.110 ) s.111 e.112  =
    <F35 (e.109 ) (s.111 e.112 e.110) s.115 > ;
}
```

Рис. 3: Протокол MESI: остаточная Рефал-программа

ния на веб-приложения [37]. К счастью, существуют достаточно простые методы анализа программ, которые позволяют вычислять аппроксимации сверху для множеств значений функций даже в случае языков с динамической типизацией [6]. И эти методы способны справляться с анализом программ, подобных той, что показана на Рис. 3.

3 Предметно-ориентированная суперкомпиляция как средство анализа

3.1 Недостатки универсальной суперкомпиляции

Очевидным достоинством универсальной суперкомпиляции является именно её универсальность: разработав и реализовав суперкомпилятор один раз, мы можем, применять его снова и снова для решения разнообразных задач (теоретически – не затрачивая дополнительных усилий). Однако, у универсальной суперкомпиляции есть и недостатки. Рассмотрим их на примере применения суперкомпилятора SCP4 для анализа счетчиковых систем [30, 27, 28, 29]. В этом случае задания на суперкомпиляцию формулируются в виде Рефал-программ [25, 26], что приводит к следующим неудобствам.

- Натуральные числа во входных программах изображаются последовательностями символов, а сложение чисел выполняется посредством конкатенации. Такое представление чисел выбрано для того, чтобы надлежащим образом сработали универсальные алгоритмы, заложенные в SCP4 (свисток, обобщение), которые ничего не знают про специфику задачи. Таким образом, получается, что представление данных приходится выбирать исходя не из специфики решаемой задачи, а учитывая тонкие особенности внутренней реализации SCP4.

- В исходные программы приходится добавлять (в виде комментариев) директивы для суперкомпилятора SCP4, управляющие его работой. Это является способом сообщить SPC4 некоторую информацию о специфике решаемой задачи, и без этой информации остаточные программы, порождаемые SCP4 не обладают желательными свойствами. Однако, для добавления правильных директив требуется понимание внутреннего устройства SCP4.

- Остается неясным вопрос: насколько можно доверять результатам анализа счетчиковых систем, полученным с помощью SCP4? В силу сложности внутренних механизмов SCP4 и обширности его исходных текстов, задача верификации самого SCP4 представляется весьма сложной.

3.2 Специализированные алгоритмы суперкомпиляции

Какие именно механизмы, заложенные в SCP4, действительно существенны для анализа счетчиковых систем? Этот вопрос был исследован в работах [11, 12, 13], в которых были рассмотрены несколько специализированных алгоритмов суперкомпиляции и доказана их корректность, завершаемость, а также их способность гарантированно решать проблему достижимости для счетчиковых систем из некоторого класса.

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

- Конфигурации могут иметь более простую структуру, чем в случае суперкомпиляции для классических функциональных языков.
  - Нет вложенных вызовов функций в конфигурациях.
  - Отсутствуют повторные вхождения переменных.
  - Каждая конфигурация – это кортеж из нескольких компонент, содержащий фиксированное число компонент.
  - Компонента конфигурации – это либо натуральное число, либо переменная.
- Для обеспечения завершаемости суперкомпиляции можно применить комбинацию свистка и обобщения простого вида: если одна из компонент конфигурации является числом n, и $n \geq l$, где l – константа, задаваемая при вызове суперкомпилятора, компонента n заменяется на переменную (тем самым происходит обобщение конфигурации). Легко видеть, что для каждого фиксированного l множество возможных конфигураций – конечно.

3.3 Специализированные суперкомпиляторы для специализированных языков

Как выяснилось, простейший из известных специализированных алгоритмов суперкомпиляции для счетчиковых систем [12], может быть легко реализован, если использовать средства, предоставляемые инструментарием MRSC [16, 15]. Простота достигается за счёт нескольких факторов.

- Требуется реализовывать не универсальный алгоритм суперкомпиляции для сложного универсального языка, а упрощенный алгоритм суперкомпиляции для специализированного входного языка.
- Инструментарий MRSC реализован на языке Scala, предоставляющем мощные средства для реализации встроенных предметно-ориентированных языков (DSL). Причём, как на основе интерпретации (поверхностное встраивание), так и на основе компиляции (глубокое встраивание).
- MRSC предоставляет готовые компоненты для работы с графом конфигураций (добавления и удаления узлов), работы с множествами графов, печати графов. Требуется программировать только те части суперкомпилятора, которые зависят от входного языка и от структуры конфигураций.

```
object MESI extends Protocol {
  val start: Conf = List(Omega, 0, 0, 0)
  val rules: List[TransitionRule] = List(
    {case List(i, e, s, m) if i>=1 => List(i-1, 0, s+e+m+1, 0)},
    {case List(i, e, s, m) if e>=1 => List(i, e-1, s, m+1)},
    {case List(i, e, s, m) if s>=1 => List(i+e+s+m-1, 1, 0, 0)},
    {case List(i, e, s, m) if i>=1 => List(i+e+s+m-1, 1, 0, 0)})
  def unsafe(c: Conf) = c match {
    case List(i, e, s, m) if m>=2 => true
    case List(i, e, s, m) if s>=1 && m>=1 => true
    case _ => false
  }
}
```

Рис. 4: Протокол MESI: модель протокола в виде DSL-программы

```
package object counters {
  type Conf = List[Expr]
  type TransitionRule = PartialFunction[Conf, Conf]
  ...
}

sealed trait Expr { ... }

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

Рис. 5: Реализация DSL для описания счетчиковых систем: базовые понятия

Если мы разрабатываем специализированный алгоритм суперкомпиляции для задач из какой-то области, то вполне логично выбрать в качестве входного языка суперкомпилятора не универсальный язык, а предметно-ориентированный язык.

Задания для специализированного суперкомпилятора записываются на предметно-ориентированном языке, и выглядят почти так же компактно и естественно, как и их неформальные описания. В качестве примера можно сравнить три спецификации протокола MESI: неформальное (Рис. 1), в виде Рефал-программы (Рис. 2) и на специализированном языке (Рис. 4).

DSL-программа, описывающая протокол, представляет собой объект, реализующий трейт (абстрактный класс) `Protocol` (Рис. 5). При этом программа не является константой первого порядка (как подразумевается при примене-

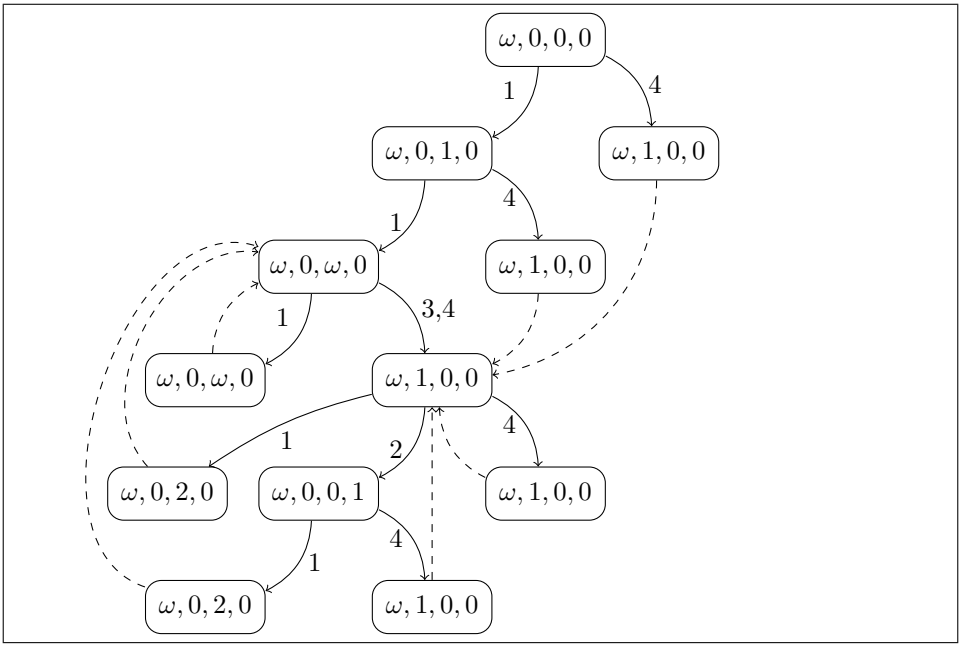

Рис. 6: Протокол MESI: граф конфигураций (однорезультатная суперкомпиляция)

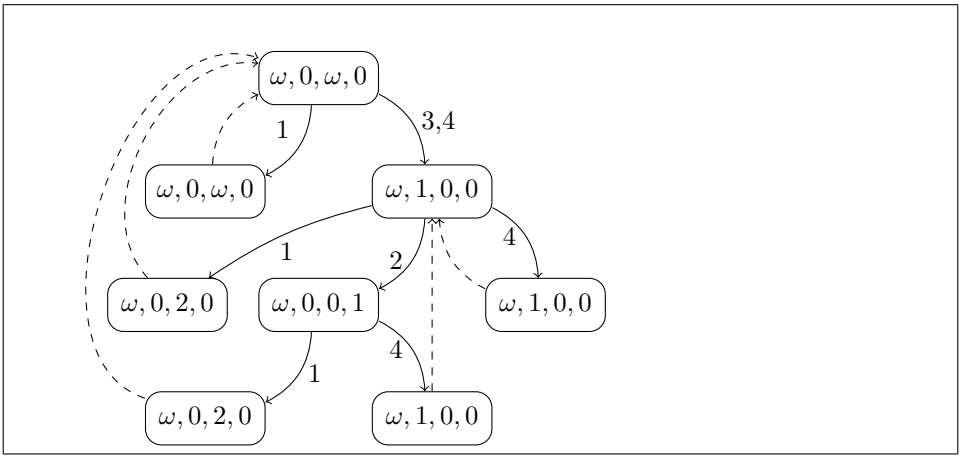

Рис. 7: Протокол MESI: минимальный граф конфигураций (многорезультатная суперкомпиляция)

нии проекций Футамуры [4]), а представляет собой смесь из значений первого порядка (чисел, списков) и значений высшего порядка (функций). Этот подход близок к технике реализации DSL, известной как “поверхностное вло-

жение".

В результате суперкомпиляции протокола MESI получается граф конфигураций, показанный на Рис. 6.

4 Применение многорезультатной суперкомпиляции для поиска коротких доказательств

При применении суперкомпиляции для анализа систем, граф конфигураций, порождаемый суперкомпилятором, является описанием множества достижимых состояний (аппроксимацией этого множества сверху). Этот граф затем может быть преобразован в доказательство того, что достижимы только состояния, удовлетворяющие некоторым требованиям ("надежные" состояния).

Чем меньше размер такого графа, тем он понятнее для человека (и тем короче доказательство, которое получается из этого графа). Однако, в случае традиционной однорезультатной суперкомпиляции выдается только один из возможных графов, и необязательно минимальный из возможных.

Многорезультатная суперкомпиляция порождает не один граф, а множество графов конфигураций. Затем можно автоматически профильтровать множество этих графов, отобрав среди них "самые лучшие". В простейшем случае, "самые лучшие" – это имеющие наименьший размер, хотя можно фильтровать и по другим критериям (например, пытаться выделять "хорошо структурированные" графы, чтобы потом преобразовывать их в "хорошо структурированные" доказательства).

Например, в то время как граф, построенный для протокола MESI (см. Рис. 6) с помощью однорезультатной позитивной суперкомпиляции [35, 36] содержит 12 вершин, в множестве графов, порожденных многорезультатным суперкомпилятором, обнаруживается граф, показанный на Рис. 7, содержащий только 8 вершин.

Как можно заметить, однорезультатный суперкомпилятор строит не минимальный граф из-за того, что он пытается избегать обобщений, что естественно и полезно в случае оптимизирующей суперкомпиляции, но не всегда разумно в случае суперкомпиляции, выполняемой с целью анализа систем. В результате этого, однорезультатный суперкомпилятор начинает с конфигурации $(\omega, 0, 0, 0)$ и только через некоторое время приходит к конфигурации $(\omega, 0, \omega, 0)$, являющейся обобщением конфигурации $(\omega, 0, 0, 0)$.

В то же время, многорезультатный суперкомпилятор (как бы в силу "гениального озарения") начинает с того, что выполняет обобщение стартовой конфигурации. С точки зрения оптимизирующей суперкомпиляции такое действие кажется странным и бессмысленным. Однако, оно приводит к тому, что порождается граф, показанный на Рис. 7), который является подграфом графа на Рис. 6.

Отметим, что в случае однорезультатной суперкомпиляции обобщение происходит в тех случаях, когда срабатывает свисток, поэтому алгоритм

```
case object MOESI extends Protocol {
  val start: Conf = List(Omega, 0, 0, 0, 0)
  val rules: List[TransitionRule] =
    List({ // rm
      case List(i, m, s, e, o) if i>=1 =>
        List(i-1, 0, s+e+1, 0, o+m)
    }, { // wh2
      case List(i, m, s, e, o) if e>=1 =>
        List(i, m+1, s, e-1, o)
    }, { // wh3
      case List(i, m, s, e, o) if s+o>=1 =>
        List(i+e+s+m+o-1, 0, 0, 1, 0)
    }, { // wm
      case List(i, m, s, e, o) if i>=1 =>
        List(i+e+s+m+o-1, 0, 0, 1, 0)
    })

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

Рис. 8: Протокол MOESI: модель протокола в виде DSL-программы

обобщения и свисток тесно связаны между собой и должны разрабатываться одновременно. А в случае многорезультатной суперкомпиляции свисток и обобщение работают совершенно независимо друг от друга. В частности, обобщение может происходить в самые неожиданные моменты времени, когда в этом (с точки зрения свистка) нет никакой необходимости. В результате этого, многорезультатный суперкомпилятор может обнаруживать такие графы конфигураций, которые не обнаруживаются однорезультатным суперкомпилятором.

Рассмотрим это на примере верификации протокола MOESI (Рис. 8). Граф, построенный однорезультатным суперкомпилятором (Рис. 9) содержит 20 вершин, в то время как многорезультатный суперкомпилятор обнаруживает граф (Рис.10), содержащий только 8 вершин.

В данном случае, это происходит за счет того, что многорезультатный суперкомпилятор делает “гениальную догадку” о том, что начальную конфигурацию $(\omega, 0, 0, 0, 0)$ можно сразу обобщить до конфигурации $(\omega, 0, \omega, 0, \omega)$. В результате этого получается граф из 8-ми вершин, который *не содержится* в качестве подграфа в графе, порожденном однорезультатной суперкомпиля-

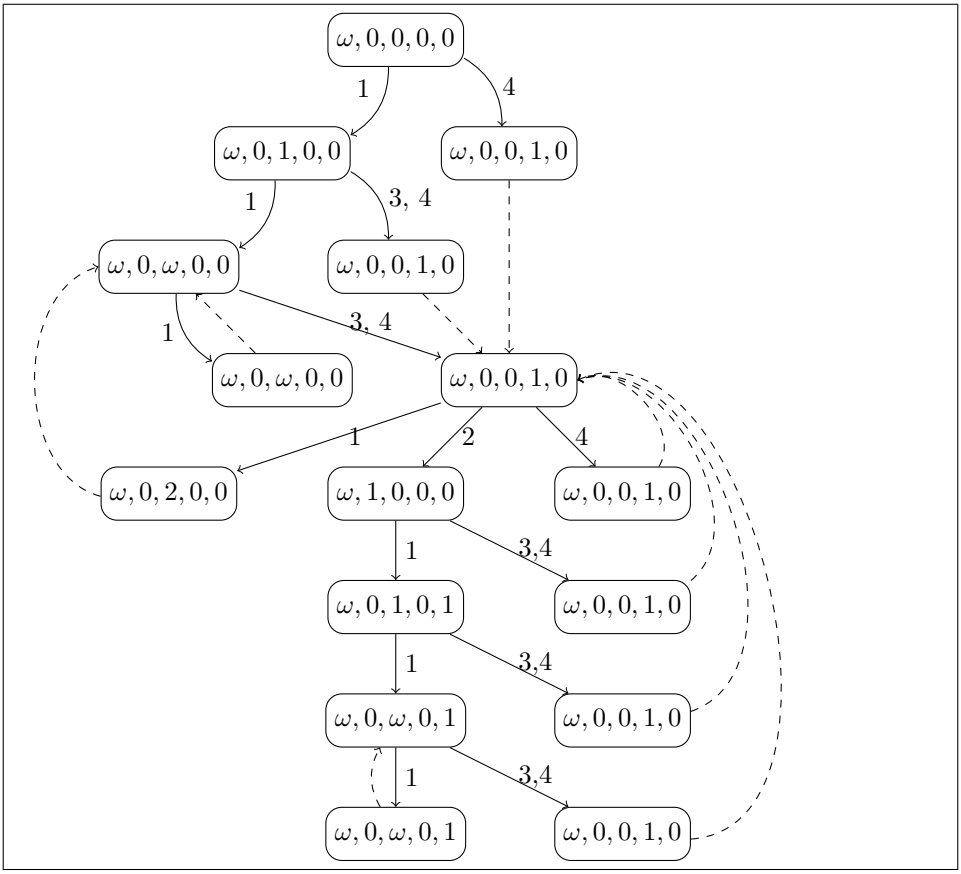

Рис. 9: Протокол MOESI: граф конфигураций (однорезультатная суперкомпиляция)

цией. При этом конфигурация $(\omega, 0, \omega, 0, \omega)$ вообще *ни разу не появляется* в графе, порожденном однорезультатной суперкомпиляцией, и вообще, графы на Рис. 9 и Рис.10 имеют совершенно разную структуру.

Отметим, что существует предметно-ориентированный алгоритм суперкомпиляции для счетчиковых систем [13], который, даже в случае однорезультатной суперкомпиляции, способен распознавать некоторые случаи, когда можно уменьшить размер графа за счет более энергичного обобщения конфигураций. Например, в случае протокола MESI получается тот же самый минимальный граф конфигураций (Рис. 7), что был найден с помощью многорезультатной суперкомпиляции.

Основная идея этого алгоритма заключается в следующем. Допустим, что в процессе суперкомпиляции возникла конфигурация c, являющаяся частным случаем конфигурации c', уже имеющейся в графе. Тогда c должна быть

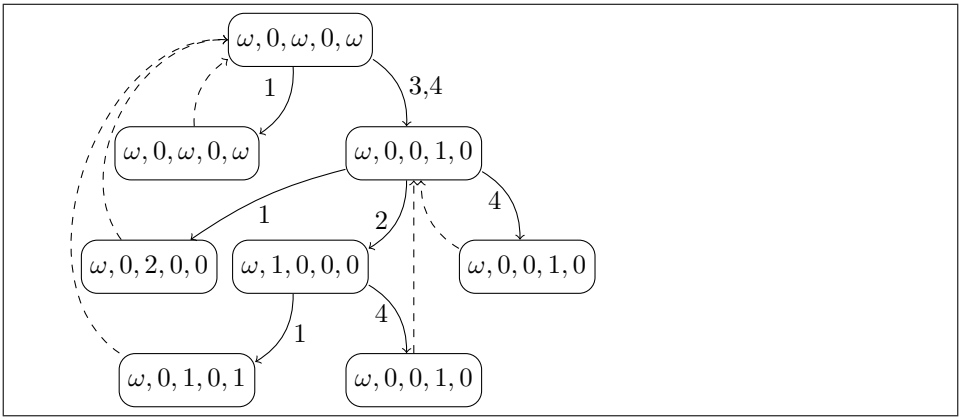

Рис. 10: Протокол MOESI: минимальный граф конфигураций (многорезультатная суперкомпиляция)

	SC	MRSC
Synapse	11	6
MSI	8	6
MOSI	26	14
MESI	14	9
MOESI	20	9
Illinois	15	13
Berkley	17	6
Firefly	12	10
Futurebus	45	24
Xerox	22	13
Java	35	25
ReaderWriter	48	9
DataRace	9	5

Рис. 11: Сравнение размеров доказательств (графов конфигураций)

обобщена до c'.

Однако, это не всегда приводит к построению минимальных графов конфигураций. Например, в случае протокола MOESI, многорезультатная суперкомпиляция находит такие конфигурации, которые вообще не возникают в случае использования классического алгоритма позитивной суперкомпиляции [35, 36].

На Рис. 11 показаны результаты верификации 13-ти протоколов. Столбец SC показывает количество узлов в графах, порожденных с помощью классической однорезультатной позитивной суперкомпиляции, а столбец MRSC –

количество узлов в графах, порожденных с помощью “наивной” многорезультатной суперкомпиляции, построенной на основе позитивной суперкомпиляции [35, 36] с помощью процедуры, описанной в [15]. Видно, что (практически всегда) многорезультатная суперкомпиляция позволяет найти графы меньшего размера, чем в случае использования однорезультатной суперкомпиляции.

5 Предметно-ориентированная резидуализация графов конфигураций

В случае классической универсальной суперкомпиляции, суперкомпиляция происходит в два этапа. На первом этапе строится конечный граф конфигураций, а на следующем этапе этот граф преобразуется в остаточную программу (“резидуализируется”). Например, результатом работы суперкомпилятора SCP4 является программа на языке Рефал (Рис. 3).

В случае анализа счетчиковых систем, остаточная программа не предназначена не для исполнения: интерес представляют только некоторые её свойства. Например, в случае программы на Рис. 3, существенным является только ответ на вопрос: может эта программа выдать `False` или нет? Это можно установить либо путем анализа остаточной программы человеком, либо применив к остаточной программе один из алгоритмов анализа потока данных [6].

Однако, в случае применения суперкомпиляции для анализа поведения систем, во многих случаях, можно вообще обойтись без порождения и анализа остаточной программы, а вместо этого анализировать получающиеся графы конфигураций. Затем, после выявления графа с нужными свойствами, можно преобразовывать этот граф не в программу на языке программирования, а в скрипт для системы доказательства теорем [18].

Реализованный нами предметно-ориентированный суперкомпилятор автоматически преобразует графы конфигураций в скрипты для системы доказательства теорем Isabelle [32], который содержит формулировку задачи достижимости для заданного протокола и набор тактик, применяя которые Isabelle способна доказать недостижимость ненадежных состояний.

Например, в случае протокола MESI порождается скрипт, показанный на Рис. 12. Этот скрипт состоит из следующих частей.

- Индуктивные определения (в виде предикатов) для множества достижимых состояний `mesi` и множества ненадежных состояний `unsafe`. Эти определения, с точностью до обозначений, повторяют определения из исходной постановки задачи.

- Индуктивное определение (в виде предиката) множества `mesi'`. Это – самая важная и нетривиальная часть скрипта. Множество `mesi'` вклю-

```
theory mesi
imports Main
begin

inductive mesi :: "(nat * nat * nat * nat) => bool" where
  "mesi (i, 0, 0, 0)" |
  "mesi (Suc i, e, s, m) ==> mesi (i, 0, Suc (s + e + m), 0)" |
  "mesi (i, Suc e, s, m) ==> mesi (i, e, s, Suc m)" |
  "mesi (i, e, Suc s, m) ==> mesi (i + e + s + m, Suc 0, 0, 0)" |
  "mesi (Suc i, e, s, m) ==> mesi (i + e + s + m, Suc 0, 0, 0)"

inductive unsafe :: "(nat * nat * nat * nat) => bool" where
  "unsafe (i, e, s, Suc (Suc m))" |
  "unsafe (i, e, Suc s, Suc m)"


inductive mesi' :: "(nat * nat * nat * nat) => bool" where
  "mesi'(_, Suc 0, 0, 0)" |
  "mesi'(_, 0, 0, Suc 0)" |
  "mesi'(_, 0, _, 0)"

lemma inclusion: "mesi c ==> mesi' c"
  apply(erule mesi.induct)
  apply(erule mesi'.cases | simp add: mesi'.intros)+
done

lemma safety: "mesi' c ==> ~unsafe c"
  apply(erule mesi'.cases)
  apply(erule unsafe.cases | auto)+
done

theorem valid: "mesi c ==> ~unsafe c"
  apply(insert inclusion safety, simp)
done

end
```

Рис. 12: Скрипт для верификации протокола MESI для системы доказательства теорем Isabelle

чает в себя множество `mesi`. Его определение, с точностью до обозначений, является перечислением конфигурации из графа конфигураций на Рис. 7. При этом, для уменьшения размера скрипта, выполняется простая оптимизация: если в графе имеются две конфигурации c' и c, и c' является частным случаем c, то c' не включается в определение

предиката `mesi'`.

- Лемма `inclusion`, утверждающая, что все достижимые состояния c (для которых верно `mesi` c) входят и в множество `mesi'` (т.е. для них верно `mesi'` c).

- Лемма `safety`, утверждающая, что все состояния c из `mesi'` являются надежными (т.е. из `mesi'` c следует $\neg$`unsafe` c).

- Основной теоремы: все достижимые состояния c являются надежными (из `mesi` c следует $\neg$`unsafe` c). Это тривиально следует из лемм `inclusion` и `safety`).

Главное различие между определениями предикатов `mesi` и `mesi'` состоит в том, что `mesi` определяется рекурсивно, в то время как определение `mesi'` – это простой набор случаев. Поэтому, доказательство того, что все состояния из `mesi'` являются надежными – тривиально, и производится простым разбором случаев.

Таким образом, роль суперкомпиляции при анализе счетчиковых систем, по существу, и состоит в нахождении такого обобщения для множества достижимых состояний, для которого доказательство недостижимости становится тривиальным. Следовательно, суперкомпиляция может служить полезным дополнением к другим известным методам доказательства теорем и верификации.

6 Повышение эффективности суперкомпиляции за счет учета специфики предметной области

6.1 Использование свойств предметно-ориентированных операций

В случае суперкомпиляции счетчиковых систем, достаточно работать с конфигурациями вида $(a_1, \ldots, a_n)$, каждая компонента которых a_i является либо натуральным числом N, либо символом ω. При переходах от одной конфигурации к другой (прогонке) 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 $\omega + N = \omega - N = \omega + \omega = \omega$. Такие операции легко реализуются средствами языка Scala (Рис. 13).

В случае же использования универсального суперкомпилятора, работающего с программами на универсальном языке, суперкомпилятор не обладает сведениями о конкретной предметной области и о специфике операций над такими данными. Например, в случае использования суперкомпилятора SCP4,

```
package object counters {
  ...
  implicit def intToExpr(i: Int): Expr = Num(i)
}

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. 13: Счетчиковые системы: реализация операций над компонентами конфигураций на языке Scala

натуральные числа приходится изображать в виде линейных последовательностей символов, а сложение чисел – в виде конкатенации последовательностей (Рис. 2 и 3). В результате, тот факт, что программа работает именно с натуральными числами, становится неочевидным (как для человека, так и для суперкомпилятора).

6.2 Прямая реализация недетерминизма в суперкомпиляторе

Входные языки универсальных оптимизирующих компиляторов, как правило, предназначены для записи детерминированных программ. Это создает определенные неудобства в случае использования суперкомпиляции для анализа поведения недетерминированных систем. Действительно, если мы формализуем поведение таких систем в виде программ, то такие программы удобно писать на недетерминированном языке программирования. Если же модель недетерминированной системы приходится записывать на детерминированном языке, приходится использовать различные трюки и искусственные приемы. Это усложняет программу и затемняет её смысл.

Например, рассмотрим модель протокола MESI в виде Рефал-программы (Рис. 2). Входной точкой программы является функция *Go*, которая принимает два параметра: `e.A` и `e.I` [25, 26]

```
$ENTRY Go {e.A (e.I) =
    <Loop (e.A) (Invalid e.I)(Modified )(Shared )(Exclusive ) >;}
```

Параметр `e.I` служит для формирования начального состояния, а параметр `e.A` вводится для моделирования недетерминизма. Поскольку система переходов состоит из нескольких правил, и на каждом шаге может быть применимо более чем одно правило, параметр `e.A` указывает, какое из правил следует выбрать.

Естественно, этот дополнительный параметр появляется на только в исходной программе, но и в конфигурациях, возникающих в процессе суперкомпиляции. Сохраняется он и в остаточной программе (Рис. 3), что усложняет её и затемняет её смысл.

Если же модель недетерминированной системы записывается на недетерминированном предметно-ориентированном языке (Рис. 4), отпадает и необходимость использования искусственных приемов для изображения недетерминизма. При этом, недетерминизм входной программы не создает дополнительных проблем при суперкомпиляции, ибо суперкомпилятор, в отличие от обычного интерпретатора, всё равно должен изучать не один конкретный способ исполнения программы, а все возможные способы исполнения (для заданного множества исходных состояний).

6.3 Фильтрация графов конфигураций вместо фильтрации остаточных программ

Одним из применений многорезультатной суперкомпиляции является поиск остаточных программ, удовлетворяющих некоторым требованиям. При этом, многорезультатный суперкомпилятор может порождать сотни или даже тысячи остаточных программ. Ясно, что в такой ситуации приходится приме-

нять автоматическую фильтрацию остаточных программ, отбирая программы, удовлетворяющие заданным критериям.

Например, в случае применения универсальной суперкомпиляции для анализа счетчиковых систем, требуется отобрать такие остаточные программы, которые заведомо не выдают `False`, и такую фильтрацию можно автоматизировать, применив, например, к остаточным программам один из известных алгоритмов анализа потока данных [6].

Однако, в случае проблемно-ориентированной суперкомпиляции, процесс отбора остаточных программ можно упростить. Например, в случае счетчиковых систем, "остаточными программами" могут быть скрипты для систем доказательства теорем (см. раздел 5), а фильтрацию результатов суперкомпиляции можно производить ещё до генерации скриптов, анализируя построенные графы конфигураций.

Таким образом, фильтрация остаточных программ заменяется на фильтрацию графов конфигураций. Однако, приняв во внимание специфику предметной области, можно оптимизировать и процесс фильтрации графов, отбрасывая некоторые графы ещё до того, как они полностью построены. Это значительно сокращает перебор в процессе многорезультатной суперкомпиляции, поскольку при этом отсекается целое множество графов (которые могли бы получиться при достройке текущего графа).

В случае счетчиковых систем можно воспользоваться следующими особенностями предметной области. 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 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 появилась ненадежная конфигурация, достраивать этот граф не имеет смысла, ибо во всех полученных из него графах будет содержаться хотя бы одна ненадежная конфигурация.

При этом, эффективность многорезультатной суперкомпиляции зависит от того, в каких местах алгоритма суперкомпиляции производится обнаружение ненадежных конфигураций.

Вторая оптимизация, учитывающая специфику предметной области, основана на свойствах множества возможных обобщений для заданной конфигурации c.

А именно, обобщения c получаются путем замены некоторых компонент-чисел в c на ω. Например, конфигурацию $(0, 0)$. можно обобщить тремя различными способами, получив конфигурации $(\omega, 0)$, $(0, \omega)$, (ω, ω). Легко видеть, что конфигурация (ω, ω) является обобщением по отношению к конфигурациям $(\omega, 0)$ и $(0, \omega)$.

Наивный алгоритм многорезультатной суперкомпиляции, пытаясь перестроить конфигурацию c путем её замены на некоторое обобщение c', рассматривает сразу все множество возможных обобщений. Некоторые обобще-

ния не являются максимальными, и соответствующие конфигурации, через некоторое время, обобщаются снова. Например, $(\omega, 0)$, и $(0, \omega)$ обобщаются до (ω, ω). В результате получается, что мы приходим к графу, содержащему (ω, ω) три раза: один раз непосредственно, и два раза через графы, содержащие $(\omega, 0)$, и $(0, \omega)$.

Можно значительно сократить количество рассматриваемых графов, если рассматривать для заданной конфигурации c не всё множество её возможных обобщений, а только те обобщения c', которые получаются заменой только одной компоненты-числа в c на ω.

Нами были рассмотрены 5 вариантов алгоритма суперкомпиляции SC1, SC2, SC3, SC4 и SC5. Каждый из которых отличается от предыдущего тем, что в нём реализована одна дополнительная оптимизация.

- SC1. Фильтрация и генерация графов полностью отделены друг от друга. Графы достраиваются до конца, а затем проверяются на наличие в них ненадежных конфигураций. Таким образом, не используется особенности предметной области: возможность разложить любое обобщение в последовательность элементарных обобщений и монотонность предиката `unsafe`. Достоинство такой конструкции – её модульность, а недостаток – неэффективность суперкомпилятора.

- SC2. Добавлена оптимизация: при перестройке конфигурации рассматриваются только те обобщения, которые получаются при изменении только одной компоненты конфигурации.

- SC3. Добавлена проверка на опасные конфигурации при перестройке конфигурации. Те обобщения конфигурации, которые являются ненадежными, не рассматриваются.

- SC4. Добавлена проверка на опасные конфигурации при прогонке конфигурации. Если выполнение шага прогонки привело бы к порождению хотя бы одной ненадежной конфигурации, прогонка для данной конфигурации не выполняется.

- SC5. Добавлено отбрасывание графов имеющих слишком большой размер. А именно, если среди уже достроенных графов имеется граф, размер которого меньше, чем размер текущего графа, то текущий граф отбрасывается. (Заметим, что в силу предыдущих оптимизаций, достроенные графы содержат только надежные конфигурации).

Легко заметить, что оптимизация, реализованная в SC5 характерна для алгоритмов из области искусственного интеллекта, где она известна как “отсечение” (pruning) [34].

Таблица на Рис. 14 показывает, какие ресурсы были потреблены в процессе верификации 13-ти протоколов (кеш-когерентности и коммуникационных) для 5-ти версий алгоритма суперкомпиляции. Строчки completed показывают

		SC1	SC2	SC3	SC4	SC5
Synapse	completed	48	37	3	3	1
	pruned	0	0	0	0	2
	commands	321	252	25	25	15
MSI	completed	22	18	2	2	1
	pruned	0	0	0	0	1
	commands	122	102	15	15	12
MOSI	completed	1233	699	6	6	1
	pruned	0	0	0	0	5
	commands	19925	11476	109	109	35
MESI	completed	1627	899	6	3	1
	pruned	0	0	27	20	21
	commands	16329	9265	211	70	56
MOESI	completed	179380	60724	81	30	2
	pruned	0	0	0	24	36
	commands	2001708	711784	922	384	126
Illinois	completed	2346	1237	2	2	1
	pruned	0	0	21	17	18
	commands	48364	26636	224	74	61
Berkley	completed	3405	1463	30	30	2
	pruned	0	0	0	0	14
	commands	26618	12023	282	282	56
Firefly	completed	2503	1450	2	2	1
	pruned	0	0	2	2	3
	commands	39924	24572	47	25	21
Futurebus	completed	-	-	-	-	4
	pruned	-	-	-	-	148328
	commands	-	-	-	-	516457
Xerox	completed	317569	111122	29	29	2
	pruned	0	0	0	0	1
	commands	5718691	2031754	482	482	72
Java	completed	-	-	-	-	10
	pruned	-	-	-	-	329886
	commands	-	-	-	-	1043563
ReaderWriter	completed	892371	402136	898	898	6
	pruned	0	0	19033	19033	1170
	commands	24963661	11872211	123371	45411	3213
DataRace	completed	51	39	8	8	3
	pruned	0	0	0	0	4
	commands	360	279	57	57	31

Рис. 14: Сравнение ресурсов, потребляемых различными версиями многорезультатных суперкомпиляторов

количество построенных завершенных графов (среди которых могут встречаться и повторы), pruned – количество отброшенных незавершенных графов (которые суперкомпилятор не стал достраивать), commands – количество элементарных операций по построению графов, которые были выполнены в процессе работы суперкомпилятора.

Для протоколов Futurebus и Java приводятся данные только для варианта SC5, поскольку для других вариантов суперкомпиляции время работы оказалось слишком большим, и результаты не были получены.

Приведенные данные показывают, что использование специфики предметной области приводит к значительному уменьшению ресурсов, потребляемых многорезультатной суперкомпиляцией.

7 Заключение

Многорезультатная суперкомпиляция является не теоретическим курьезом, а инструментом, который при разумном использовании может приносить реальную пользу.

- Многорезультатная суперкомпиляция позволяет повысить качество результатов анализа систем переходов благодаря рассмотрению разных вариантов анализа и отбору наилучших результатов.
- Многорезультатная суперкомпиляция позволяет упростить структуру суперкомпилятора за счет использования независимых друг от друга свистка и алгоритма обобщения, а также за счет использования конфигураций упрощенной структуры.

Польза от языково- и предметно-ориентированной суперкомпиляции проявляется в следующем.

- Задания для суперкомпилятора можно писать на предметно-ориентированном языке, который лучше соответствует понятиям рассматриваемой предметной области, чем универсальный язык. (Например, позволяет естественным путем выражать недетерминизм. Или предоставляет типы данных и операции над данными, специфические для рассматриваемой предметной области.)
- Можно упростить алгоритм суперкомпиляции благодаря тому, что не приходится реализовывать то, что не является существенным и/или полезным для рассматриваемой предметной области.
- Можно повысить эффективность многорезультатной суперкомпиляции за счет раннего отбрасывания неудачных вариантов суперкомпиляции.
- Инструментарий MRSC делает изготовление специализированных суперкомпиляторов делом малотрудоёмким (стало быть – практичным).

Таким образом, сочетание предметно-ориентированной и многорезультатной суперкомпиляции дает синергетический эффект: многорезультатность позволяет выбирать наилучшие решения проблемы, а учет особенностей предметной области позволяет уменьшить количество ресурсов, потребляемых многорезультатной суперкомпиляцией.

Благодарности

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

Список литературы

[1] D. Bjørner, M. Broy, and I. V. Pottosin, editors. *Perspectives of Systems Informatics, Second International Andrei Ershov Memorial Conference, Akademgorodok, Novosibirsk, Russia, June 25–28, 1996*, volume 1181 of *Lecture Notes in Computer Science*. Springer, 1996.

[2] 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.

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

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

[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] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci.*, 375(1-3):120–136, 2007.

[7] J. Jørgensen and M. Leuschel. Efficiently generating efficient generating extensions in Prolog. In O. Danvy, R. Glück, and P. Thiemann, editors, *Dagstuhl Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 238–262. Springer, 1996.

[8] 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.

[9] A. V. Klimov. JVer project: Verification of Java programs by the Java Supercompiler. http://pat.keldysh.ru/jver/, 2008.

[10] A. V. Klimov. A Java Supercompiler and its application to verification of cache-coherence protocols. In Pnueli et al. [33], pages 185–192.

[11] 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.

[12] 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.

[13] A. V. Klimov. Solving coverability problem for monotonic counter systems by supercompilation. In Clarke et al. [2], pages 193–209.

[14] I. G. Klyuchnikov and S. A. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In Pnueli et al. [33], pages 193–205.

[15] 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?id=2011-77.

[16] I. G. Klyuchnikov and S. A. Romanenko. Multi-result supercompilation as branching growth of the penultimate level in metasystem transitions. In Clarke et al. [2], pages 210–226.

[17] D. Krustev. A simple supercompiler formally verified in Coq. In A. P. Nemytykh, editor, *Second International Valentin Turchin Memorial Workshop on Metacomputation in Russia, Pereslavl-Zalessky, Russia, July 1–5, 2010*, pages 102–127. Ailamazyan University of Pereslavl, Pereslavl-Zalessky, 2010.

[18] H. Lehmann and M. Leuschel. Inductive theorem proving by program specialisation: Generating proofs for Isabelle using Ecce. In M. Bruynooghe, editor, *LOPSTR*, volume 3018 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.

[19] M. Leuschel and J. Jørgensen. Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. *Electr. Notes Theor. Comput. Sci.*, 30(2):157–162, 1999.

[20] M. Leuschel and H. Lehmann. Coverability of reset Petri nets and other well-structured transition systems by partial deduction. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 101–115, London, UK, 2000. Springer.

[21] M. Leuschel and H. Lehmann. Solving coverability problems of Petri nets by partial deduction. In *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[22] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Trans. Program. Lang. Syst.*, 20:208–258, January 1998.

[23] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In *Selected papers from the 9th International Workshop on Logic Programming Synthesis and Transformation*, pages 62–81, London, UK, 2000. Springer.

[24] M. Leuschel and D. D. Schreye. Logic program specialisation: How to be more specific. In H. Kuchen and S. D. Swierstra, editors, *PLILP*, volume 1140 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 1996.

[25] A. P. Lisitsa and A. P. Nemytykh. SCP4: Verification of protocols. http://refal.botik.ru/protocols/.

[26] A. P. Lisitsa and A. P. Nemytykh. Verification of MESI cache coherence protocol. http://www.csc.liv.ac.uk/~alexei/VeriSuper/node5.html.

[27] A. P. Lisitsa and A. P. Nemytykh. Towards verification via supercompilation. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), 25-28 July 2005, Edinburgh, Scotland, UK*, pages 9–10. IEEE Computer Society, 2005.

[28] A. P. Lisitsa and A. P. Nemytykh. Verification as a parameterized testing (experiments with the SCP4 supercompiler). *Programming and Computer Software*, 33(1):14–23, 2007.

[29] A. P. Lisitsa and A. P. Nemytykh. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.

[30] 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.

[31] A. P. Nemytykh and V. A. Pinchuk. Program transformation with metasystem transitions: Experiments with a supercompiler. In Bjørner et al. [1], pages 249–260.

[32] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, Berlin, Heidelberg, 2002.

[33] A. Pnueli, I. Virbitskaite, and A. Voronkov, editors. *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Akademgorodok, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, volume 5947 of *Lecture Notes in Computer Science*. Springer, 2010.

[34] D. Poole and A. K. Mackworth. *Artificial Intelligence – Foundations of Computational Agents*. Cambridge University Press, 2010.

[35] 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.

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

[37] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, POPL '06, pages 372–382. ACM, 2006.

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

[39] 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.

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

[41] V. F. Turchin. Supercompilation: Techniques and results. In Bjørner et al. [1], pages 227–248.

[42] 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.