И. Г. Ключников, Суперкомпиляция функций высших порядков, *Программные системы: теория и приложения*, 2010, том 1, выпуск 3, 37–71

ISSN 2079-3316 ПРОГРАММНЫЕ СИСТЕМЫ: ТЕОРИЯ И ПРИЛОЖЕНИЯ № 3(3), 2010, с. 37–71

И. Г. Ключников

Суперкомпиляция функций высших порядков

Аннотация. В работе описана внутренняя структура экспериментального суперкомпилятора HOSC. Дано полное описание всех существенных понятий и алгоритмов суперкомпилятора, работающего с функциональным языком высшего порядка (подмножеством языка Haskell). Особое внимание уделяется проблемам связанным с обобщением и отношением гомеоморфного вложения для выражений со связанными переменными.

Ключевые слова и фразы: суперкомпиляция, анализ программ, функциональное программирование.

Введение

Суперкомпиляция – метод преобразования программ, предложенный В.Ф. Турчиным в 70-х годах прошлого века [24].

В работе рассматривается специализатор HOSC – экспериментальный суперкомпилятор для простого функционального языка HLL, являющегося подмножеством языка Haskell.

Несмотря на то, что алгоритмы суперкомпиляции для функциональных языков с функциями высших порядков уже описывались ранее [8, 16], эти описания не являются исчерпывающими и самодостаточными, поскольку некоторые существенные детали в них опущены или описаны неформально.

Целью данной работы является *полное* и *формальное* описание алгоритмов суперкомпилятора HOSC. Данная работа является развитием предыдущих работ автора [9–11].

Исходный код суперкомпилятора HOSC публично доступен в сети Интернет по адресу `http://hosc.googlecode.com`.

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

1. Язык HLL: синтаксис и семантика

Входным языком суперкомпилятора HOSC является язык HLL (Higher-order Lazy Language), являющийся подмножеством ядра языка Haskell [6].

1.1. Синтаксис языка HLL

Синтаксис языка HLL представлен на Рис. 1. Программа на языке HLL состоит из определений типов данных и определений.

Левая часть определения типа содержит имя типа (точнее, имя конструктора типа), за которым следует список типовых переменных. Правая часть – декларации конструкторов данных (разделенные вертикальной чертой, то есть $\overline{dCon_i} \Rightarrow dCon_1 \mid \ldots \mid dCon_n$).

Выражение – это локальная переменная, конструктор, глобальная переменная, λ-абстракция, аппликация, `case`-выражение, локальное рекурсивное определение (`letrec`-выражение), локальное определение переменных (`let`-выражение) или выражение в скобках. Порядок привязок в `let`-выражениях и порядок ветвей в `case`-выражениях не является существенным.

Выражение e_0 в `case`-выражении называется селектором, выражения $\overline{e_i}$ – ветвями. Мы требуем, чтобы в case-выражении были перечислены *все* конструкторы соответствующего типа данных – т.е. образцы в case-выражении должны быть *полны и ортогональны*.

Мы будем использовать две записи для обозначения аппликации: $e_1\ e_2$ и $e_0\ \overline{e_i}$. В первом случае e_1 может быть любым выражением. Во втором случае мы требуем, чтобы список аргументов $\overline{e_i}$ был непустым, а выражение e_0 не было аппликацией.

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 1 (Множества связанных, глобальных и свободных переменных выражения). Множества $bv[\![e]\!]$, $gv[\![e]\!]$, $fv[\![e]\!]$ связанных, глобальных и свободных переменных выражения e определяются по правилам, представленных на Рис. 2, Рис. 3 и Рис. 4 соответственно.

$tDef ::= \mathbf{data}\ tCon = \overline{dCon_i};$	определение типа
$tCon ::= tn\ \overline{tv_i}$	конструктор типа
$dCon ::= c\ \overline{type_i}$	конструктор данных
$type ::= tv \mid tCon \mid type \to type \mid (type)$	типовое выражение
$prog ::= \overline{tDef_i}\ \overline{f_i = e_i};$	программа
$e ::= v$	переменная
$\mid c\ \overline{e_i}$	конструктор
$\mid f_g$	глобальная переменная
$\mid \lambda\ \overline{v_i} \to e$	λ-абстракция
$\mid e_1\ e_2$	аппликация
$\mid \mathbf{case}\ e_0\ \mathbf{of}\ \{\overline{p_i \to e_i;}\}$	`case`-выражение
$\mid \mathbf{letrec}\ f = e_0\ \mathbf{in}\ e_1$	локальное определение
$\mid \mathbf{let}\ \overline{v_i = e_i;}\ \mathbf{in}\ e$	`let`-выражение
$\mid (e)$	выражение в скобках
$p ::= c\ \overline{v_i}$	образец

Рис. 1. Синтаксис языка HLL

$$
\begin{array}{ll}
bv[\![f_g]\!] & = \{\} \\
bv[\![v]\!] & = \{\} \\
bv[\![c\ \overline{e_i}]\!] & = \bigcup bv[\![e_i]\!] \\
bv[\![\lambda v \to e]\!] & = bv[\![e]\!]\ \cup\ \{v\} \\
bv[\![e_1\ e_2]\!] & = bv[\![e_1]\!] \cup bv[\![e_2]\!] \\
bv[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] & = bv[\![e_0]\!] \cup (\bigcup bv[\![e_i]\!]) \cup (\bigcup v_{ik}) \\
bv[\![let\ \overline{v_i = e_i;}\ in\ e]\!] & = bv[\![e]\!] \cup (\bigcup bv[\![e_i]\!])\ \cup\ (\bigcup v_i) \\
bv[\![letrec\ f = e_1\ in\ e_2]\!] & = bv[\![e_1]\!] \cup bv[\![e_2]\!]\ \cup\ \{f\}
\end{array}
$$

Рис. 2. Множество связанных переменных выражения

Определение 2 (Мультимножество связанных переменных выражения). Правила, определяющие мультимножество $bv'[\![e]\!]$ связанных переменных выражения e, совпадают с правилами на Рис. 2 с учетом того, что результат операции – мультимножество.

$$
\begin{array}{ll}
gv[\![f_g]\!] & = \{f_g\} \\
gv[\![v]\!] & = \{\} \\
gv[\![c\ \overline{e_i}]\!] & = \bigcup gv[\![e_i]\!] \\
gv[\![\lambda v \to e]\!] & = gv[\![e]\!] \\
gv[\![e_1\ e_2]\!] & = gv[\![e_1]\!] \cup gv[\![e_2]\!] \\
gv[\![case\ e\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] & = gv[\![e]\!] \cup (\bigcup gv[\![e_i]\!]) \\
gv[\![let\ \overline{v_i = e_i;}\ in\ e]\!] & = gv[\![e]\!] \cup (\bigcup gv[\![e_i]\!]) \\
gv[\![letrec\ f = e_1\ in\ e_2]\!] & = gv[\![e_1]\!] \cup gv[\![e_2]\!]
\end{array}
$$

Рис. 3. Множество глобальных переменных выражения

$$
\begin{array}{ll}
fv[\![f_g]\!] & = \{\} \\
fv[\![v]\!] & = \{v\} \\
fv[\![c\ \overline{e_i}]\!] & = \bigcup fv[\![e_i]\!] \\
fv[\![\lambda v \to e]\!] & = fv[\![e]\!] \setminus \{v\} \\
fv[\![e_1\ e_2]\!] & = fv[\![e_1]\!] \cup fv[\![e_2]\!] \\
fv[\![case\ e\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] & = fv[\![e]\!] \cup (\bigcup fv[\![e_i]\!] \setminus \{\overline{v_{ik}}\}) \\
fv[\![let\ \overline{v_i = e_i;}\ in\ e]\!] & = (fv[\![e]\!] \setminus (\bigcup v_i)) \cup (\bigcup fv[\![e_i]\!]) \\
fv[\![letrec\ f = e_1\ in\ e_2]\!] & = fv[\![e_1]\!] \cup fv[\![e_2]\!] \setminus \{f\}
\end{array}
$$

Рис. 4. Множество свободных переменных выражения

На использование имен переменных в выражениях накладываются следующие ограничения.

Соглашение 3 (Именование 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.

Соглашение 3 не только устраняет неоднозначность в классификации переменных, но также накладывает дополнительное ограничение на `let`-выражение: локальная переменная, введенная в `let`-выражении может использоваться только в теле `let`-выражения и не может использоваться в определении другой переменной того же `let`-выражения, – это гарантирует *независимость* локальных переменных `let`-выражения друг от друга.

Определение 4 (Замкнутое HLL-выражение). HLL-выражение называется замкнутым, если множество его свободных переменных пусто, т.е. $fv[\![e]\!] = \emptyset$

Правые части всех определений в программе должны быть замкнутыми.

Определение 5 (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 $fresh[\![e]\!]$.

Язык HLL типизирован по Хиндли-Милнеру [5, 18]. Далее везде предполагается, что рассматриваются только корректно типизированные программы и выражения.

1.2. Подстановка

Определение 6 (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 θ определяется как $domain(\theta) = \{\overline{v_i}\}$. Область значений θ определяется как $range(\theta) = \{\overline{e_i}\}$.

Язык HLL основан на λ-исчислении. В λ-исчислении подстановка является фундаментальной операцией. Чтобы обеспечить корректность подстановки, выражения рассматриваются с точностью до переименования связанных переменных, то есть с точностью до $\equiv_\alpha$ ([3], 2.1.11). Таким образом, операция подстановки должна быть корректной на классах $\equiv_\alpha$ -эквивалентности ([3], Приложение C). Т.е.

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

В контексте суперкомпиляции (точнее – в контексте нахождения тесного обобщения двух выражений) удобно ввести следующее соглашение о подстановках:

Определение 7 (Допустимая HLL-подстановка). Подстановка θ допустима по отношению к выражению e, если

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

$$
\begin{array}{lcll}
v\theta & = & fresh[\![e]\!] & \text{если } v := e \in \theta \\
 & = & v & \text{иначе} \\
f_g\theta & = & f_g & \\
(c\ \overline{e_i})\theta & = & c\ \overline{(e_i\theta)} & \\
(\lambda v \to e)\theta & = & \lambda v \to (e\theta) & \\
(e_1\ e_2)\theta & = & (e_1\theta)\ (e_2\theta) & \\
(case\ e\ of\ \{\overline{p_i\ \to e_i;}\})\theta & = & case\ (e\theta)\ of\ \{\overline{p_i\ \to (e_i\theta);}\} & \\
(let\ \overline{v_i = e_i;}\ in\ e)\theta & = & let\ \overline{v_i = (e_i\theta);}\ in\ (e\theta) & \\
(letrec\ f = e_1\ in\ e_2)\theta & = & letrec\ f = (e_1\theta)\ in\ (e_2\theta) &
\end{array}
$$

Рис. 5. HLL: подстановка

Соглашение 8 (Использование подстановок). В дальнейшем мы рассматриваем только допустимые подстановки.

Соглашение 8 сродни так называемому соглашению Барендрегта ([3], 2.1.13). Использование соглашений 3 и 8 позволяет нам обеспечить корректность простого (“наивного”) применения подстановки:

Определение 9 (Применение HLL-подстановки). Результат применения подстановки θ к выражению e, $e\theta$, вычисляется по правилам на Рис. 5.

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

1.3. Алгебра HLL-выражений

Определение 10 (Равенство выражений). Два выражения e_1 и e_2 считаются равными, если они различаются только именами соответствующих связанных переменных. Равенство выражений e_1 и e_2 записывается как $e_1 \equiv e_2$.

Нужно понимать, что равенство определяется с точностью до перестановок привязок в `let`-выражениях и ветвей в `case`-выражениях.

Определение 11 (Частный случай выражения). Выражение e_2 называется *частным случаем* выражения e_1, $e_1 \lessdot e_2$, если существует подстановка θ такая, что $e_1\theta \equiv e_2$. Для выражений языка HLL такая подстановка является единственной и обозначается $\theta = e_1 \lessdot e_2$.

Определение 12 (Переименование выражения). Выражение e_2 называется *переименованием* выражения e_1, $e_1 \simeq e_2$, если $e_1 \preccurlyeq e_2$, и $e_2 \preccurlyeq e_1$. Другими словами, e_1 и e_2 различаются только именами свободных переменных.

Определение 13 (Обобщение). Обобщением выражений e_1 и e_2, называется тройка $(e_g, \theta_1, \theta_2)$, где e_g – выражение, θ_1 и θ_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 14 (Тесное обобщение). Обобщение $(e_g, \theta_1, \theta_2)$ называется 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 $e'_g \preccurlyeq e_g$, т.е. e_g является частным случаем e'_g. Мы предполагаем, что определена операция $\sqcap$ такая, что $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$.

1.4. Семантика языка HLL

В данном разделе описывается семантика языка HLL, являющегося функциональным языком с передачей параметров по имени (call-by-name). При рассмотрении выражений языка HLL подразумевается, что задана некоторая программа, по отношению к которой и определяется смысл выражений (и может быть выполнено их вычисление).

Определение 15 (Контекст). Контекст – выражение с дырой $\langle\ \rangle$ вместо одного из подвыражений. $C\langle e\rangle$ - выражение, полученное из контекста заменой дыры на выражение e.

Определение 16 (Наблюдаемое выражение, контекст редукции, редекс). Грамматика декомпозиции HLL-выражений представлена на Рис. 6. Контекст редукции *con* является частным случаем контекста.

Любое выражение языка HLL можно представить либо в виде наблюдаемого выражения, либо в виде контекста редукции с помещенным в дыру редексом.

Лемма 17 (Декомпозиция HLL-выражений). Любое корректно типизированное выражений e есть либо наблюдаемое выражение: $e = obs$, либо представимо в виде контекста редукции *con* с помещенным в дыру редексом red: $e = con\langle red\rangle$.

$$
\begin{array}{lcl}
obs & ::= & v\ \overline{e_i} \\
 & | & c\ \overline{e_i} \\
 & | & (\lambda v \rightarrow\ e) \\
con & ::= & \langle\rangle \\
 & | & con\ e \\
 & | & case\ con\ of\ \{\overline{p_i \rightarrow e_i;}\} \\
red & ::= & f_g \\
 & | & (\lambda v \rightarrow e_0)\ e_1 \\
 & | & case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \rightarrow e_i;}\} \\
 & | & case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \rightarrow e_i;}\} \\
 & | & let\ \overline{v_i = e_i;}\ in\ e \\
 & | & letrec\ f = e_1\ in\ e_2
\end{array}
$$

Рис. 6. HLL: декомпозиция выражений

Определение 18 (Оператор неподвижной точки). Оператор неподвижной точки определяется как специальная функция fix:

$$fix = \lambda f \rightarrow f(fix\ f);$$

Определение 19 (Слабая головная нормальная форма). Выражение e языка HLL находится в слабой головной нормальной форме, если в нем на верхнем уровне находится конструктор (то есть $e = c\ \overline{e_i}$) или λ-абстракция (то есть $e = \lambda v \rightarrow\ e_1$).

Определение 20 (Раскрытие глобального определения). Пусть в программе есть определение $f_g =: e$, тогда:

$$unfold\langle f_g \rangle = f_g\{f_g =: e\}$$

Отметим, что, в силу определения операции подстановки, при раскрытии глобального определения происходит обновление связанных переменных в подставляемом определении.

Определение 21 (Шаг редукции). Шаг редукции $\mapsto$ – наименьшее отношение на множестве замкнутых HLL выражений, удовлетворяющее правилам на Рис. 7.

Определение 22 (Сходимость). Замкнутое выражение e сходится к слабой головной нормальной форме w, $e \Downarrow w$, если $e \mapsto^* w$.

$e \Downarrow$ обозначает, что существует w такое, что $e \Downarrow w$.

$$
\begin{array}{lll}
c\ \overline{e_i} & \mapsto c\ \overline{e_i} & (E_1)\\
\lambda v_0 \to e_0 & \mapsto \lambda v_0 \to e_0 & (E_2)\\
con\langle f_0\rangle & \mapsto con\langle unfold(f_0)\rangle & (E_3)\\
con\langle(\lambda v \to e_0)\ e_1\rangle & \mapsto con\langle e_0\{v\ :=\ e_1\}\rangle & (E_4)\\
con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\}\rangle & \mapsto con\langle e_j\{\overline{v_{jk}\ :=\ e'_k}\}\rangle & (E_5)\\
con\langle let\ \overline{v_i = e_i};\ in\ e\rangle & \mapsto con\langle e\{\overline{v_i := e_i}\}\rangle & (E_6)\\
con\langle letrec\ f = e\ in\ e'\rangle & \mapsto con\langle(\lambda f \to e')(fix\ (\lambda f \to e))\rangle & (E_7)
\end{array}
$$

Рис. 7. Операционная семантика HLL

```
data List a = Nil | Cons a (List a);

const   = λx → (λy → x);
iterate = λf z → Cons z (iterate f (f z));
```

Рис. 8. Пример программы на языке HLL

Определение 23 (Аварийное завершение с ошибкой времени выполнения). Вычисление замкнутого выражения e завершается с ошибкой времени выполнения, $e \Uparrow error$, если $e \mapsto^* e'$, e' не является слабой головной формой и к e' не применимо ни одно из правил Рис. 7.

Введем понятие операционной эквивалентности выражений [19]. Упрощенно говоря, два HLL-выражения эквивалентны, если они взаимозаменяемы в любых контекстах.

Определение 24 (Аппроксимация). Выражение e_1 является операционной аппроксимацией выражения e_2, $e_1 \sqsubseteq e_2$, если в любом контексте C, таком, что $C\langle e_1\rangle$ и $C\langle e_2\rangle$ – замкнутые выражения, из $C\langle e_1\rangle \Downarrow$ следует $C\langle e_2\rangle \Downarrow$ и из $C\langle e_1\rangle \Uparrow error$ следует $C\langle e_2\rangle \Uparrow error$.

Определение 25 (Эквивалентность). Выражение e_1 операционно эквивалентно выражению e_2, $e_1 \cong e_2$, если $e_1 \sqsubseteq e_2$ и $e_2 \sqsubseteq e_1$.

В определениях 24 и 25 для упрощения мы считаем, что рассматриваются только такие контексты C, в которых оба выражения $C\langle e_1\rangle$ и $C\langle e_2\rangle$ корректно типизируемы. В [10] рассматривается эквивалентность в общем случае.

Пример программы на языке HLL приведен на Рис. 8.

В соответствии с введенной семантикой результатом вычисления выражения

```
iterate (const Nil) Nil
```

будет

```
Cons Nil (iterate (const Nil) (const Nil Nil))
```

2. Отношение трансформации HOSC

Понятия “суперкомпиляции” и “суперкомпилятора” не совсем тождественны, поскольку конкретные суперкомпиляторы могут использовать методы анализа и преобразования программ, позаимствованные из других областей информатики (и не являющиеся суперкомпиляцией).

С другой стороны, принципы самой суперкомпиляции достаточно абстрактны и нуждаются в конкретизации в зависимости от рассматриваемого языка программирования и предполагаемой области применения суперкомпилятора. Необходимо уточнить, как представляются конфигурации, как строится частичное дерево процессов, как по частичному дереву процессов строится остаточная программа.

В данном разделе описывается *методологическая* основа суперкомпилятора HOSC в виде отношения трансформации (а не конкретного алгоритма) и обосновываются выбранные методы с точки зрения выбранной цели – трансформационного анализа программ.

2.1. Устранение локальных определений

Непосредственно перед суперкомпиляцией исходная программа p преобразуется методом λ-лифтинга [7] в программу p', – в результате в программе p' отсутствуют `let`-выражения и `letrec`-выражения. Решено устранять `let`-выражения по следующим причинам:

(1) Для выражений с `let`-выражениями понятия переименования, частного случая и обобщения программируются гораздо сложнее, – нужно учитывать возможность перестановки привязок в `let`-выражениях.

(2) Традиционно `let`-выражения используются в суперкомпиляции для обозначения результата обобщения. Мы не будем отступать от этой практики. Если не устранять `let`-выражения, то необходимо различать `let`-выражения исходной программы и результаты обобщения, что приводит к излишним усложнения алгоритмов.

(3) Поскольку суперкомпилятор HOSC предназначен для анализа программ, устранение `let`-выражений позволяет агрессивнее распространять позитивную информацию, что обеспечивает более глубокое преобразование программы. Поэтому в дальнейшем, если это особо не оговорено, рассматриваются HLL-выражения без конструкций `let` и `letrec`.

При разработке оптимизирующего суперкомпилятора для Haskell ([4, 16, 17]) устранять `let`-выражения таким образом недопустимо, так как при таком подходе остаточная программа может дублировать вычисления выражений, которые вычислялись в исходной программе один раз, и размер кода может сильно вырасти по сравнению с размером кода исходной программы.

2.2. Представление множеств

Построение дерева процессов и построение частичного дерева процессов являются метавычислениями. В метавычислениях рассматривают не только одиночное состояние вычисления, но также *множества* состояний вычислений. Как отмечено в [1], множества должны быть представлены конструктивно – в виде выражений некоторого языка.

Определение 26 (Конфигурация). HLL-выражения (без `let`-выражений и `letrec`-выражений) со свободными переменными будем называть HLL-конфигурациями, а свободные переменные – конфигурационными переменными.

Определение 27 (Декомпозиция конфигурации). Let-выражение

$$e_l = let\ \overline{v_i = e_i};\ in\ e_0$$

называется *декомпозицией* HLL-конфигурации e, $e_l \sqsubset e$, если каждое выражение e_i – конфигурация, $e_0 \lessdot e$, $e_0 \not\simeq e$ и $e = e_0\{\overline{v_i := e_i}\}$.

Соответственно, `let`-выражения представляют декомпозицию конфигураций.

Существует и другие способы задания языка для представления конфигураций, например в работе “Rethinking Supercompilation” [17] конфигурацией является только выражение в *специальной* форме. В работе “Supercompilation by Evaluation” [4] в качестве конфигурации рассматривается состояние абстрактной машины, состоящее из кучи, стека и активного выражения.

2.3. Построение частичного дерева процессов

Определение 28 (Дерево процессов). Деревом процессов называется ориентированное (возможно бесконечное) дерево, каждому узлу которого приписана HLL-конфигурация. Дуги дерева процессов будем обозначать $\rightarrow$.

В результате метавычислений в общем случае строится бесконечное дерево процессов. Задача суперкомпиляции – превратить его в конечный (возможно, циклический) граф, который называется *частичным деревом процессов*.

Определение 29 (Частичное дерево процессов). Частичное дерево процессов является расширением понятия дерева процессов и отличается от обычного дерева процессов следующими свойствами:

- В узлах частичного дерева процессов помимо конфигураций могут присутствовать декомпозиции конфигураций.
- В частичном дереве процессов могут быть *циклы*: пусть лист β является потомком узла α и выражение в узле β является переименованием выражения в узле α ($\beta.expr \simeq \alpha.expr$). Тогда можно провести специальную дугу $\beta \rightrightarrows \alpha$ из *повторного* (рекурсивного) узла β в *базовый* (или функциональный) узел α. Из узла может выходить не более одной специальной дуги $\rightrightarrows$.

Определение 30 (Обработанный лист). Лист дерева β называется обработанным, если находящееся в нем выражение $\beta.expr$ является константой, конфигурационной переменной или из него выходит специальная дуга $\rightrightarrows$.

Определение 31 (Завершенное частичное дерево процессов). Частичное дерево процессов является завершенным, если все его листья являются обработанными.

Операции над частичным деревом, которые нам понадобятся в дальнейшем, представлены на Рис. 9 в формальном виде. Некоторые операции не используются в данном разделе, но будут использоваться дальше – в разделе 4.

Остановимся подробно на операциях прогонки – $drive_0$ и $drive$. Операция $drive_0$ делает шаг прогонки без распространения позитивной информации. Оператор $\mathcal{D}_0$ (Рис. 10) принимает в качестве аргумента выражение, находящееся в листе, и возвращает ноль или более выражений $e_1, \ldots, e_k$. Результатом выполнения операции $drive_0(t, \beta)$

$incomplete(t)$	Возвращает $true$, если дерево t не является завершенным.
$trivial(node)$	Возвращает $true$ или $false$ в зависимости от того, является данный узел тривиальным или нет.
$children(t,\ \alpha)$	Возвращает упорядоченный список дочерних узлов узла α дерева t
$addChildren(t,\ \beta,\ es)$	Подвешивает к узлу β дерева t дочерние узлы и помещает в них выражения es. Количество подвешенных узлов совпадает с количеством элементов списка es.
$replace(t,\ \alpha,\ expr)$	Заменяет узел α на новый узел β такой, что $\beta.expr = expr$. Поддерево, имеющее своим корнем α, удаляется.
$ancestors(t,\ \beta)$	Возвращает узел всех предков узла β
$find(ns,\ \beta,\ p)$	Находит первый среди узлов ns узел α, такой, что верен предикат $p(\alpha.expr,\ \beta.expr)$.
$fold(t,\ \alpha,\ \beta)$	"Зацикливает" α и β: $\beta \rightrightarrows \alpha$.
$abstract(t,\ \alpha,\ \beta)$	$= replace(t,\ \alpha,\ let\ \overline{v_i := e_i;}\ in\ e_g)$, где $(e_g, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $e_g\theta_1 = e_g\{\overline{v_i := e_i}\} = let\ \overline{v_i = e_i;}\ in\ e_g$.
$\lceil \alpha \between t \rceil$	Возвращает все повторные узлы α, $\lceil \alpha \between t \rceil = [\overline{\beta_i}] : \beta_i \rightrightarrows \alpha$, или $\bullet$, если α не является базовым узлом.
$\lceil \alpha \Uparrow t \rceil$	Возвращает базовый узел α, $\lceil \alpha \Uparrow t \rceil = \beta : \alpha \rightrightarrows \beta$, или $\bullet$, если α не является повторным узлом.
$drive(t, \alpha)$	$= addChildren(t,\ \alpha,\ \mathcal{D}[\![\alpha.expr]\!])$ - делает шаг прогонки с распространением позитивной информации.
$drive_0(t, \alpha)$	$= addChildren(t,\ \alpha,\ \mathcal{D}_0[\![\alpha.expr]\!])$ - делает шаг прогонки без распространения позитивной информации.
$drive^*(t, \alpha)$	$= choice\{drive_0(t, \alpha),\ drive(t, \alpha)\}$
$decompose(t,\ \beta)$	$= replace(t,\ \beta,\ e_1)$, где e_1 – некоторая декомпозиция конфигурации $\beta.expr$.
$unprocessedLeaf(t)$	Возвращает самый левый необработанный лист α дерева t, или $\bullet$, если все листья обработаны.

Рис. 9. Операции над частичным деревом процессов

$$
\begin{array}{lcll}
\mathcal{D}_0[\![v\ \overline{e_i}]\!] & \Rightarrow & [v,\ \overline{e_i}] & (D_1^0)\\
\mathcal{D}_0[\![c\ \overline{e_i}]\!] & \Rightarrow & [\overline{e_i}] & (D_2^0)\\
\mathcal{D}_0[\![\lambda v_0 \to e_0]\!] & \Rightarrow & [e_0] & (D_3^0)\\
\mathcal{D}_0[\![con\langle f_0\rangle]\!] & \Rightarrow & [con\langle unfold(f_0)\rangle] & (D_4^0)\\
\mathcal{D}_0[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] & \Rightarrow & [con\langle e_0\{v_0 := e_1\}\rangle] & (D_5^0)\\
\mathcal{D}_0[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}\rangle]\!] & \Rightarrow & \left[con\langle e_j\{\overline{v_{jk} := e'_k}\}\rangle\right] & (D_6^0)\\
\mathcal{D}_0[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!] & \Rightarrow & \left[v\ \overline{e'_j},\ \overline{con\langle e_i\rangle}\right] & (D_7^0)\\
\mathcal{D}_0[\![let\ \overline{v_i = e_i;}\ in\ e]\!] & \Rightarrow & [e,\ \overline{e_i}] & (D_8^0)
\end{array}
$$

Рис. 10. Шаг прогонки без распространения позитивной информации

$$
\begin{array}{lcll}
\mathcal{D}[\![v\ \overline{e_i}]\!] & \Rightarrow & [v,\ \overline{e_i}] & (D_1)\\
\mathcal{D}[\![c\ \overline{e_i}]\!] & \Rightarrow & [\overline{e_i}] & (D_2)\\
\mathcal{D}[\![\lambda v_0 \to e_0]\!] & \Rightarrow & [e_0] & (D_3)\\
\mathcal{D}[\![con\langle f_0\rangle]\!] & \Rightarrow & [con\langle unfold(f_0)\rangle] & (D_4)\\
\mathcal{D}[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] & \Rightarrow & [con\langle e_0\{v_0 := e_1\}\rangle] & (D_5)\\
\mathcal{D}[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}\rangle]\!] & \Rightarrow & \left[con\langle e_j\{\overline{v_{jk} := e'_k}\}\rangle\right] & (D_6)\\
\mathcal{D}[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!] & \Rightarrow & \left[v\ \overline{e'_j},\ \overline{con\langle e_i\rangle\{v\ \overline{e'_j} := p_i\}}\right] & (D_7)\\
\mathcal{D}[\![let\ \overline{v_i = e_i;}\ in\ e]\!] & \Rightarrow & [e,\ \overline{e_i}] & (D_8)
\end{array}
$$

Рис. 11. Шаг прогонки с распространением позитивной информации

является подвешивание к листу β k дочерних узлов с выражениями $e_1, \ldots, e_k$. Операция $drive$ отличается от операции $drive_0$ только тем, что для получения выражений в дочерних узлах используется оператор $\mathcal{D}$ (Рис. 10).

$t = \boxed{e_0}$
while $incomplete(t)$ **do**
  $\beta = unprocessedLeaf(t)$
  $t = drive(t, \beta)$
end

Рис. 12. *HOSC*: построение дерева процессов для конфигурации e_0

$t = \boxed{e_0}$
while $incomplete(t)$ **do**
  $\beta = unprocessedLeaf(t)$
  $t = choice\{drive^*(t, \beta), generalize(t, \beta), fold(t, \beta)\}$
end

Рис. 13. *HOSC*: построение частичного дерева процессов для конфигурации e_0

Операторы $\mathcal{D}_0$ и $\mathcal{D}$ отличаются только седьмым правилом. При использовании оператора $\mathcal{D}$, когда необходимо сделать шаг прогонки для выражения вида $con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$, рассматриваются различные ветви с учетом того, что вычисление может достигнуть i-й ветви case-выражения только в том случае, когда $v\ \overline{e'_j}$ имеет вид p_i, – информация об этом распространяется в тело выбранной ветви. Оператор $\mathcal{D}_0$ не распространяет эту информацию. Особенностью оператора $\mathcal{D}_0$ является то, что при применении седьмого правила все выражения в дочерних узлах буду строго меньше по размеру, чем выражение в родительском узле – это факт будет использован в разделе 4, для того, чтобы гарантировать завершаемость суперкомпилятора.

Рассмотрим построение дерева процессов для конфигурации e_0. Соответствующий алгоритм представлен на Рис. 12. Вначале создается дерево t с единственным узлом, в который помещается стартовая конфигурация: $t = \boxed{e_0}$. Затем, пока есть хотя бы один необработанный узел, к нему применяются правила прогонки. В большинстве случаев дерево процессов будет бесконечным.

Перейдем к рассмотрению построения частичного дерева процессов для стартовой конфигурации e_0. Недетерминированный алгоритм построения частичного дерева процессов для выражения e_0 приведен

на Рис. 13. Оператор *choice* используется как оператор недетерминированного выбора – выполняется любая (допустимая) операция из аргументов оператора *choice*.

Алгоритм построения частичного дерева процесса процессов является расширением алгоритма построения дерева процессов – при построении дерева процессов для необработанного узла осуществляется шаг прогонки с распространением позитивной информации. При построении частичного дерева процессов по алгоритму на Рис. 13 разрешается для необработанного узла либо осуществить шаг прогонки с распространением позитивной информации, либо осуществить шаг прогонки без распространения позитивной информации, либо произвести декомпозицию конфигурации, либо, если среди предков есть совпадающая конфигурация (с точностью до переименования конфигурационных переменных), выбрать любую такую конфигурацию и провести специальную дугу $\rightrightarrows$.

Можно рекурсивно перечислить все частичные деревья процессов, которые могут быть построены по данному недетерминированному алгоритму для стартовой конфигурации e_0 и программы p – достаточно на каждом шаге рассмотреть все возможные выборы. Обозначим такое множество завершенных частичных деревьев процессов как $T_{HOSC}(p, e_0)$. Отметим, что в общем случае среди всех возможных трасс выборов построение частичного дерева процессов завершается только на малой части трасс.

2.4. Генерация остаточного выражения

Любое частичное дерево процессов $t \in T_{HOSC}(p, e_0)$, построенное для стартовой конфигурации e_0, является достаточным для вычисления любого замкнутого выражения $e \in e_0$ в любом контексте. Дерево процессов t также можно преобразовать в остаточное выражение (программу). Однако, как и при построении частичного дерева процессов, у нас есть степени свободы – можно разными способами преобразовать частичное дерево процессов в остаточное выражение, которое будет эквивалентно исходной.

В данном разделе описывается конкретный алгоритм преобразования графа конфигураций t в программу на языке HLL.

$\mathcal{C}\ [\![\alpha]\!]_{t,\Sigma}$

$\Rightarrow letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma'})\theta'\ in\ f'\overline{v'_i}$ если $\lceil \alpha \downarrow\!\!\searrow t \rceil \neq \bullet$ (C_1^*)

где

$[\overline{\beta_i}] = \lceil \alpha \downarrow\!\!\searrow t \rceil$, $\theta_i = \alpha.expr \otimes \beta_i.expr$,

$\overline{v'_i} = domain(\bigcup \overline{\theta_i})$, $\theta' = \{\overline{v'_i := v_i}\}$,

$\Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i})$, f' и $\overline{v_i}$ – новые

$\Rightarrow f'_{sig}\theta$ если $\lceil \alpha \uparrow\!\!\uparrow t \rceil \neq \bullet$ (C_2^*)

где

$f'_{sig} = \Sigma(\lceil \alpha \uparrow\!\!\uparrow t \rceil)$,

$\theta = \lceil \alpha \uparrow\!\!\uparrow t \rceil.expr \otimes \alpha.expr$

$\Rightarrow \mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma}$ иначе (C_3^*)

$\mathcal{C}'[\![let\ \overline{v_i = e_i};\ in\ e]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}}\}$ (C'_1)

$\mathcal{C}'[\![v\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}}$ (C'_2)

$\mathcal{C}'[\![c\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}}$ (C'_3)

$\mathcal{C}'[\![\lambda v_0 \to e_0]\!]_{t,\alpha,\Sigma} \Rightarrow \lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma}$ (C'_4)

$\mathcal{C}'[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma}$

$\Rightarrow case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'} of \{\overline{p_i \to \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma'};}\}$ (C'_5)

$\mathcal{C}'[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}$ (C'_6)

$\mathcal{C}'[\![con\langle case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}$ (C'_7)

$\mathcal{C}'[\![con\langle f\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}$ (C'_8)

Чтобы уменьшить громоздкость правил, принято следующее соглашение. Если в правой части правила используется γ_i, считается, что: $[\overline{\gamma_i}] = children(t,\ \alpha)$, – в этом случае все дочерние узлы рассматриваются единообразно. Если же в правой части используются γ' и γ'_i, считается, что: $[\gamma',\ \overline{\gamma'_i}] = children(t,\ \alpha)$, – это соглашение используется в случае, когда дочерний узел заведомо один (в правой части правила используется только γ') или же первый дочерний узел требует “особого рассмотрения”.

Рис. 14. *HOSC*: генерация остаточного выражения

Алгоритм (представленный на Рис. 14) определяется через два взаимно рекурсивных оператора (функции): $\mathcal{C}$ и $\mathcal{C}'$. Остаточная программа для графа конфигураций t определяется как

$$\mathcal{C}\ [\![t.root]\!]_{t,\{\}}$$

Для построения остаточной программы мы обходим частичное дерево сверху вниз (начиная, естественно, с корня). При обходе базового узла, генерируется определение рекурсивной функции в форме letrec-а. Соответствие между базовым узлом и сигнатурой заносится в Σ. Впоследствии Σ используется для генерации рекурсивного вызова.

Алгоритм обладает следующими особенностями:

(1) Остаточная программа является одним самодостаточным выражением без глобальных определений – все рекурсивные функции определяются локально.
(2) В остаточной программе есть только рекурсивные функции.
(3) При конструировании рекурсивных функций аргументами становятся только *изменившиеся* части конфигураций, что позволяет понизить арность рекурсивных функций в остаточной программе.

Как показывают эксперименты [12, 14], эти особенности положительно сказываются на способности суперкомпилятора $HOSC$ к распознаванию эквивалентности выражений и улучшающих лемм.

2.5. Отношение транформации HOSC

Определение 32 (Отношение трансформации $HOSC$). Пусть дана программа p, HLL-выражение e и остаточное выражение p'. Выражения e и e' связаны отношением трансформации $HOSC$ ($e \ \ HOSC_p \ \ e'$), если существует частичное дерево процессов $t \in T_{HOSC}(p, e)$ такое, что $e' = \mathcal{C}\ [\![t.root]\!]_{t,\{\}}$.

Отношение трансформации $HOSC$ задает для данной входной программы p и стартового выражения e множество остаточных выражений. Как было отмечено ранее, для целей анализа может быть полезно иметь несколько остаточных выражений, – возможно, что некоторые остаточные выражения будут легче поддаваться анализу, чем другие.

Теорема 33 (Корректность суперкомпилятора HOSC). Остаточное выражение, связанное со стартовым выражением отношением трансформации HOSC, и стартовое выражение операционно эквивалентны:

$$e\ HOSC_p\ e' \Rightarrow e \cong e'$$

Тесное обобщение

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

Правило общего функтора

- $v \widetilde{\sqcap} v = (v, \{\}, \{\})$
- $c\ \overline{e'_i} \widetilde{\sqcap} c\ \overline{e''_i} = (c\ \overline{e_i},\ \bigcup \theta'_i,\ \bigcup \theta''_i)$, где
  - $(e_i, \theta'_i, \theta''_i) = e'_i \widetilde{\sqcap} e''_i$
- $e'_1\ e'_2 \widetilde{\sqcap} e''_1\ e''_2 = (e_1\ e_2,\ \theta'_1 \widetilde{\sqcap} \theta'_2,\ \theta''_1 \widetilde{\sqcap} \theta''_2)$, где
  - $(e_i, \theta'_i, \theta''_i) = e'_i \widetilde{\sqcap} e''_i$
- $\lambda v' \to e' \sqcap \lambda v'' \to e'' = (e_g, \theta', \theta'')$, если θ' и θ'' допустимы для e_g, где
  - $e_g = \lambda v \to e$
  - $(e, \theta', \theta'') = \underline{e'\{v' := v\}} \widetilde{\sqcap} \underline{e''\{v'' := v\}}$
- $case\ e'_0\ of\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \widetilde{\sqcap} case\ e''_0\ of\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\} = (e_g, \bigcup \theta'_i, \bigcup \theta''_i)$, если все θ'_i и θ''_i допустимы e_g, где
  - $e_g = case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$
  - $(e_0, \theta'_0, \theta''_0) = e'_0 \widetilde{\sqcap} e''_0$
  - $(e_i, \theta'_i, \theta''_i) = \underline{e'_i\{\overline{v'_{ik} := v_{ik}}\}} \widetilde{\sqcap} \underline{e''_i\{\overline{v''_{ik} := v_{ik}}\}}$
- $e_1 \widetilde{\sqcap} e_2 = (v, \{v := e_1\}, \{v := e_2\})$

Правило общего подвыражения

- $s(e, \{\}, \{\}) = s(e, \{\}, \{\})$
- $s(e, \{v_1 := e'\} \cup \theta', \{v_1 := e''\} \cup \theta'') = s'(s(e, \theta', \theta''))$ где
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e\{v_1 := v_2\},\ \theta'_1,\ \theta''_1)$
    если $\exists v_2 : \{v_2 := e'\} \in \theta'_1, \{v_2 := e''\} \in \theta''_1$
  - $s'(e,\ \theta'_1,\ \theta''_1) = (e,\ \{v_1 := e'\} \cup \theta'_1,\ \{v_1 := e''\} \cup \theta''_1)$
    в противном случае

Рис. 15. HLL: алгоритм обобщения

3. Гомеоморфное вложение и обобщение конфигураций

В данном разделе в качестве HLL-выражений рассматриваются только конфигурации (то есть выражения без `let`-выражений и `letrec`-выражений).

3.1. Нахождение тесного обобщения

Алгоритм нахождения тесного обобщения для выражений (конфигураций) первого порядка хорошо описан в литературе по суперкомпиляции. Однако, в работах, описывающих суперкомпиляторы для языков высшего порядка ([8, 16]), вопрос нахождения тесного обобщения выражений со связанными переменными обходится стороной.

Если рассматривать λ-абстракции и `case`-выражения как специальные конструкторы и "наивным" образом расширить алгоритм из [21, 23], то в обобщении, найденном таким образом, могут появляться недопустимые подстановки (см. Определение 7). В результате обобщения должна получиться такая тройка $(e_g, \theta_1, \theta_2)$, где θ_1 и θ_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, полученных в результате обобщения.

Обобщение λ-абстракций и `case`-выражений необходимо рассматривать особым образом – если в результате решения подзадач обобщения соответствующих подвыражений возникает недопустимая подстановка, это означает, что обобщением рассматриваемых выражений является переменная.

Определение 34 (Алгоритм нахождения тесного обобщения конфигураций). $e' \sqcap e'' = s(e' \widetilde{\sqcap} e'')$, где операции $\widetilde{\sqcap}$ и s определены на Рис. 15.

Алгоритм 34 применим для нахождения тесного обобщения двух *любых конфигураций* языка HLL и учитывает требования соглашений 3 и 8. 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. Отметим, что алгоритм 34 сформулирован в рекурсивной форме. В литературе по суперкомпиляции алгоритм обобщения традиционно описывается в итеративной форме. Сформулировать алгоритм обобщения выражений со связанными переменными в итеративной форме представляется крайне затруднительно.

Вложение

$e' \trianglelefteq e''$ если $e' \trianglelefteq_v e''$, $e' \trianglelefteq_c e''$ или $e' \trianglelefteq_d e''$

Вложение переменных

$f_g \trianglelefteq_v f_g$

$v \trianglelefteq_v v'$

Сцепление (Coupling)

$c\ \overline{e'_i} \trianglelefteq_c c\ \overline{e''_i}$ если $\forall i : e'_i \trianglelefteq e''_i$

$\lambda v' \to e' \trianglelefteq_c \lambda v'' \to e''$ если $e' \trianglelefteq e''$

$e'_1\ e'_2 \trianglelefteq_c e''_1\ e''_2$ если $\forall i : e'_i \trianglelefteq e''_i$

$case\ e'\ of\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \trianglelefteq_c case\ e''\ of\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\}$
если $e' \trianglelefteq e''$ и $\forall i : e'_i \trianglelefteq e''_i$

Погружение (Diving)

$e \trianglelefteq_d c\ \overline{e_i}$ если $\exists i : e \trianglelefteq e_i$

$e \trianglelefteq_d \lambda v_0 \to e_0$ если $e \trianglelefteq e_0$

$e \trianglelefteq_d e_1\ e_2$ если $\exists i : e \trianglelefteq e_i$

$e \trianglelefteq_d case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}$ если $\exists i : e \trianglelefteq e_i$

Рис. 16. HLL: простое гомеоморфное вложение

Определение 35 (Несопоставимые конфигурации). Конфигурации 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 обобщенная конфигурация является переменной.

3.2. Гомеоморфное вложение

В литературе по суперкомпиляции для языков высшего порядка используется классическое гомеоморфное вложение, "адаптированное" для выражений со связанными переменными, – λ-абстракции и `case`-выражения рассматриваются как специальные конструкторы и связанные переменные и свободные переменные не различаются.

Определение 36 (Простое гомеоморфное вложение $\trianglelefteq$). Простое вложение HLL-выражений определяется индуктивно в соответствии с правилами на Рис. 16.

Однако, в контексте суперкомпиляции, простое гомеоморфное вложение обладает существенным недостатком: несопоставимые выражения могут вкладываться через сцепление: $e_1 \trianglelefteq_c e_2$. В результате

нам придется делать декомпозицию одного из выражений без учета истории построения частичного дерева, что противоречит принципу обобщения в суперкопиляции (см. [25]).

Однако, если различать свободные и связанные переменные и потребовать, чтобы связанные переменные могли вкладываться только в соответствующие связанные переменные, то можно сформулировать уточненное гомеоморфное вложение, не допускающее вложение через сцепление несопоставимых выражений.

При проверке двух HLL-выражений на уточненное вложение используется таблица соответствия связанных переменных ρ:

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

Определение 37 (Уточненное гомеоморфное вложение $\trianglelefteq^*$). Вложение $\trianglelefteq^*$ HLL-выражений определяется индуктивно в соответствии с правилами на Рис. 17.

При использовании уточненного вложения несопоставимые конфигурации не вкладываются через сцепление.

4. Суперкомпилятор HOSC

В разделе 2 суперкомпилятор для языка HLL был описан в виде *отношения трансформации HOSC*, – недетерминированного алгоритма получения эквивалентного выражения. В данном разделе будут рассмотрены детерминированные алгоритмы суперкомпиляции для языка, гарантированно завершающиеся на любой входной конфигурации и удовлетворяющие введенному отношению трансформации HOSC.

4.1. Схема суперкомпилятора

Уточненное вложение $\trianglelefteq^*$ (Рис. 17) является достаточно сильным усложнением простого вложения $\trianglelefteq$ (Рис. 16). Чтобы обосновать практичность уточненного гомеоморфного вложения применительно к использованию суперкомпилятора для трансформационного анализа, рассмотрим различные варианты суперкомпилятора (как с использованием адаптированного вложения, так и с использованием уточненного вложения) и сравним их на модельной задаче – доказательстве эквивалентности выражений.

Поскольку при использовании адаптированного вложения $\trianglelefteq$ несопоставимые выражения могут вкладываться через сцепление, нам

Вложение

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

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

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

Вложение с учетом таблицы связанных переменных

$e' \trianglelefteq^* e'' \mid_\rho$ если $e' \trianglelefteq_v^* e'' \mid_\rho$, $e' \trianglelefteq_d^* e'' \mid_\rho$ или $e' \trianglelefteq_c^* e'' \mid_\rho$

Вложение переменных

$f \trianglelefteq_v^* f$

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

$v' \trianglelefteq_v^* v'' \mid_\rho$ если $v' \notin domain(\rho)$ и $v'' \notin range(\rho)$

Сцепление (Coupling)

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

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

$e'_1\ e'_2 \trianglelefteq_c^* e''_1\ e''_2 \mid_\rho$ если $\forall i : e'_i \trianglelefteq^* e''_i \mid_\rho$

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

Погружение (Diving) только если $fv(e) \cap domain(\rho) = \emptyset$

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

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

$e \trianglelefteq_d^* e_0\ \overline{e_i} \mid_\rho$ если $\exists i : e \trianglelefteq^* e_i \mid_\rho$

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

Рис. 17. HLL: уточненное гомеоморфное вложение

$split(t, \beta, e_1 e_2) = replace(t, \beta, e_s)$, где
$e_s = let\ v_1 = e_1;\ v_2 = e_2;\ in\ v_1 v_2$

$split(t, \beta, case\ v\ of\ \{\overline{p_i \to e_i;}\}) = addChildren(t, \beta, [v, \overline{e_i}])$

$split(t, \beta, case\ e\ of\ \{\overline{p_i \to e_i;}\}) = replace(t, \beta, e_s)$, где
$e_s = let\ v = e\ in\ case\ v\ of\ \{\overline{p_i \to e_i;}\}$

$split(t, \beta, e) = drive(t, \beta)$

Рис. 18. HLL: расщепление конфигурации

придется делать декомпозицию текущей конфигурации. В случае языка SLL это делается достаточно просто – текущей конфигурацией является вызов функции, и мы отдельно рассматриваем аргументы функции. Такая декомпозиция обладает важным свойством, что размеры всех компонент получившейся декомпозиции меньше размера обобщаемого выражения, что критически важно для того, чтобы гарантировать завершаемость.

В случае HLL-выражений операция расщепления конфигурации, несопоставимой с вложенной конфигурацией, усложняется из-за наличия `case`-выражений, – необходимо учитывать связанные переменные. Одновременно, желательно, чтобы размер выражений в дочерних узлах был строго меньше размера расщепляемого выражения, – чтобы гарантировать завершаемость суперкомпилятора.

Определение 38 (Операция декомпозиции конфигурации *split*). Операция *split* осуществляется в соответствии с правилами на Рис. 18.

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 (это важно для завершаемости суперкомпилятора). Если селектор не является переменной, то мы обобщаем селектор.

Рассмотрим алгоритм построения частичного дерева процессов, представленный на Рис. 19. Алгоритм является обобщением алгоритмов для языка первого порядка, представленных в [20, 22, 23], и допускает следующую параметризацию:

(1) При поиске кандидата на зацикливание для листа β рассматриваются не все предки, а только $computeRelAncs(\beta)$.
(2) В качестве свистка рассматривается любой предикат $whistle$.

Соответственно для конкретной реализации HLL-суперкомпилятора необходимо определить операции $computeRelAncs$ и $whistle$.

4.2. Суперкомпилятор $\mathbf{SC_{ijk}}$

Под суперкомпилятором HOSC далее понимается параметризованный суперкомпилятор $\mathbf{SC_{ijk}}$, позволяющий получать в качестве

$t = \boxed{e_0}$
while $incomplete(t)$ **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. 19. HLL суперкомпилятор

$$
\begin{array}{lcl}
eClass(let\ \overline{v_i = e_i}\ in\ e) & = & 0 \\
eClass(v\ \overline{e_i}) & = & 0 \\
eClass(c\ \overline{e_i}) & = & 0 \\
eClass(\lambda v \rightarrow\ e) & = & 0 \\
eClass(con\langle(\lambda v \rightarrow e_0)\ e_1\rangle) & = & 1 \\
eClass(con\langle f_g\rangle) & = & 2 \\
eClass(con\langle case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \rightarrow e_i;}\}\rangle) & = & 3 \\
eClass(con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \rightarrow e_i;}\}\rangle) & = & 4
\end{array}
$$

Рис. 20. Типы выражений

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 [14].

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

Определение 39 (Классы HLL-выражений). Все выражения языка HLL разбиваются на 5 классов в соответствии с правилами на Рис. 20.

Определение 40 (Тривиальный узел). Узел β называется тривиальным, если находящееся в нем выражение $\beta.expr$ является наблюдаемым или `let`-выражением. ($eClass(\beta.expr) = 0$)

Определение 41 (β-транзитный узел). Узел β называется β-транзитным, если $\beta.expr = con\langle(\lambda v_0 \to e_0)\ e_1\rangle$. ($eClass(\beta.expr) = 1$)

Определение 42 (Глобальный узел). Узел β – глобальный, если $\beta.expr = con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$. ($eClass(\beta.expr) = 4$)

Определение 43 (Локальный узел). Все узлы за исключением глобальных считаются локальными.

Определение 44 (Кандидат на зацикливание). Кандидатом на зацикливание являются все узлы, за исключением тривиальных и β-транзитных узлов. ($expClass(\beta.expr) > 1$)

Замечание 45 (Обоснование выбора кандидатов). Если включать β-транзитные узлы в число кандидатов на зацикливание, то все рассматриваемые далее суперкомпиляторы показывают неудовлетворительные результаты. Исключение β-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 46 (Релевантные с учетом контроля предки-кандидаты). Пусть β – узел-кандидат. Релевантные с учетом контроля предки-кандидаты узла β определяются следующим образом:

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

Рассмотрим конкретизацию алгоритма – алгоритм SC_{ijk}. Алгоритм SC_{ijk} зависит от трех параметров:

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

Рис. 21. Тесты

(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 [23] $(+)$ 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 $eClass(e_1) = eClass(e_2)$, если $j = -$ и $k = +$
  - $e_1 \trianglelefteq_c^* e_2$ и $eClass(e_1) = eClass(e_2)$, если $j = +$ и $k = +$
- $computeRelAncs(\beta) =$
  - предки-кандидаты узла β, если $j = -$
  - предки-кандидаты узла β с учетом контроля, если $j = +$

Таким образом, алгоритм SC_{ijk} описывает восемь вариантов суперкомпилятора.

Теорема 47 (Корректность суперкомпилятора $\mathbf{SC_{ijk}}$). Все представленные суперкомпиляторы $\mathbf{SC_{ijk}}$ корректны в смысле операционной эквивалентности исходной и остаточной программы.

Теорема 48 (Завершаемость суперкомпилятора $\mathbf{SC_{ijk}}$). Все представленные суперкомпиляторы $\mathbf{SC_{ijk}}$ завершаются на любом корректном задании.

Полные доказательства теорем приведены в работах [10, 11].

```
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);
};
```

Рис. 22. Определения функций для тестов

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

Рис. 23. Сравнение суперкомпиляторов на тестах

4.3. Сравнение суперкомпиляторов

В некоторых работах, вышедших в последние годы, исследуется, как различные аспекты алгоритма суперкомпилятора 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 [12]. Рассматриваемые тестовые примеры используют функции высших порядков и оперируют потенциально бесконечными данными. Функции над парами, списками и числами, используемые в тестах, показаны на Рис. 22. Сами тесты представлены на Рис. 21. Тест считается пройденным суперкомпилятором, если он сумел привести оба выражения к одной и той же синтаксической форме, и не пройденным в противном случае.

Результаты эксперимента, представленные на Рис. 23, показывают, что наилучшим сочетанием параметров является использование уточненного гомеоморфного вложения, разделения узлов на локальные и глобальные и разделение 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 компенсировать учет связанных переменных при использовании уточненного 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
    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:

```
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 утонченного 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. 24 и Рис. 25.

```
letrec f1 = λz →
  case z of { Nil → Nil; Cons p q → Cons (f p) (f1 q); }
in
  f1
    (letrec
      g = λx → case x of {
          Nil → Nil;
          Cons k l →
            (letrec h = λy → case y of
              { Nil → g l; Cons s t → Cons s (h t); } in (h k));
        }
    in g xs)
```

Рис. 24. `map f (concat xs)`: результат работы $\mathbf{Sc}_{-++}$

```
letrec
  g = λx →
    case x of {
      Nil → Nil;
      Cons p q →
        letrec h = λy → case y of
          { Nil → (g q); Cons s t → Cons (f s) (h t); }
        in h p;
    }
in g xs
```

Рис. 25. `concat (map (map f) xs)`: результат работы $\mathbf{Sc}_{+++}$

5. Обзор литературы

Первоначально суперкомпиляция разрабатывалась для конкретного языка Рефал [24] и была сформулирована в его терминах. В настоящее время, наиболее продвинутым суперкомпилятором для языка Рефал является SCP4 [15].

В 90-е годы выходят работы, рассматривающие суперкомпиляцию более простых (по сравнению с Рефалом) функциональных языков. Цель этих работ состояла в том, чтобы разобраться, какие особенности суперкомпиляции (в формулировке Турчина) были обусловлены спецификой языка Рефал, а какие – носили фундаментальный характер.

Первое *полное* и *формальное* описание алгоритмов, составляющих необходимую часть любого суперкомпилятора (прогонка, обобщение, генерация остаточной программы), встречается в магистерской работе Сёренсена [20]. В этой работе описан суперкомпилятор простейшего функционального языка первого порядка (семантика вычислений – вызов по имени). Тот же самый суперкомпилятор с небольшими модификациями (расширен входной язык) описан в совместных статьях Глюка и Сёренсена [21, 22]. В работе [13] суперкомпилятор Сёренсена приведен в виде работающей программы на языке Scala.

Полное описание алгоритмов для суперкомпилятора языка TSG (плоский функциональный язык первого порядка) представлено у Абрамова [2].

В последние несколько лет возрос интерес к суперкомпиляции функциональных языков с функциями высших порядков [4, 8, 16, 17] для оптимизации.

В работах [8, 16] в качестве свистка используется адаптация классического гомеоморфного вложения $\trianglelefteq$ для языка первого порядка [21] на случай функций высшего порядка – связанные переменные рассматриваются таким же образом, как и конфигурационные переменные. Однако, как показывают эксперименты, использование такого вложения в суперкомпиляторе, предназначенном для анализа программ, дает неудовлетворительные результаты.

Использование же предложенного автором уточненного гомеоморфного вложения $\trianglelefteq^*$, учитывающего свойства связанных переменных, напротив, позволяет проводить более глубокие и содержательные преобразования программ [12, 14].

Более того, в упомянутых работах [8, 16] не приведен алгоритм нахождения тесного обобщения выражений со связанными переменными, – лишь сказано, что он является расширением алгоритма для нахождения тесного обобщения для выражений первого порядка. В данной работе полностью и формально описан алгоритм нахождения тесного обобщения для HLL-выражений. В основе алгоритма лежит

выбранное синтаксическое соглашение о допустимой по отношению к выражению подстановки.

Ранее изучалось, как различные аспекты алгоритма суперкомпилятора влияют на качество оптимизации программ. Однако никто ранее не исследовал на практике, как различные детали суперкомпилятора влияют на способность к трансформационному анализу. В данной работе сравниваются восем вариантов суперкомпилятора 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.

Представляет интерес изучить и влияние других деталей – например, в данной работе при вложении всегда обобщается верхняя конфигурация, однако можно обобщать и нижнюю конфигурацию.

Заключение

В работе описана внутренняя структура экспериментального суперкомпилятора HOSC, работающего с ядром языка Haskell.

Дано *полное и формальное* описание всех существенных понятий и алгоритмов, использованных в суперкомпиляторе. Особое внимание уделено гомеоморфному вложению и обобщению выражений со связанными переменными.

Список литературы

[1] Абрамов С. М. Метавычисления и их применение : Наука. Физматлит, 1995. ↑2.2

[2] Абрамов С. М., Пармёнова Л. В. Метавычисления и их применение. Суперкомпиляция : Университет города Переславля, 2006. ↑5

[3] Barendregt H. P. The lambda calculus: its syntax and semantics : North-Holland, 1984. ↑1.2, 1.2

[4] Bolingbroke M., Peyton Jones S. *Supercompilation by Evaluation* // Haskell 2010 Symposium, 2010. ↑2.1, 2.2, 5

[5] Damas L., Milner R. *Principal type-schemes for functional programs* // Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1982, p. 207–212. ↑1.1

[6] The GHC Team Haskell 2010 Language Report, 2010, http://haskell.org/definition/haskell2010.pdf. ↑1

[7] Johnsson T. *Lambda lifting: Transforming programs to recursive equations* // Functional Programming Languages and Computer Architecture. LNCS : Springer, 1985. Vol. **201**, p. 190–203. ↑2.1

[8] Jonsson P. A., Nordlander J. *Positive Supercompilation for a higher order call-by-value language* // IFL 2007, 2007, p. 441–456. ↑[], 3.1, 5

[9] Klyuchnikov I. *Supercompiler HOSC 1.0: under the hood* : Preprint. Moscow : Keldysh Institute of Applied Mathematics, 2009, no. 63. ↑[]

[10] Klyuchnikov I. *Supercompiler HOSC: proof of correctness* : Preprint. Moscow : Keldysh Institute of Applied Mathematics, 2010, no. 31. ↑1.4, 4.2

[11] Klyuchnikov I. *Supercompiler HOSC 1.1: proof of termination* : Preprint. Moscow : Keldysh Institute of Applied Mathematics, 2010, no. 21. ↑[], 4.2

[12] Klyuchnikov I., Romanenko S. *Proving the Equivalence of Higher-Order Terms by Means of Supercompilation* // Perspectives of Systems Informatics. LNCS : Springer, 2010. Vol. **5947**, p. 193–205. ↑2.4, 4.3, 5

[13] Klyuchnikov I., Romanenko S. *SPSC: a Simple Supercompiler in Scala* // PU'09 (International Workshop on Program Understanding), 2009. ↑5

[14] Klyuchnikov I., Romanenko S. *Towards Higher-Level Supercompilation* // Second International Workshop on Metacomputation in Russia, 2010. ↑2.4, 4.2, 5

[15] Nemytykh A. P. *The supercompiler SCP4: general structure* // PSI 2003. LNCS : Springer, 2003. Vol. **2890**, p. 162-170. ↑5

[16] Mitchell N., Runciman C. *A Supercompiler for Core Haskell* // Implementation and Application of Functional Languages. LNCS : Springer, 2008. Vol. **5083**, p. 147–164. ↑[], 2.1, 3.1, 5

[17] Mitchell N. *Rethinking Supercompilation* // ICFP 2010, 2010. ↑2.1, 2.2, 5

[18] Peyton Jones S. The Implementation of Functional Programming Languages : Prentice-Hall, Inc., 1987. ↑1.1

[19] Sands D. *Total correctness by local improvement in the transformation of functional programs* // ACM Trans. Program. Lang. Syst., 1996. **18**, no. 2, p. 175–234. ↑1.4

[20] Sørensen M. H. Turchin's Supercompiler Revisited: an Operational Theory of Positive Information Propagation : Master's thesis, 1994. ↑4.1, 5

[21] Sørensen M. H., Glück R. *An algorithm of generalization in positive supercompilation* // Logic Programming: The 1995 International Symposium ed. Lloyd J. W., 1995, p. 465–479. ↑3.1, 5

[22] Sørensen M. H., Glück R., Jones N. D. *A Positive Supercompiler* // Journal of Functional Programming, 1996. **6**, no. 6, p. 811-838. ↑4.1, 5

[23] Sørensen M. H., Glück R. *Introduction to Supercompilation* // Partial Evaluation. Practice and Theory. LNCS : Springer, 1998. Vol. **1706**, p. 246-270. ↑3.1, 4.1, 2

[24] Turchin V. F. *The concept of a supercompiler* // ACM Transactions on Programming Languages and Systems (TOPLAS), 1986. **8**, no. 3, p. 292-325. ↑[], 5

[25] Turchin V. F. *The algorithm of generalization in the supercompiler* // Partial Evaluation and Mixed Computation. Proceedings of the IFIP TC2 Workshop, 1988. ↑3.2

I. Klyuchnikov. *Higher-Order Supercompilation.*

Abstract. The paper describes the internal structure of HOSC, an experimental supercompiler dealing with programs written in a higher-order functional language (a subset of Haskell). A detailed and formal account is given of the concepts and algorithms the supercompiler is based upon. Particular attention is paid to the problems related to generalization and homeomorphic embedding of expressions with bound variables.

Key Words and Phrases: supercompilation, program analysis, functional programming.

Поступила в редакцию 03.10.2010. *Образец ссылки на статью:*

И. Г. Ключников. *Суперкомпиляция функций высших порядков* // Программные системы: теория и приложения : электрон. научн. журн. 2010. № 3(3), с. 37–71. URL: http://psta.psiras.ru/read/psta2010_3_37-71.pdf (дата обращения: 09.10.2010)