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

Илья Ключников

# Суперкомпилятор HOSC 1.5: гомеоморфное вложение и обобщение для выражений высшего порядка

Москва
2010

**Ilya Klyuchnikov. Supercompiler HOSC 1.5: homeomorphic embedding and generalization in a higher-order setting**

The paper describes the algorithm of the supercompiler HOSC 1.5, an experimental specializer dealing with programs written in a higher-order functional language. The design decisions behind the algorithm are illustrated through a series of examples. Of particular interest are the decisions related to generalization and homeomorphic embedding of expressions with bound variables.

Supported by Russian Foundation for Basic Research projects No. 08-07-00280-a and No. 09-01-00834-a.

**Илья Ключников. Суперкомпилятор HOSC 1.5: гомеоморфное вложение и обобщение для выражений высшего порядка**

В работе приводится алгоритм экспериментального суперкомпилятора HOSC 1.5, работающего с функциями высших порядков. Детали алгоритма обосновываются на ряде примеров. Особое внимание уделяется обобщению и гомеоморфному вложению выражений со связанными переменными.

Работа выполнена при поддержке проектов РФФИ № 08-07-00280-а и № 09-01-00834-а.

# Содержание

# 1 Введение

Целью экспериментального суперкомпилятора HOSC[1] является анализ программ. Эта цель достигается за счет агрессивного распространения позитивной информации. В статье [11] показана применимость суперкомпилятора HOSC для распознаванию эквивалентных программ на языке Haskell, в частности, отмечена склонность суперкомпилятора HOSC к нормализации выражений – приведению эквивалентных выражений к одной и той же синтаксической форме. На базе алгоритма распознавания эквивалентных выражений строится алгоритм распознавания улучшающих лемм, лежащий в основе нового метода многоуровневой суперкомпиляции [12].

В работе [8] был описан алгоритм суперкомпилятора HOSC 1.0.

В работе [10] суперкомпилятор HOSC рассматривается в самом общем виде – в виде *отношения преобразования* и доказывается корректность отношения преобразования HOSC.

Во время работы над доказательством завершаемости суперкомпилятора HOSC, автором было обнаружено, что суперкомпилятор HOSC 1.0 может не завершаться [9]. Источник возможной незавершаемости был устранен в суперкомпиляторе HOSC 1.1 и была доказана завершаемость суперкомпилятора HOSC 1.1 [9].

Стоит отметить, что завершаемость суперкомпилятора гарантируется в [9] в некоторой степени “ad hoc” образом – разбиение узлов на тривиальные и нетривиальные происходит на основании размера выражения, находящегося в узле дерева, после шага редукции. Вдобавок, из-за нового разбиения узлов на тривиальные и нетривиальные, склонность суперкомпилятора HOSC 1.1 к нормализации выражений заметно уменьшилась по сравнению с суперкомпилятором HOSC 1.0.

В данной работе описывается суперкомпилятор HOSC 1.5, обладающий большей склонностью к нормализации, чем HOSC 1.0, и завершающийся на любой входной программе. Можно сказать, что проблема завершаемости решается в суперкомпиляторе HOSC 1.5 более универсальным и элегантным способом, нежели в суперкомпиляторе HOSC 1.1.

В работах [8, 9, 10] изложено лишь частичное обоснование целесообразности решений, лежащих в основе алгоритма суперкомпилятора HOSC. Отчасти неполнота обоснования обусловлена отсутствием материала для сравнения: другие существующие суперкомпиляторы для языков с функциями высших порядков преследуют другую цель – оптимизацию программ – и поэтому результаты работы суперкомпилятора HOSC трудно сравнивать с результатами работы других суперкомпиляторов.

Особенностью суперкомпиляторов HOSC 1.0 и HOSC 1.1 являются тесно взаимосвязанные определения уточненного гомеоморфного вложения и алгоритма, находящего тесное обобщение двух выражений. А именно, алгоритм обобщения определяется только для выражений, вложенных через сцепление и, таким образом, проблемы, возникающие при обобщении из-за связанных переменных, фактически не решаются, а обходятся стороной: алгоритм обобщения, описанный в [8, 9], *не затрагивает связанные переменные* по построению.

[1] Тексты суперкомпилятора HOSC доступны по адресу `http://hosc.googlecode.com`

**Рис. 1** SLL: грамматика

| | | | |
|---|---|---|---|
| $P$ | ::= | $d_1 \ldots d_n$ | program |
| $d$ | ::= | $f(v_1, \ldots, v_n) = e;$ | f-function |
| | \| | $g(p_1, v_1, \ldots, v_n) = e_1;$ | g-function |
| | | $\ldots$ | |
| | | $g(p_m, v_1, \ldots, v_n) = e_m;$ | |
| $e$ | ::= | $v$ | variable |
| | \| | $C(e_1, \ldots, e_n)$ | constructor |
| | \| | $f(e_1, \ldots, e_n)$ | call to f-function |
| | \| | $g(e_1, \ldots, e_n)$ | call to g-function |
| $p$ | ::= | $C(v_1, \ldots, v_n)$ | pattern |

В данной работе произведена ревизия алгоритма обобщения: теперь этот алгоритм может применяться к *любым* парам выражений, и наиболее тесное обобщение строится даже в случаях, когда выражения содержат связанные переменные. При этом, если выражения не содержат связанные переменные, этот алгоритм вырождается в классический, хорошо известный алгоритм, применяющийся в суперкомпиляторах для языков первого порядка. Это облегчает сравнение нового алгоритма с классическим.

В данной работе рассматривается несколько алгоритмов суперкомпиляции для ядра языка Haskell. Целесообразность решений, лежащих в основе алгоритма суперкомпилятора HOSC 1.5, обосновывается с помощью примеров, показывающих его преимущества перед другими алгоритмами.

## 2 Язык SLL: вложение и обобщение

Сначала мы рассмотрим как устроено обобщение выражений и отношение гомеоморфного вложения в случае *языка первого порядка без связанных переменных*.

На Рис. 1 показан синтаксис языка SLL[2], рассматриваемого в работах [17, 18]. Выражение языка SLL – либо переменная, либо конструктор, либо вызов функции. Арность конструкторов и функций фиксирована. Количество конструкторов и функций – конечно.

В данном разделе применительно к выражениям языка SLL мы используем понятия *подстановки*, *частного случая выражения*, *обобщения* и *наиболее тесного обобщения* так, как они определяются в [18].

В [5, 20] рассматривается слегка расширенный язык SLL – с конструкцией `if` для проверки двух выражений на равенство (`if`-выражение рассматривается как специальная конструкция от четырех аргументов).

[2]Сам Сёренсен никак не называет рассматриваемый язык. Для удобства ссылок будем называть этот язык SLL – Simple Lazy Language.

**Рис. 2** SLL: итеративное обобщение выражений $e'$ и $e''$

**Initial trivial generalization**

$$(v, \{v := e_1\}, \{v := e_2\})$$

**Common functor rule**

$$\left(\begin{array}{c} e \\ \{v := h(e'_1,\ \ldots,\ e'_n)\} \cup \theta' \\ \{v := h(e''_1,\ \ldots,\ e''_n)\} \cup \theta'' \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v := h(v_1,\ \ldots,\ v_n)\} \\ \{v_1 := e'_1,\ \ldots,\ v_n := e'_n\} \cup \theta' \\ \{v_1 := e''_1,\ \ldots,\ v_n := e''_n\} \cup \theta'' \end{array}\right)$$

**Common subexpression rule**

$$\left(\begin{array}{c} e \\ \{v_1 := e',\ v_2 := e'\} \cup \theta' \\ \{v_1 := e'',\ v_2 := e''\} \cup \theta'' \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v_1 := v_2\}) \\ \{v_2 := e'\} \cup \theta' \\ \{v_2 := e''\} \cup \theta'' \end{array}\right)$$

Отношение гомеоморфного вложения для выражений языка SLL хорошо разработаны в [17, 18, 20]. В работах [17, 18] приводится алгоритм нахождения обобщения для выражений языка SLL.

Будем рассматривать вложение и обобщение, описанные в [17, 18, 20] как некоторую точку отсчета для дальнейшего построения вложения и обобщения в случае связанных переменных и наличия функций высших порядков.

В работе [19] рассматривается SLL, дополненный `case`-выражениями (со связанными переменными), но отношение вложения и алгоритм обобщения приводятся только для случая выражений без связанных переменных, – читателю предлагается самостоятельно расширить приведенные соотношения на случай связанных переменных в `case`-выражениях, поэтому в данном разделе мы не затрагиваем вопросы вложения и обобщения выражений со связанными переменными.

В дальнейшем в данном разделе конструкция $h(e'_1,\ \ldots,\ e'_n)$ используется для обозначения конструктора или вызова функции.

**Определение 1** (Итеративный алгоритм обобщения SLL-выражений)**.** Тесное обобщение выражений $e'$ и $e''$, $e' \sqcap e''$ находится с помощью применений правил общего выражения и общего подвыражения (Рис. 2) к начальному тривиальному обобщению $(v, \{v := e'\}, \{v := e''\})$.

Алгоритм 1 является традиционной [19, 20, 18] записью алгоритма обобщения в итеративной форме (определяется “малый шаг” обобщения).

Однако, процесс обобщения удобнее (как мы увидим в дальнейшем) определить напрямую через рекурсивные функции (семантика “большого шага”):

**Определение 2** (Рекурсивный алгоритм обобщения SLL-выражений)**.** $e' \sqcap e'' = s(e' \widetilde{\sqcap} e'')$, где операции $\widetilde{\sqcap}$ и $s$ определены на Рис. 3.

Процесс нахождения обобщения происходит в 2 этапа: первый этап соответствует применению правила общего выражения, – нахождение обобщения (вычисление $e_1 \widetilde{\sqcap} e_2$) двух совпадающих по оболочке выражений сводится к задаче нахождения обобщений их частей. Второй этап (упрощение обобщения с помощью операции $s$) заключается в удалении из результата повторяющихся частей и соответствует правилу общего подвыражения. Правила применяются в порядке их объявления.

**Рис. 3** SLL: рекурсивный алгоритм обобщения

**Most specific generalization**

- $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$

**Common functor rule**

- $v \mathbin{\widetilde{\sqcap}} v = (v, \{\}, \{\})$
- $h(e'_1, \ldots, e'_n) \mathbin{\widetilde{\sqcap}} h(e''_1, \ldots, e''_n) = (h(e_1, \ldots, e_n), \bigcup \theta'_i, \bigcup \theta''_i)$ where
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $e_1 \mathbin{\widetilde{\sqcap}} e_2 = (v, \{v := e_1\}, \{v := e_2\})$

**Common subexpression rule**

- $s(e, \{\}, \{\}) = s(e, \{\}, \{\})$
- $s(e, \{v_1 := e'\} \cup \theta', \{v_1 := e''\} \cup \theta'') = s'(s(e, \theta', \theta''))$ where
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e\{v_1 := v_2\},\ \theta'_1,\ \theta''_1)$ if $\exists v_2 : \{v_2 := e'\} \in \theta'_1, \{v_2 := e''\} \in \theta''_1$
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e,\ \{v_1 := e'\} \cup \theta'_1,\ \{v_1 := e''\} \cup \theta''_1)$ otherwise

**Рис. 4** SLL: гомеоморфное вложение

**Embedding**

$e' \trianglelefteq e''$ if $e' \trianglelefteq_d e''$ or $e' \trianglelefteq_c e''$ or $e' \trianglelefteq_v e''$

**Variables**

$v' \trianglelefteq_v v''$

**Coupling**

$h(e'_1, \ldots, e'_n) \trianglelefteq_c h(e''_1, \ldots, e''_n)$ if $\forall i : e'_i \trianglelefteq e''_i$

**Diving**

$e \trianglelefteq_d h(e'_1, \ldots, e'_n)$ if $\exists i : e \trianglelefteq e'_i$

Отношения гомеморфного вложения, как оно определяется в [17, 19] для языка SLL приведено на Рис. 4. Мы явным образом разделяем вложение через погружение и вложение через сцепление.

Будем называть множеством выражений программы $P$ множенство выражений языка SLL, соответствующих грамматике на Рис. 1, в которых присутствуют конструкторы и функции только из программы $P$.

В силу фиксированной арности конструкторов и функций, для любой программы $P$ отношение $\trianglelefteq$ является вполне-квазиупорядочением на множестве выраже-

**Рис. 5** HLL: грамматика

| | |
|---|---|
| $tDef$ ::= data $tCon = \overline{dCon_i}$; | type definition |
| $tCon$ ::= $tn\ \overline{tv_i}$ | type constructor |
| $dCon$ ::= $c\ \overline{type_i}$ | data constructor |
| $type$ ::= $tv \mid tCon \mid type \to type \mid (type)$ | type expression |
| | |
| $prog$ ::= $\overline{tDef_i}\ e$ **where** $\overline{f_i = e_i}$; | program |
| | |
| $e$ ::= $v$ | variable |
| $\mid c\ \overline{e_i}$ | constructor |
| $\mid f_g$ | global variable |
| $\mid \lambda\ \overline{v_i} \to e$ | $\lambda$-abstraction |
| $\mid e_1\ e_2$ | application |
| $\mid$ **case** $e_0$ **of** $\{\overline{p_i \to e_i;}\}$ | case-expression |
| $\mid$ **let** $\overline{v_i = e_i;}$ **in** $e$ | let-expression |
| $\mid (e)$ | parenthesized expression |
| | |
| $p$ :: = $c\ \overline{v_i}$ | pattern |

ний программы $P$. Данный результат известен как лемма Хигмана.

Легко показывается, что отношение $\trianglelefteq_c$ также является вполне-квазиупорядочением на множестве выражений программы $P$.

Также легко показывается, что для любых выражений программы $P$, если $e_1 \trianglelefteq_c e_2$, то обобщение $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$ – нетривиально ($e_g$ - не переменная). Это означает, что если при суперкомпиляции использовать в качестве свистка отношение $\trianglelefteq_c$, то можно избежать расщепления конфигурации безотносительно к истории вычислений, ибо всегда находится обобщение конфигурации, учитывающее историю вычислений.

# 3 Язык HLL: вложение и обобщение

В данном разделе рассматривается язык HLL[3], – входной язык суперкомпилятора HOSC [8]. Язык HLL очень близок к ядру языка Haskell [22, 23]. Перечисление образцов в `case`-выражениях должно быть полным и ортогональным. Язык типизирован по Хиндли-Милнеру[4].

**Замечание 3** (Соглашение о порядке образцов в `case`-выражениях)**.** Будем считать, что в `case`-выражениях образцы перечислены в том же порядке, что и конструкторы в декларации соответствующего типа данных.

[3]HLL = Higher-order Lazy Language

[4]На самом деле все рассматриваемые далее суперкомпиляторы корректно работают и для более общих систем типизации. Мы рассматриваем типизацию по Хиндли-Милнеру исключительно для упрощения подачи материала.

**Рис. 6** HLL: множество связанных переменных выражения

| | |
|---|---|
| $bv[\![f_g]\!]$ | $= \{\}$ |
| $bv[\![v]\!]$ | $= \{\}$ |
| $bv[\![c\ \overline{e_i}]\!]$ | $= \bigcup bv[\![e_i]\!]$ |
| $bv[\![\lambda v \to e]\!]$ | $= bv[\![e]\!]\ \cup\ \{v\}$ |
| $bv[\![e_1\ e_2]\!]$ | $= bv[\![e_1]\!] \cup bv[\![e_2]\!]$ |
| $bv[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$ | $= bv[\![e_0]\!] \cup (\bigcup bv[\![e_i]\!]) \cup (\bigcup v_{ik})$ |
| $bv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$ | $= bv[\![e]\!] \cup (\bigcup bv[\![e_i]\!])\ \cup\ (\bigcup v_i)$ |

**Рис. 7** HLL: множество глобальных переменных выражения

| | |
|---|---|
| $gv[\![f_g]\!]$ | $= \{f_g\}$ |
| $gv[\![v]\!]$ | $= \{\}$ |
| $gv[\![c\ \overline{e_i}]\!]$ | $= \bigcup gv[\![e_i]\!]$ |
| $gv[\![\lambda v \to e]\!]$ | $= gv[\![e]\!]$ |
| $gv[\![e_1\ e_2]\!]$ | $= gv[\![e_1]\!] \cup gv[\![e_2]\!]$ |
| $gv[\![case\ e\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$ | $= gv[\![e]\!] \cup (\bigcup gv[\![e_i]\!])$ |
| $gv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$ | $= gv[\![e]\!] \cup (\bigcup gv[\![e_i]\!])$ |

**Рис. 8** HLL: множество свободных переменные выражения

| | |
|---|---|
| $fv[\![f_g]\!]$ | $= \{\}$ |
| $fv[\![v]\!]$ | $= \{v\}$ |
| $fv[\![c\ \overline{e_i}]\!]$ | $= \bigcup fv[\![e_i]\!]$ |
| $fv[\![\lambda v \to e]\!]$ | $= fv[\![e]\!] \setminus \{v\}$ |
| $fv[\![e_1\ e_2]\!]$ | $= fv[\![e_1]\!] \cup fv[\![e_2]\!]$ |
| $fv[\![case\ e\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!]$ | $= fv[\![e]\!] \cup (\bigcup fv[\![e_i]\!] \setminus \{\overline{v_{ik}}\})$ |
| $fv[\![let\ \overline{v_i = e_i;}\ in\ e]\!]$ | $= (fv[\![e]\!] \setminus (\bigcup v_i)) \cup (\bigcup fv[\![e_i]\!])$ |

Поскольку в HLL-выражениях присутствуют связанные переменные, нам необходимо детально формализовать работу со связанными переменными.

Чтобы показать, что переменная $f$ определена в программе (то есть в программе есть определение $f = e;$), мы будем дописывать к переменной индекс $g$ (global) – $f_g$.

**Определение 4** (Множества связанных, глобальных и свободных переменных выражения)**.** Множества $bv[\![e]\!]$, $gv[\![e]\!]$, $fv[\![e]\!]$ связанных переменных, глобальных и свободных переменных выражения $e$ определяются по правилам, представленных на Рис. 6, 7 и 8 соответственно.

**Определение 5** (Мультимножество связанных переменных выражения)**.** Правила, определяющие мультимножество $bv'[\![e]\!]$ связанных переменных выражения $e$, совпадают с правилами на Рис. 6 с учетом того, что результат операции – мультимножество.

**Замечание 6** (Соглашение об именах переменных)**.** Чтобы избежать конфликта имен, мы требуем, чтобы для любого выражения $e$ множества $bv[\![e]\!]$, $gv[\![e]\!]$, $fv[\![e]\!]$

попарно не пересекались и множество $bv'[\![e]\!]$ не содержало повторных элементов.

**Определение 7** (Обновление связанных переменных)**.** Обновление связанных переменных выражения $e$ – согласованная замена в выражении $e$ каждой переменной $v \in bv'[\![e]\!]$ на новую, ранее не встречавшуюся переменную. Будем обозначать операцию обновления переменных выражения $e$ как $fresh[\![e]\!]$.

**Определение 8** (HLL-подстановка)**.** Подстановкой будем называть конечный список пар вида $\theta = \{\overline{v_i := e_i}\}$, каждая пара в котором связывает переменную $v_i$ с ее значением $e_i$. Область определения $\theta$ определяется как $domain(\theta) = \{\overline{v_i}\}$. Область значений $\theta$ определяется как $range(\theta) = \{\overline{e_i}\}$.

Язык HLL основан на $\lambda$-исчислении. В $\lambda$-исчислении подстановка является фундаментальной операцией. Чтобы обеспечить корректность подстановки, выражения рассматриваются с точностью до переименования связанных переменных, то есть с точностью до $\equiv_\alpha$ ([2], 2.1.11). Таким образом, операция подстановки должна быть корректной на классах $\equiv_\alpha$ -эквивалентности ([2], Приложение C). То есть:

$$e \equiv_\alpha e',\ e_i \equiv_\alpha e'_i \Rightarrow e\{\overline{v_i := e_i}\} \equiv_\alpha e'\{\overline{v_i := e'_i}\}$$

Существуют различные способы решения этой задачи:

1. Можно рассматривать "канонические" безымянные выражения, где связанные переменные не имеют имен [4]. В данном подходе целый класс $\alpha$-конгруэнтных обычных выражений соответствует ровно одному выражению с безымянными переменными. Такой подход хорош для машинных преобразований, но затруднителен для восприятия человеком.

2. Можно при применении операции подстановки заменять связанные переменные на "свежие" переменные по мере необходимости, чтобы избежать "захвата" связанных переменных [3, 21]. Такой подход неудобен в случае, когда подстановка является результатом некоторой операции (например, обобщения), – нужно учитывать, что какая-то составляющая подстановки реально не будет применяться.

3. Можно вообще избежать подстановок, используя механизм наподобие *явных подстановок* [1, 14]. Однако в случае суперкомпиляции применение такого подхода сильно бы усложнило конфигурационный анализ.

4. Можно расширить соглашение из Замечания 6 на случай применения подстановки – определить понятие корректности (или допустимой) подстановки по отношению к выражению $e$ и рассматривать только корректные подстановки.

В контексте суперкомпиляции (точнее, – в контексте нахождения тесного обобщения двух выражений) наиболее удобен последний вариант.

**Определение 9** (Допустимая HLL-подстановка)**.** HLL-подстановка $\theta$ является допустимой по отношению к выражению $e$, если

**Рис. 9** HLL: подстановка

| | | | |
|---|---|---|---|
| $v\theta$ | $=$ | $fresh[\![e]\!]$ | if $v := e \in \theta$ |
| | $=$ | $v$ | otherwise |
| $f_g\theta$ | $=$ | $f_g$ | |
| $(c\ \overline{e_i})\theta$ | $=$ | $c\ \overline{(e_i\theta)}$ | |
| $(\lambda v \to e)\theta$ | $=$ | $\lambda v \to (e\theta)$ | |
| $(e_1\ e_2)\theta$ | $=$ | $(e_1\theta)\ (e_2\theta)$ | |
| $(case\ e\ of\ \{\overline{p_i\ \to e_i;}\})\theta$ | $=$ | $case\ (e\theta)\ of\ \{\overline{p_i\ \to (e_i\theta);}\}$ | |
| $(let\ \overline{v_i = e_i;}\ in\ e)\theta$ | $=$ | $let\ \overline{v_i = (e_i\theta);}\ in\ (e\theta)$ | |

1. $bv[\![e]\!] \cap domain(\theta) = \emptyset$
2. $\forall e_i \in range(\theta) : bv[\![e]\!] \cap fv[\![e_i]\!] = \emptyset$

**Замечание 10** (Соглашение о подстановках)**.** В дальнейшем мы рассматриваем только допустимые подстановки.

Соглашение 10 сродни так называемому соглашению Барендрегта ([2], 2.1.13). Использование соглашений 6 и 10 позволяет нам обеспечить корректность простого ("наивного") применения подстановки:

**Определение 11** (Применение HLL-подстановки)**.** Результат применения подстановки $\theta$ к выражению $e$, $e\theta$ вычисляется по правилам на Рис. 9.

Стоит отметить, что при применении подстановки связанные переменные подставляемого выражения обновляются.

**Замечание 12** (Устранение `let`-выражений перед суперкомпиляцией)**.** Как и в [8] непосредственно перед суперкомпиляцией исходная программа $p$ преобразуется методом $\lambda$-лифтинга [6] в программу $p'$, – в результате в программе $p'$ отсутствуют `let`-выражения. Причины устранения `let`-выражений таковы: (1) для выражений с `let`-выражениями понятия переименования, частного случая и обобщения определяются гораздо сложнее, – нужно учитывать возможность перестановки связок в `let`-выражениях, (2) let-выражения появляются в суперкомпиляции в результате обобщения, – нужно различать `let`-выражения исходной программы и результаты обобщения, (3) поскольку суперкомпилятор HOSC предназначен для анализа программ, устранение `let`-выражений позволяет агрессивнее распространять позитивную информацию, что обеспечивает более глубокий анализ программы. Поэтому в дальнейшем, если это особо не оговорено, рассматриваются HLL-выражения без конструкций `let`.

**Определение 13** (Частный случай)**.** Выражение $e_2$ называется *частным случаем* выражения $e_1$, $e_1 \lessdot e_2$, если существует подстановка $\theta$ такая, что $e_1\theta \equiv e_2$. Для выражений языка HLL такая подстановка является единственной и обозначается $\theta = e_1 \circledS e_2$.

**Определение 14** (Переименование)**.** Выражение $e_2$ называется *переименованием* выражения $e_1$, $e_1 \simeq e_2$, если $e_1 \lessdot e_2$, и $e_2 \lessdot e_1$. Другими словами, $e_1$ и $e_2$ различаются только именами свободных переменных.

**Рис. 10** HLL: алгоритм обобщения

**Most specific generalization**

- $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$

**Common functor rule**

- $v \mathbin{\widetilde{\sqcap}} v = (v, \{\}, \{\})$
- $c\ \overline{e'_i} \mathbin{\widetilde{\sqcap}} c\ \overline{e''_i} = (c\ \overline{e_i},\ \bigcup \theta'_i,\ \bigcup \theta''_i)$, where
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $e'_1\ e'_2 \mathbin{\widetilde{\sqcap}} e''_1\ e''_2 = (e_1\ e_2,\ \theta'_1 \mathbin{\widetilde{\sqcap}} \theta'_2,\ \theta''_1 \mathbin{\widetilde{\sqcap}} \theta''_2)$, where
  - $(e_i, \theta'_i, \theta''_i) = e'_i \mathbin{\widetilde{\sqcap}} e''_i$
- $\lambda v' \to e' \sqcap \lambda v'' \to e'' = (e_g, \theta', \theta'')$, if $\theta'$ and $\theta''$ are allowed wrt $e_g$, where
  - $e_g = \lambda v \to e$
  - $(e, \theta', \theta'') = e'\{v' := v\} \mathbin{\widetilde{\sqcap}} e''\{v'' := v\}$
- $case\ e'_0\ of\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \mathbin{\widetilde{\sqcap}} case\ e''_0\ of\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\} = (e_g, \bigcup \theta'_i, \bigcup \theta''_i)$, if all $\theta'_i$ and $\theta''_i$ are allowed wrt $e_g$, where
  - $e_g = case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$
  - $(e_0, \theta'_0, \theta''_0) = e'_0 \mathbin{\widetilde{\sqcap}} e''_0$
  - $(e_i, \theta'_i, \theta''_i) = e'_i\{\overline{v'_{ik} := v_{ik}}\} \mathbin{\widetilde{\sqcap}} e''_i\{\overline{v''_{ik} := v_{ik}}\}$
- $e_1 \mathbin{\widetilde{\sqcap}} e_2 = (v, \{v := e_1\}, \{v := e_2\})$

**Common subexpression rule**

- $s(e, \{\}, \{\}) = s(e, \{\}, \{\})$
- $s(e, \{v_1 := e'\} \cup \theta', \{v_1 := e''\} \cup \theta'') = s'(s(e, \theta', \theta''))$ where
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e\{v_1 := v_2\},\ \theta'_1,\ \theta''_1)$
    if $\exists v_2 : \{v_2 := e'\} \in \theta'_1, \{v_2 := e''\} \in \theta''_1$
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e,\ \{v_1 := e'\} \cup \theta'_1,\ \{v_1 := e''\} \cup \theta''_1)$
    otherwise

---

**Определение 15** (Обобщение)**.** Обобщением выражений $e_1$ и $e_2$, называется тройка $(e_g, \theta_1, \theta_2)$, где $e_g$ – выражение, $\theta_1$ и $\theta_2$ – подстановки, такие, что $e_g\theta_1 \equiv e_1$ и $e_g\theta_2 \equiv e_2$. Множество всех обобщений для выражений $e_1$ и $e_2$ обозначается как $e_1 \frown e_2$.

**Определение 16** (Тесное обобщение)**.** Обобщение $(e_g, \theta_1, \theta_2)$ называется наиболее

**Рис. 11** HLL: простое гомеоморфное вложение

**Embedding**

| | |
|---|---|
| $e' \trianglelefteq e''$ | if $e' \trianglelefteq_d e''$, $e' \trianglelefteq_c e''$ or $e' \trianglelefteq_v e''$ |

**Variables**

| | |
|---|---|
| $f_g \trianglelefteq_v f_g$ | |
| $v \trianglelefteq_v v'$ | |

**Coupling**

| | |
|---|---|
| $c\ \overline{e'_i} \trianglelefteq_c c\ \overline{e''_i}$ | if $\forall i : e'_i \trianglelefteq e''_i$ |
| $\lambda v' \to e' \trianglelefteq_c \lambda v'' \to e''$ | if $e' \trianglelefteq e''$ |
| $e'_1\ e'_2 \trianglelefteq_c e''_1\ e''_2$ | if $\forall i : e'_i \trianglelefteq e''_i$ |
| $case\ e'\ of\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \trianglelefteq_c case\ e''\ of\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\}$ | if $e' \trianglelefteq e''$ and $\forall i : e'_i \trianglelefteq e''_i$ |

**Diving**

| | |
|---|---|
| $e \trianglelefteq_d c\ \overline{e_i}$ | if $\exists i : e \trianglelefteq e_i$ |
| $e \trianglelefteq_d \lambda v_0 \to e_0$ | if $e \trianglelefteq e_0$ |
| $e \trianglelefteq_d e_1\ e_2$ | if $\exists i : e \trianglelefteq e_i$ |
| $e \trianglelefteq_d case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$ | if $\exists i : e \trianglelefteq e_i$ |

тесным обобщением выражений $e_1$ и $e_2$, если для любого обобщения $(e'_g, \theta'_1, \theta'_2) \in e_1 \frown e_2$ верно, что $e'_g \lessdot e_g$, т.е. $e_g$ является частным случаем $e'_g$. Операция нахождения тесного обобщения обозначается $\sqcap$: $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$.

**Определение 17** (Несопоставимые выражения)**.** Выражения $e_1$ и $e_2$ называются несопоставимыми, $e_1 \leftrightarrow e_2$, если $e_1 \sqcap e_2 = (v, \theta_1, \theta_2)$, то есть обощенное выражение является переменной.

В результате обобщения должна получиться такая тройка $(e_g, \theta_1, \theta_2)$, где $\theta_1$ и $\theta_2$ – допустимые по отношению к $e_g$ подстановки. В свете выбранных нами соглашений о переменных и подстановках, алгоритм нахождения тесного обобщения должен гарантировать допустимость подстановок обобщения.

**Определение 18** (Рекурсивный алгоритм обобщения HLL-выражений)**.** $e' \sqcap e'' = s(e' \mathbin{\widetilde{\sqcap}} e'')$, где операции $\mathbin{\widetilde{\sqcap}}$ и $s$ определены на Рис. 10.

Алгоритм 18 применим для нахождения тесного обобщения двух *любых выражений* языка HLL и учитывает требования соглашений 6 и 10. Правила применяются в порядке их перечисления. Переменные $v$ и $v_{ik}$ в 3-м, 4-м и 5-м правилах общего подвыражения – новые, ранее не встречавшиеся, переменные. Наиболее интересные детали алгоритма, учитывающие новые обстоятельства, подчеркнуты.

**Определение 19** (Простое гомеоморфное вложение $\trianglelefteq$)**.** Простое вложение HLL-выражений определяется индуктивно в соответствии с правилами на Рис. 11.

При проверке двух HLL-выражений на уточненное вложение используется таблица соответствия связанных переменных $\rho$:

$$\rho = \{(v'_1,\ v''_1),\ \ldots,\ (v'_n,\ v''_n)\}$$

**Рис. 12** HLL: уточненное гомеоморфное вложение

**Embedding**

$e' \trianglelefteq^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq^* e'' \mid_{\{\}}$

$e' \trianglelefteq_c^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq_c^* e'' \mid_{\{\}}$

$e' \trianglelefteq_d^* e'' \stackrel{\text{def}}{=} e' \trianglelefteq_d^* e'' \mid_{\{\}}$

**Embedding wrt table of bound variables**

$e' \trianglelefteq^* e'' \mid_\rho$ if $e' \trianglelefteq_d^* e \mid''_\rho$, $e' \trianglelefteq_c^* e'' \mid_\rho$ or $e' \trianglelefteq_v^* e'' \mid_\rho$

**Variables**

$f_g \trianglelefteq_v^* f_g$

$v' \trianglelefteq_v^* v'' \mid_\rho$ if $(v', v'') \in \rho$

$v' \trianglelefteq_v^* v'' \mid_\rho$ if $v' \notin domain(\rho)$ and $v'' \notin range(\rho)$

**Coupling**

$c\ \overline{e'_i} \trianglelefteq_c^* c\ \overline{e''_i} \mid_\rho$ if $\forall i : e'_i \trianglelefteq^* e''_i \mid_\rho$

$\lambda v' \to e' \trianglelefteq_c^* \lambda v'' \to e'' \mid_\rho$ if $e' \trianglelefteq^* e'' \mid_{\rho \cup \{(v', v'')\}}$

$e'_1\ e'_2 \trianglelefteq_c^* e''_1\ \underline{e''_2 \mid_\rho}$ if $\forall i : e'_i \underline{\trianglelefteq^* e''_i \mid_\rho}$

$case\ e'\ of\ \{c_i\ \overline{v'_{ik}} \to e'_i;\} \trianglelefteq_c^* case\ e''\ of\ \{c_i\ \overline{v''_{ik}} \to e''_i;\} \mid_\rho$
if $e' \trianglelefteq^* e'' \mid_\rho$ and $\forall i : e'_i \trianglelefteq^* e''_i \mid_{\rho \cup \{\overline{(v'_{ik}, v''_{ik})}\}}$

**Diving** only if $fv(e) \cap domain(\rho) = \emptyset$

$e \trianglelefteq_d^* c\ \overline{e_i} \mid_\rho$ if $\exists i : e \trianglelefteq^* e_i \mid_\rho$

$e \trianglelefteq_d^* \lambda v_0 \to e_0 \mid_\rho$ if $e \trianglelefteq^* e_0 \mid_{\rho \cup \{(\bullet, v_0)\}}$

$e \trianglelefteq_d^* e_1\ e_2 \mid_\rho$ if $\exists i : e \trianglelefteq^* e_i \mid_\rho$

$e \trianglelefteq_d^* case\ e'\ of\ \{\overline{c_i\ v_{ik} \to e_i;}\} \mid_\rho$ if $e \trianglelefteq^* e' \mid_\rho$ or $\exists i : e \trianglelefteq^* e_i \mid_{\rho \cup \{\overline{(\bullet, v_{ik})}\}}$

**Определение 20** (Уточненное гомеоморфное вложение $\trianglelefteq^*$)**.** Уточненное вложение HLL-выражений определяется индуктивно в соответствии с правилами на Рис. 12.

Напомним, что для SLL-выражений верно следующее: если $e_1 \trianglelefteq_c e_2$, то обобщение $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$ – нетривиально ($e_g$ - не переменная). Это свойство сохраняется для уточненного гомеоморфного вложения HLL-выражений. Для простого гомеоморфного вложения HLL-выражений это свойство больше не сохраняется, – нам придется учитывать это при конструировании суперкомпилятора для языка HLL и (чтобы гарантировать завершаемость суперкомпилятора) расщеплять нижнюю конфигурацию без учета истории предыдущих вычислений.

# 4 Алгоритм суперкомпиляции в общем виде

Понятия прогонки, частичного дерева процессов, операций над частичным деревом процессов, используемые в данном разделе, были даны в [8]. Отстаточная про-

**Рис. 13** Алгоритм суперкомпиляции в общем виде

$t = (e \rightarrow)$
**while** $unprocessedLeaf(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaf(t)$
  $relAncs = computeRelAncs(\beta)$
  $\alpha = find(relAncs, \beta, whistle)$
  **if** $\alpha \neq \bullet$ *and* $\alpha.expr \simeq \beta.expr$ **then**
    $t = fold(t,\ \alpha,\ \beta)$
  **else if** $\alpha \neq \bullet$ *and* $\alpha.expr \lessdot \beta.expr$ **then**
    $t = abstract(t, \beta, \alpha)$
  **else if** $\alpha \neq \bullet$ **then**
    **if** $\alpha.expr \leftrightarrow \beta.expr$ **then**
      $t = split(t, \beta, \beta.expr)$
    **else**
      $t = abstract(t, \alpha, \beta)$
    **end**
  **else**
    $t = drive(t, \beta)$
  **end**
**end**

**Рис. 14** HLL: расщепление конфигурации

| | | |
|---|---|---|
| $split(t, \beta, e_1 e_2)$ | $=$ | $replace(t, \beta, e_s)$, where $e_s = let\ v_1 = e_1;\ v_2 = e_2;\ in\ v_1 v_2$ |
| $split(t, \beta, case\ v\ of\ \{\overline{p_i \rightarrow e_i;}\})$ | $=$ | $addChildren(t, \beta, [v, \overline{e_i}])$ |
| $split(t, \beta, case\ e\ of\ \{\overline{p_i \rightarrow e_i;}\})$ | $=$ | $replace(t, \beta, e_s)$, where $e_s = let\ v = e\ in\ case\ v\ of\ \{\overline{p_i \rightarrow e_i;}\}$ |
| $split(t, \beta, e)$ | $=$ | $drive(t, \beta)$ |

грамма строится по частичному дереву процессов по правилам построения остаточной программы отношения трансформации HOSC ([10], Рис. 15).

Алгоритм суперкопмиляции, представленный на Рис. 13 является обобщением алгоритма суперкомпиляции из [20].

Каждый вариант реализации алгоритма конкретизирует следующие детали:

1. *computeRelAncs* - выбор релевантных (для поиска совпадающих конфигураций, надконфигураций и гомеоморфно вложенных конфигураций) данному узлу предков

2. *whistle* - проверка на вложение

В случае HLL-выражений операция расщепления конфигурации, несопоставимой с вложенной конфигурацией, усложняется из-за наличия `case`-выражений, – необходимо учитывать связанные переменные. Одновременно, желательно, чтобы

**Рис. 15** Типы выражений

| | | |
|---|---|---|
| $expClass(let\ \overline{v_i = e_i}\ in\ e)$ | = | 0 |
| $expClass(v\ \overline{e_i})$ | = | 0 |
| $expClass(c\ \overline{e_i})$ | = | 0 |
| $expClass(\lambda v \rightarrow\ e)$ | = | 0 |
| $expClass(con\langle(\lambda v \rightarrow e_0)\ e_1\rangle)$ | = | 1 |
| $expClass(con\langle f_g\rangle)$ | = | 2 |
| $expClass(con\langle case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \rightarrow e_i;}\}\rangle)$ | = | 3 |
| $expClass(con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \rightarrow e_i;}\}\rangle)$ | = | 4 |

размер выражений в дочерних узлах был меньше строго меньше размера расщепляемого выражения, – чтобы гарантировать завершаемость суперкомпилятора.

**Определение 21** (Операция расщепления конфигурации split)**.** Операция split осуществляется в соответсвии с правилами на Рис. 14.

Особенность определенной операции split состоит в работе с `case`-выражениями. Если селектор `case`-выражения является переменной, то делаем шаг, похожий на прогонку, – рассматриваем ветви, но *не распространяем позитивную информацию* – таким образом, размер выражений в новых дочерних узлах будет строго меньше размера расщепляемого выражения. Если селектор не является переменной, то мы обобщаем селектор.

Недостатком расщепления является то, что расщепление происходит безотносительно к узлам-предкам – не учитывается история вычислений. Суперкомпилятор HOSC устроен так, что такого расщепления никогда не происходит, то есть *история вычислений учитывается всегда.*

# 5 Суперкомпилятор HOSC 1.5

Под суперкомпилятором HOSC далее понимается параметризованный суперкомпилятор $\mathbf{SC_{ijk}}$, позволяющий получать в качестве результата в общем случае несколько остаточных программ (в зависимости от значений параметров). Наличие нескольких остаточных программ бывает полезным при использовании суперкомпиляции для анализа программ, – например, для распознавания улучшающих лемм [12].

Прежде чем перейти к рассмотрению конкретных функций *whistle* и *computeRelAnc* реализованных в $\mathbf{SC_{ijk}}$, нам необходимо ввести несколько определений.

**Определение 22** (Классы HLL-выражений)**.** Все выражения языка HLL разбиваются на 5 классов в соответствии с правилами на Рис. 15.

**Определение 23** (Тривиальный узел)**.** Узел $\beta$ называется тривиальным, если находящееся в нем выражение $\beta.expr$ является наблюдаемым или `let`-выражением. $(expClass(\beta.expr) = 0)$

**Определение 24** ($\beta$-транзитный узел)**.** Узел $\beta$ называется $\beta$-транзитным, если $\beta.expr = con\langle(\lambda v_0 \rightarrow e_0)\ e_1\rangle$. $(expClass(\beta.expr) = 1)$

**Определение 25** (Глобальный узел)**.** Узел $\beta$ – глобальный, если $\beta.expr = con\langle case\ v\ \overline{e'_j}$ $(expClass(\beta.expr) = 4)$

**Определение 26** (Локальный узел)**.** Все узлы за исключением глобальных считаются локальными.

**Определение 27** (Кандидат на зацикливание)**.** Кандидатом на зацикливание являются все узлы, за исключением тривиальных и $\beta$-транзитных узлов. ($expClass(\beta.expr$ 1)

**Замечание 28** (Обоснование выбора кандидатов)**.** Если включать $\beta$-транзитные узлы в число кандидатов на зацикливание, то все рассматриваемые далее суперкомпиляторы показывают неудовлетворительные результаты. Исключение $\beta$-транзитных узлов из списка кандидатов является безопасным для завершаемости суперкомпилятора – типизация гарантирует, что в частичном дереве процессов не будет бесконечной ветки, на которой нет ни одного кандидата.

**Определение 29** (Релевантные с учетом контроля предки-кандидаты)**.** Пусть $\beta$ – узел-кандидат. Релевантные с учетом контроля предки-кандидаты узла $\beta$ определяются следующим образом:

- все глобальные предки $\overline{\alpha_i}$ узла $\beta$, являющиеся кандидатами, если $\beta$ – глобальный узел
- все локальные предки $\overline{\alpha_i}$ узла $\beta$, являющиеся кандидатами, такие, что на пути от $\alpha_i$ до $\beta$ не встречается глобальный узел, если $\beta$ – локальный узел

Рассмотрим конкретизации алгоритма – алгоритмы $SC_{ijk}$. Каждый алгоритм $SC_{ijk}$ зависит от трех параметров:

1. $i$ - Какое гомеоморфное вложение использовать в качестве свистка: $\trianglelefteq_c^*$ $(+)$ или $\trianglelefteq_c$ $(-)$?
2. $j$ - Разделять ли узлы на глобальные и локальные при поиске релевантных кандидатов [20] $(+)$ или не разделять $(-)$?
3. $k$ - Требовать ли дополнительно при поиске кандидата совпадение типов выражений $(+)$ или нет $(-)$?

Более формально, операции *whistle* и *computeRelAncs* для суперкомпилятора $\mathbf{SC_{ijk}}$ определяются следующим образом:

- $whistle(e_1, e_2) =$
  - $e_1 \trianglelefteq_c e_2$, если $j = -$ и $k = -$
  - $e_1 \trianglelefteq_c^* e_2$, если $j = +$ и $k = -$
  - $e_1 \trianglelefteq_c e_2$ и $expClass(e_1) = expClass(e_2)$, если $j = -$ и $k = +$
  - $e_1 \trianglelefteq_c^* e_2$ и $expClass(e_1) = expClass(e_2)$, если $j = +$ и $k = +$
- $computeRelAncs(\beta) =$
  - предки-кандидаты узла $\beta$, если $j = -$
  - предки-кандидаты узла $\beta$ с учетом контроля, если $j = +$

Таким образом, алгоритмы $SC_{ijk}$ дают нам восемь вариантов суперкомпилятора.

**Теорема 30** (Корректность)**.** *Все представленные суперкомпиляторы* $\mathbf{SC_{ijk}}$ *корректны в смысле операционной эквивалентности исходной и остаточной программы.*

*Доказательство.* Если убрать шаг расщепления `case`-выражения с переменной в качестве селектора, то все суперкомпиляторы удовлетворяют отношению трансформации *HOSC* [10]. Шаг расщепления `case`-выражения с переменной в качестве селектора присутствует в дефорестации. Корректность этого шага показана в [15]. □

**Теорема 31** (Завершаемость)**.** *Все представленные суперкомпиляторы завершаются на любой корректно типизированной программе.*

*Доказательство.* Типизация гарантирует, что в частичном дереве процессов на любой ветке последовательность идущих друг за другом $\beta$-транзитных узлов конечна, мы можем исключить их из рассмотрения. Отношения $\trianglelefteq, \trianglelefteq^*$ – well-quasi-order. Из Леммы 38 из [9] следует завершаемость суперкомпиляторов без учета глобального и локального контроля. Легко показывается, что учет глобального и локального контроля не влияет на завершаемость. Действительно, предположим, что суперкомпилятор не завершается. Локальный контроль гарантирует, что на любой бесконечной ветке все участки, состоящие из локальных узлов, конечны. Значит, на бесконечно ветке должно быть бесконечное число глобальных узлов, а глобальный контроль предотвращает такую ситуацию. □

Стоит отметить, что завершаемость всех суперкомпиляторов гарантируется более элегантным способом, нежели в суперкомпиляторе HOSC 1.1 [9], где кандидаты определялись на основании сравнения размеров выражения до и после редукции.

Дополнительное условие на совпадений типов выражений при тестировании на вложение в суперкомпиляторе HOSC 1.5 в некоторой степени аналогично “стековому” свистку Турчина [25]. Если представлять конфигурации в виде стека, то совпадение типов выражений аналогично совпадению префиксов стека, так как редексы находятся на вершине стека. А условие вложения через сцепление аналогично совпадению суффиксов стека.

# 6 Сравнение суперкомпиляторов

В некоторых работах, вышедших в последние годы, исследуется, как различные аспекты алгоритма суперкомпиляции влияют на качество оптимизации программ. Однако никто ранее не исследовал на практике, как различные детали суперкомпилятора влияют на способность к трансформационному анализу.

Для сравнения восьми вариантов суперкомпилятора используется модельная задача: доказательство эквивалентности выражений [11]. Рассматриваемые тестовые примеры используют функции высших порядков и оперируют потенциально бесконечными данными. Функции над парами, списками и числами, используемые

**Рис. 16** Тесты

| | | | |
|---|---|---|---|
| `length (concat xs)` | $\cong$ | `sum (map length xs)` | (1) |
| `map f (append xs ys)` | $\cong$ | `append (map f xs) (map f ys)` | (2) |
| `filter p (map f xs)` | $\cong$ | `map f (filter (compose p f) xs)` | (3) |
| `map f (concat xs)` | $\cong$ | `concat (map (map f) xs)` | (4) |
| `iterate f (f x)` | $\cong$ | `map f (iterate f x)` | (5) |
| `map (compose f g)` | $\cong$ | `compose (map f) (map g)` | (6) |
| `map f xs` | $\cong$ | `join xs (compose return f)` | (7) |

**Рис. 17** Сравнение суперкомпиляторов на тестах

| | $\mathbf{Sc}_{---}$ | $\mathbf{Sc}_{-+-}$ | $\mathbf{Sc}_{--+}$ | $\mathbf{Sc}_{-++}$ | $\mathbf{Sc}_{+--}$ | $\mathbf{Sc}_{++-}$ | $\mathbf{Sc}_{+-+}$ | $\mathbf{Sc}_{+++}$ |
|---|---|---|---|---|---|---|---|---|
| (1) | – | – | – | – | – | + | – | + |
| (2) | – | + | + | + | – | + | + | + |
| (3) | – | + | – | + | – | + | – | + |
| (4) | – | – | – | – | – | + | – | + |
| (5) | – | – | + | + | – | – | + | + |
| (6) | – | + | + | + | – | + | + | + |
| (7) | + | + | + | + | + | + | + | + |

в тестах, показаны на Рис. 18. Сами тесты представлены на Рис. 16. Тест считается пройденным суперкомпилятором, если он сумел привести оба выражения к одной и той же синтаксической форме. В противном случае — не пройденным.

Результаты эксперимента, представленные на Рис. 17, показывают, что наилучшим сочетанием параметров является использование уточненного гомеоморфного вложения, разделения узлов на локальные и глобальные и разделения выражений на классы в соответствии с типом редекса. То есть дополнительные приемы (разделение узлов на локальные и глобальные, требование совпадения редекса) не могут в полной мере дать тот эффект, который достигается за счет учета связанных переменных при использовании уточненного гомеоморфного вложения $\trianglelefteq_c^*$.

Рассмотрим особенности уточненного вложения $\trianglelefteq^*$ на примере преобразования выражения `map f (concat xs)` суперкомпиляторами $\mathbf{Sc}_{-++}$ и $\mathbf{Sc}_{+++}$. После нескольких шагов работы суперкомпилятора $\mathbf{Sc}_{-++}$ срабатывает свисток:

```
case
  case xs of {
    Nil → Nil;
    Cons v54 v55 → (append v54) (foldr Nil append v55);
  }
of {
  Nil → Nil;
  Cons v56 v57 →
    (λv58 v59 → Cons (f v58) v59)
      v56 (foldr Nil (λv60 v61 → Cons (f v60) v61) v57);
}
```

$\trianglelefteq_c$

```
  case
```

**Рис. 18** Определения функций для тестов

```
data List a = Nil | Cons a (List a);
data Nat = Z | S Nat;
data Boolean = True | False;
data Pair a b = P a b;

compose = λf g x → f (g x);
outl = λp → case p of { P a b → a;};
outr = λp → case p of { P a b → b;};
uncurry = λf p → case p of {P x y → f x y;};
curry = λf b c → f (P b c);
cond = λp f g a → case (p a) of {True → f a; False → g a;};
foldn = λc h x → case x of {
  Z → c;
  S x1 → h (foldn c h x1);
};
plus = foldn (λx → x) (λf y → S (f y));
foldr = λc h xs → case xs of {
  Nil → c;
  Cons y ys → h y (foldr c h ys);
};
concat = foldr Nil append;
sum = foldr Z plus;
filter = λp → foldr Nil
  (curry (cond (compose p outl) (uncurry (λx xs → Cons x xs)) outr));
iterate = λf x → Cons x (iterate f (f x));
length = foldr Z (λx y → S y);
join = λm k → foldr Nil (compose append k) m;
return = λx → Cons x Nil;
map = λf → foldr Nil (λx xs → Cons (f x) xs);
append = λxs ys → case xs of {
  Nil → ys;
  Cons x1 xs1 → Cons x1 (append xs1 ys);
};
```

---

```
    case v54 of {
      Nil → foldr Nil append v55;
      Cons v87 v88 → Cons v87 (append v88 (foldr Nil append v55));
    }
  of {
    Nil → Nil;
    Cons v89 v90 →
      (λv91 v92→  Cons (f v91) v92)
        v89 (foldr Nil (λv93 v94→  Cons (f v93) v94) v90);
  }
```

Верхнее выражение обобщается суперкомпилятором $\mathbf{Sc}_{-++}$ следующим образом:

**Рис. 19** `map f (concat xs)`: результат работы $\mathbf{Sc}_{-++}$

```
letrec f1 = λz → case z of { Nil →Nil; Cons v101 v102 →
Cons (f v101) (f1 v102); }
in
  f1
    (letrec
      g = λx → case x of {
          Nil →Nil;
          Cons v54 v55 →
            (letrec h = λy →case y of { Nil →g v55; Cons v118 v119 →
Cons v118 (h v119); } in (h v54));
        }
    in
      g xs)
```

**Рис. 20** `concat (map (map f) xs)`: результат работы $\mathbf{Sc}_{+++}$

```
letrec
  g = λx→
    case x of {
      Nil →Nil;
      Cons v225 v226 →
        (letrec h = λy→  case y of { Nil →(g v226); Cons v308 v309 →
Cons (f v308) (h v309); } in h v225);
    }
in
  g xs
```

```
let
v100 = case xs of {
  Nil →Nil;
  Cons v54 v55 →(append v54) (foldr Nil append v55);
}
in
case v100 of {
  Nil →Nil;
  Cons v89 v90 →
    (λv91 v92→  Cons (f v91) v92)
    v89 (foldr Nil (λv93 v94→  Cons (f v93) v94) v90);
}
```

При использовании уточненного вложения приведенные конфигурации не вкладываются через сцепление и преобразование завершается без применения операции обобщения конфигураций.

Результаты работ суперкомпиляторов $\mathbf{Sc}_{-++}$ и $\mathbf{Sc}_{+++}$ приведены на Рис. 19 и Рис. 20.

# 7 Обсуждение

Распознавание (потенциально) бесконечных ветвей дерева процессов и обобщение некоторых конфигураций (ради свертывания дерева процессов в конечный граф) является одной из центральных идей *метода суперкомпиляции* [24, 25]. Основной задачей при разработке конкретного *суперкомпилятора* является принятие разумных решений, какие именно конфигурации следует обобщать и до какой степени. Чрезмерно активное обобщение приводит к снижению глубины анализа программы. С другой стороны, недостаточно активное обобщение может приводить к тому, что сам процесс суперкомпиляции может не завершаться из-за того, что суперкомпилятор не сможет свернуть потенциально бесконечное дерево в конечный граф.

Идея использовать для распознавания потенциально бесконечных ветвей отношение гомеоморфного вложения ([19]) оказалась чрезвычайно плодотворной и работоспособной. Вторая плодотворная и работоспособная идея – обобщать при обнаружении вложения верхнюю (вкладывающуюся) или нижнюю (охватывающую) конфигурацию[5].

Однако, в каждой конкретной реализации суперкомпилятора, основанном на данных идеях, необходимо конкретизировать отношение гомеоморфного вложения[6] и алгоритм обобщения.

Первой работой, в которой это было сделано в полном объеме, была магистерская работа Сёренсена [17]. Вложение и обобщение было определено для функционального языка первого порядка, в выражениях которого есть только свободные переменнные. Тот же самый язык с добавлением явных case-выражений рассматривается в [19, 16]. В case-выражениях есть связанные переменные. Гомеоморфное вложение для выражений без связанных переменных адаптируется к выражениям со связанными переменными – связанные и свободные переменные рассматриваются как одинаковые объекты (сродни рассмотренному в данной работе отношению $\trianglelefteq$). Однако, алгоритм обобщения выражений со связанными переменными не представлен. К сожалению, алгоритм обобщения выражений первого порядка (в отличие от гомеоморфного вложения) не адаптируется простым способом для выражений высшего порядка.

Работы последних лет ([7, 13]) по суперкомпиляции также используют адаптированное вложение, но не описывают алгоритм обобщения в деталях.

Одной из причин трудностей формулировки алгоритма нахождения тесного обобщения для выражения высшего прорядка является то, что необходимо зафиксировать синтаксические соглашения для работы со связанными переменными, которые обычно принимаются неформально.

В данной работе полностью и формально описан алгоритм нахождения тесного обобщения для HLL-выражений. В основе алгоритма лежит выбранное синтаксическое соглашение о подстановках, допустимых по отношению к выражению.

Также, в данной работе показано, что использование такого вложения $\trianglelefteq$ для выражений высшего порядка в суперкомпиляторе, предназначенном для анализа программ, дает неудовлетворительные результаты. Использование же предложенного автором уточненного гомеоморфного вложения $\trianglelefteq^*$, учитывающего свойства

[5] Первый вариант является более распространенным.

[6] Отношение гомеоморфного вложения является достаточно общим понятием

связанных переменных, напротив, позволяет проводить более глубокие и содержательные преобразования программ [11, 12].

Ранее изучалось, как различные аспекты алгоритма суперкомпиляции влияют на качество оптимизации программ. Однако не исследовалось на практике, как различные детали алгоритма суперкомпиляции влияют на его пригодность в качестве инструмента для трансформационного анализа. В данной работе сравниваются восем вариантов суперкомпилятора для языка HLL. Для сравнения используется модельная задача: доказательство эквивалентности выражений. Результаты эксперимента, показывают, что наилучшим сочетанием параметров для рассматриваемого суперкомпилятора является использование уточненного гомеоморфного вложения, разделение узлов на локальные и глобальные и разделение выражений на классы в соответствии с типом редекса.

Однако, было бы интересно изучить и влияние других деталей. Например, в данной работе в случае вложения конфигураций всегда обобщается верхняя конфигурация. Но можно было бы снять это ограничение и разрешить суперкомпилятору обобщать и нижнюю конфигурацию.

## Благодарности

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

## Список литературы

[1] M. Abadi, L. Cardelli, P. Curien, and J. Lévy. Explicit substitutions. *Journal of functional programming*, 1(04):375–416, 1991.

[2] H. Barendregt. *The lambda calculus: its syntax and semantics*. North-Holland, 1984.

[3] H. Curry, R. Feys, and W. Craig. *Combinatory logic*, volume 1. North-Holland, 1958.

[4] N. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[5] R. Glück and M. H. Sørensen. A roadmap to metacomputation by supercompilation. In *Selected Papers From the International Seminar on Partial Evaluation*, volume 1110 of *LNCS*, pages 137–160, 1996.

[6] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional programming languages and computer architecture*, volume 201 of *LNCS*, pages 190–203, 1985.

[7] P. Jonsson and J. Nordlander. Positive Supercompilation for a higher order call-by-value language. In *IFL 2007*, pages 441–456, 2007.

[8] I. Klyuchnikov. Supercompiler HOSC 1.0: under the hood. Preprint 63, Keldysh Institute of Applied Mathematics, Moscow, 2009.

[9] I. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. Preprint 21, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[10] I. Klyuchnikov. Supercompiler HOSC: proof of correctness. Preprint 31, Keldysh Institute of Applied Mathematics, Moscow, 2010.

[11] I. Klyuchnikov and S. Romanenko. Proving the equivalence of higher-order terms by means of supercompilation. In *Perspectives of Systems Informatics*, volume 5947 of *LNCS*, pages 193–205, 2010.

[12] I. Klyuchnikov and S. Romanenko. Towards higher-level supercompilation. In *Second International Workshop on Metacomputation in Russia*, 2010.

[13] N. Mitchell and C. Runciman. A supercompiler for core haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes In Computer Science*, pages 147–164, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] K. H. Rose. Explicit substitution – tutorial & survey. Technical Report LS-96-3, BRICS, 1996.

[15] D. Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, 167(1-2):193–233, 1996.

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

[17] M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Københavns Universitet, Datalogisk Institut, 1994.

[18] M. H. Sørensen. Convergence of program transformers in the metric space of trees. In *Mathematics of Program Construction*, volume 1422 of *LNCS*, pages 315–337, 1998.

[19] M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Logic Programming: The 1995 International Symposium*, pages 465–479, 1995.

[20] M. H. Sørensen and R. Glück. Introduction to supercompilation. In *Partial Evaluation. Practice and Theory*, volume 1706 of *LNCS*, pages 246–270, 1998.

[21] A. Stoughton. Substitution revisited. *Theor. Comput. Sci.*, 59(3):317–325, 1988.

[22] The GHC Team. Haskell 2010 language report. `http://haskell.org/definition/haskell2010.pdf`, 2010.

[23] A. Tolmach, T. Chevalier, and The GHC Team. An external representation for the GHC core language. `http://www.haskell.org/ghc/docs/6.12.2/core.pdf`, 2010.

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

[25] V. F. Turchin. The algorithm of generalization in the supercompiler. In *Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop*, 1988.