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

Илья Ключников

Суперкомпилятор HOSC: доказательство корректности

Москва
2010

Ilya G. Klyuchnikov. Supercompiler HOSC: proof of correctness

The paper presents the proof of correctness of an experimental supercompiler HOSC dealing with higher-order functions.

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

И.Г. Ключников. Суперкомпилятор HOSC: доказательство корректности

В работе приводится доказательство корректности экспериментального суперкомпилятора HOSC, работающего с функциями высших порядков.

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

Содержание

1 Введение

В [9] было описано внутреннее устройство суперкомпилятора HOSC. В данной работе доказывается корректность преобразований, осуществляемых суперкомпилятором HOSC, – показывается, что для любой входной программы, соответствующая ей остаточная программа эквивалентна исходной.

В контексте суперкомпиляции вопрос корректности преобразований слабо связан с проблемой завершаемости. В данной работе мы абстрагируемся от проблем завершаемости и переформулируем суперкомпиляцию функций высших порядков HOSC в виде *отношения трансформации*.

В [10] было показано, что HOSC 1.1 завершается на любой входной программе. Поскольку HOSC 1.1 является суперкомпилятором, удовлетворяющим сформулированному здесь отношению, мы таким образом покажем *полную корректность* суперкомпилятора HOSC 1.1.

Для доказательства корректности мы пользуемся операционной теорией улучшений Сэндса [22, 24], в основе которой лежит понятие стоимости вычисления и в качестве единицы стоимости используется вызов

функции. Говоря неформально, выражение e_1 является *улучшением* выражения e_0, если вычисление e_1 обходится не дороже, чем вычисление выражения e_0. Выражение e_1 является *строгим улучшением* выражения e_0, если e_1 является улучшением e_0 и e_1 и e_0 – эквивалентны.

Дефорестация ([14])– техника преобразования программ, являющаяся подмножеством суперкомпиляции. В работе [23] была доказана корректность дефорестации функций высших порядков с помощью теории улучшений (среди прочего, было показано, что результат дефорестации является строгим улучшением исходной программы).

Ключевым (и очень существенным) моментом доказательства корректности дефорестации является то, что при дефорестации программы свертка двух конфигураций c_1 и c_2 допускается только в случае если $c_1 = con\langle f \rangle$ (то есть следующим шагом вычислений будет замена вызова функции f на ее определение).

Отношение трансформации HOSC не имеет такого ограничения, – допускается свертка *любых* конфигураций.

Вначале рассматривается отношение $HOSC_0$, допускающее свертку только узлов вида $con\langle f \rangle$. $HOSC_0$ отличается от дефорестации функций высших порядков только наличием шага обобщения. Достаточно прямолинейно доказывается утверждение, что любая остаточная программа, удовлетворяющая отношению $HOSC_0$ является строгим улучшением исходной программы, а следовательно остаточная програма эквивалентна исходной.

Затем доказываются более общие утверждения о корректности отношений трансформации $HOSC_{1/2}$ и $HOSC$.

Также рассматривается вопрос о корректности суперкомпиляции с учетом типизации.

2 Входной язык

Суперкомпилятор HOSC преобразует программы на языке HLL. Синтаксис и семантика языка HLL подробно описаны в [9]. Здесь мы повторим только существенные для доказательства определения и обозначения.

Синтаксис бестипового варианта языка HLL. представлен на Рис. 1. Будем считать (вплоть до раздела 6) язык HLL бестиповым и игнорировать определения типов и вопросы типизации.

Непосредственно перед выполнением суперкомпиляции, над программой осуществляется преобразование λ-лифтинга, в результате чего конструкции *letrec* превращаются в глобальные определения. Во входной программе разрешаются let-выражения, однако перед суперкомпиляцией они также устраняются путем подстановки значений переменных. Суперкомпилируется программа уже без конструкций *letrec* и let-выражений.

$prog ::= e$ **where** $\overline{f_i = e_i;}$	program
$e ::= v$	variable
$\mid c\ \overline{e_i}$	constructor
$\mid f$	function
$\mid \lambda\ \overline{v_i} \to e$	λ-abstraction
$\mid e_1\ e_2$	application
$\mid$ **case** e_0 **of** $\{\overline{p_i \to e_i;}\}$	case-expression
$\mid$ **letrec** $f = e_0$ **in** e_1	local definition
$\mid$ **let** $\overline{v_i = e_i;}$ **in** e	let-expression
$\mid (e)$	parenthesized expression
$p ::= c\ \overline{v_i}$	pattern

Рис. 1: Язык HLL (бестиповый)

$$
\begin{array}{lcl}
obs & ::= & v\ \overline{e_i} \mid c\ \overline{e_i} \mid (\lambda v \to\ e) \\
con & ::= & \langle\rangle \mid con\ e \mid case\ con\ of\ \{\overline{p_i \to e_i;}\} \\
red & ::= & f \mid (\lambda v \to e_0)\ e_1 \mid case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\} \\
 & \mid & case\ c\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}
\end{array}
$$

Рис. 2: Декомпозиция выражений

$$
\begin{array}{lcll}
\mathcal{E}[\![c\ \overline{e_i}]\!] & \Rightarrow & c\ \overline{e_i} & (E_1) \\
\mathcal{E}[\![\lambda v_0 \to e_0]\!] & \Rightarrow & \lambda v_0 \to e_0 & (E_2) \\
\mathcal{E}[\![con\langle f_0 \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle unfold(f_0) \rangle]\!] & (E_3) \\
\mathcal{E}[\![con\langle (\lambda v \to e_0)\ e_1 \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle e_0\{v\ :=\ e_1\} \rangle]\!] & (E_4) \\
\mathcal{E}[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\} \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle e_j\{\overline{v_{jk}\ :=\ e'_k}\} \rangle]\!] & (E_5) \\
\mathcal{E}[\![let\ \overline{v_i = e_i;}\ in\ e_g]\!] & \Rightarrow & \mathcal{E}[\![e_g\{\overline{v_i := e_i}\}]\!] & (E_6)
\end{array}
$$

Рис. 3: Операционная семантика HLL

Любое выражение языка HLL можно представить либо в виде наблюдаемого выражения, либо в виде контекста с помещенным в дыру редексом. Грамматика такой декомпозиции представлена на Рис. 2.

Подстановкой называется конечный список пар вида

$$\theta = \{v_1 := e_1, v_2 := e_2, \ldots, v_n := e_n\}$$

каждая пара в котором связывает некоторую переменную v_i с ее значе-

нием e_i. Область определения подстановки обозначается $domain(\theta)$. Подстановка применяется к свободным переменным и записывается в постфиксной форме.

Let-выражения используются в частичном дереве процесса для представления обобщений. Синтаксис let-выражения имеет вид:

$$let\ \overline{v_i = e_i};\ in\ e_g,$$

где $\overline{v_i = e_i}$ – пары из соответствующей подстановки. По let-выражению однозначно восстанавливается исходное (обобщенное выражение):

$$let\ \overline{v_i = e_i};\ in\ e_g = e_g\{\overline{v_i := e_i}\}$$

Семантика вычислений с вызовом по имени представлена на Рис. 3. Так как let-выражения появляются в частичном дереве процессов, мы также определяем семантику их вычислений (правило E_6[1]). Семантика вычисления конструкции $letrec$ определяется в разделе 4. Если во время вычисления ни одно из правил на Рис. 3 не применимо, происходит ошибка времени выполнения (runtime error).

Выражение e_2 называется *частным случаем* выражения e_1, $e_1 \lessdot e_2$, если существует подстановка θ такая, что $e_1\theta \equiv e_2$. Операция нахождения такой подстановки обозначается $\theta = e_1 \circledcirc e_2$.

Множество всех выражений языка HLL обозначается $\mathcal{H}$.

3 Отношение трансформации HOSC

Суперкомпиляция программы состоит из двух этапов. Вначале строится частичное дерево процессов. Затем из частичного дерева процессов генерируется остаточная программы.

Частичное дерево процессов представляет собой дерево, в котором помимо обычных дуг, обозначаемых $\rightarrow$, могут быть особые зацикливающие дуги, ведущие от листа дерева к его предку и обозначаемые $\rightrightarrows$.

Дерево процесса является результатом метавычисления целевого выражения программы. Метавычисление определяется правилами прогонки, представленными на Рис. 4. В определенном смысле правила прогонки соответствуют семантике вычислений выражений со свободными переменными. На Рис. 6 перечислены операции над частичным деревом процессов.

В данном разделе описывается отношение трансформации $HOSC_{1/2}$, на основе которого формулируются отношения $HOSC_0$ и $HOSC$. Эти отношения трансформации отражают в некоторой степени конкретные

[1]Этого правила достаточно, так как при построении частичного дерева процессов let-выражения появляются только на верхнем уровне

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

Рис. 4: Прогонка

$t = (e \to)$
while $unprocessedLeaf(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaf(t)$
  $t = choice\{drive(t, \beta), generalize(t, \beta), fold(t, \beta)\}$
end

Рис. 5: $HOSC_{1/2}$, $HOSC$: построение частичного дерева процессов

“архитектурные” решения, принятые при разработке суперкомпилятора HOSC. Суперкомпиляция как отношение между исходной и остаточной программой формализована в общем виде в [8].

Недетерминированный алгоритм построения частичного дерева процессов для выражения e приведен на Рис. 5. Оператор *choice* используется как оператор недетерминированного выбора – выполняется любая (допустимая) операция из аргументов оператора *choice*.

По частичному дереву процессов t однозначно строится остаточная программа $prog'$ по правилам на Рис. 7.

$$prog' = \mathcal{C}\ [\![t.root]\!]_{t,\{\}}$$

Определение 1 (Отношение трансформации $HOSC_{1/2}$)**.** *Исходная программа prog и остаточная программа prog′ связаны отношением трансформации $HOSC_{1/2}$ ($prog\ HOSC_{1/2}\ prog'$), если среди реализаций частичного дерева процесса для программы prog по недетерминированному алгоритму на Рис. 5 существует дерево t такое, что $prog' = \mathcal{C}\ [\![t.root]\!]_{t,\{\}}$.*

Далее аналогичным образом определяются отношения $HOSC_0$ и $HOSC$.

$children(t,\ \alpha)$	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.
$replace(t,\ \beta,\ expr)$	Заменяет лист β на новый узел γ такой, что $\gamma.expr = expr$.
$fold(t,\ \beta)$	Если среди предков β существует узел α, такой что $\alpha.expr \simeq \beta.expr$, то "зацикливает" α и β: $\beta \rightrightarrows \alpha$.
$fold_0(t,\ \beta)$	Если $\beta.expr = con\langle f_0 \rangle$ и среди предков β существует узел α, такой что $\alpha.expr \simeq \beta.expr$, то "зацикливает" α и β: $\beta \rightrightarrows \alpha$.
$generalize(t,\ \beta)$	$replace(t,\ \beta,\ e_1)$, где $e_1 = let\ \overline{v_i = e_i;}\ in\ e_g$, $e_g\theta = e_g\{\overline{v_i := e_i}\} = \beta.expr$, θ – *любая* корректная подстановка
$\lceil \alpha \upharpoonleft\!\!\downarrow t \rceil$	Возвращает все повторные узлы α, $\lceil \alpha \upharpoonleft\!\!\downarrow t \rceil = [\overline{\beta_i}] : \beta_i \rightrightarrows \alpha$, или $\bullet$, если α не является базовым узлом.
$\lceil \alpha \uparrow\!\!\uparrow t \rceil$	Возвращает базовый узел α, $\lceil \alpha \uparrow\!\!\uparrow t \rceil = \beta : \alpha \rightrightarrows \beta$, или $\bullet$, если α не является повторным узлом.
$drive(t, \alpha)$	$= addChildren(t,\ \alpha,\ \mathcal{D}[\![\alpha.expr]\!])$ - делает шаг прогонки
$unprocessedLeaf(t)$	Возвращает *любой* необработанный лист α дерева t, или $\bullet$, если все листья обработаны. Лист считается обработанным, если он помечен переменной, конструктором без аргументов (константой) или если из него выходит зацикливающая дуга $\rightrightarrows$.

Рис. 6: Операции над частичным деревом процессов

Далее мы иногда вместо $prog\ HOSC_{1/2}\ prog'$ будем писать $prog' = \mathcal{SC}_{1/2}[\![prog]\!]$ (аналогично, $\mathcal{SC}_0[\![prog]\!]$ и $\mathcal{SC}[\![prog]\!]$).

Иногда мы будем писать $\mathcal{SC}_{1/2}[\![e]\!]$ вместо $\mathcal{SC}_{1/2}[\![prog]\!]$, где e – целевое выражение программы $prog$.

$$
\begin{array}{lll}
\mathcal{C}\,[\![\alpha]\!]_{t,\Sigma} & & \\
\quad \Rightarrow letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma'})\ in\ f'\overline{v_i} & \text{if } \lceil \alpha \downarrow\!\!\downarrow t \rceil \neq \bullet & (C_1) \\
\qquad \text{where} & & \\
\qquad\quad \overline{v_i} = fv(\alpha.expr) & & \\
\qquad\quad \Sigma' = \Sigma \cup (\alpha,\ \{f' := \lambda\overline{v_i} \to \alpha.expr\}),\ f' \text{ is fresh} & & \\
\quad \Rightarrow f'\overline{v_i} & \text{if } \lceil \alpha \uparrow\!\!\uparrow t \rceil \neq \bullet & (C_2) \\
\qquad \text{where} & & \\
\qquad\quad f' = domain(\Sigma(\lceil \alpha \uparrow\!\!\uparrow t \rceil)),\ \overline{v_i} = fv(\alpha.expr) & & \\
\quad \Rightarrow \mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma} & \text{otherwise} & (C_3)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{C}'[\![let\ \overline{v_i = e_i};\ in\ e]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma}}\} & (C_1') \\
\mathcal{C}'[\![v\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} & (C_2') \\
\mathcal{C}'[\![c\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} & (C_3') \\
\mathcal{C}'[\![\lambda v_0 \to e_0]\!]_{t,\alpha,\Sigma} \Rightarrow \lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C_4') \\
\mathcal{C}'[\![con\langle case\ v\ \overline{e_j'}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} & \\
\quad \Rightarrow case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'} of\{\overline{p_i \to \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma'};}\} & (C_5') \\
\mathcal{C}'[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C_6') \\
\mathcal{C}'[\![con\langle case\ c\ \overline{e_j'}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C_7') \\
\mathcal{C}'[\![con\langle f\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} & (C_8')
\end{array}
$$

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. 7: $HOSC_0$, $HOSC_{1/2}$: Генерация остаточной программы

4 Операционная теория улучшений

Определение 2 (Контекст)**.** *Контекст – выражение с дырой [] вместо одного из подвыражений. C[e] - выражение, полученное из контекста заменой дыры на выражение e.*

Определение 3 (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 $e = \lambda v \to\ e_1$).*

Определение 4 (Шаг редукции)**.** *Шаг редукции $\mapsto$ – наименьшее отношение на множестве замкнутых HLL выражений, удовлетворяющее*

правилам на Рис. 3.

Через $\mapsto^*$ обозначается транзитивное рефлексивное замыкание отношения $\mapsto$.

Определение 5 (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 $e \Downarrow w$.

Определение 6 (Аппроксимация). *Выражение e_1 является операционной аппроксимацией выражения e_2, $e_1 \sqsubseteq e_2$, если в любом контексте C, таком, что $C[e_1]$ и $C[e_2]$ – замкнутые выражения, из $C[e_1] \Downarrow$ следует $C[e_2] \Downarrow$.*

Определение 7 (Эквивалентность). *Выражение e_1 операционно эквивалентно выражению e_2, $e_1 \cong e_2$, если $e_1 \sqsubseteq e_2$ и $e_2 \sqsubseteq e_1$.*

Определение 8 (Улучшение). *Выражение e_2 является улучшением выражения e_1, $e_1 \trianglerighteq e_2$, если в любом контексте C, таком, что $C[e_1]$ и $C[e_2]$ – замкнутые выражения, если вычисление выражения $C[e_1]$ завершается за n вызовов функций, то вычисление выражения $C[e_2]$ завершается не более чем за n вызовов функций.*

Определение 9 (Строгое улучшение). *Выражение e_2 является строгим улучшением выражения e_1, $e_1 \trianglerighteq_s e_2$, если $e_1 \trianglerighteq e_2$ и $e_1 \ncong e_2$.*

Определение 10 (Эквивалентность стоимости вычислений). *Выражения e_1 и e_2 эквивалентны по стоимости вычислений $e_1 \lhd\!\rhd e_2$, если $e_1 \trianglerighteq e_2$ и $e_2 \trianglerighteq e_1$.*

Определение 11 (Оператор неподвижной точки). *Оператор неподвижной точки определяется как специальная функция fix:*

$$fix = \lambda f \to f(fix\ f)$$

Определение 12 (Letrec). *Мы рассматриваем letrec-выражения как макрос (синтаксический сахар), который раскрывается следующим образом:*

$$letrec\ f = e\ in\ e' \stackrel{def}{=} (\lambda f \to e')(fix\ (\lambda f \to e))$$

Определение 13 (Преобразование определений). *Пусть имеется программа с определениями $\{f_i = e_i;\}$ и программа с определениями $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$, где $\overline{e'_i}$ не зависят от $\overline{g_i}$, а $\overline{g_i}$ не зависят соответственно от $\overline{f_i}$. Считается, что $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ – преобразование $\{f_i = e_i;\}$.*

Теорема 14 (Теорема об улучшении [23]). *Пусть $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ – преобразование $\{f_i = e_i;\}$ такое, что $e_i \trianglerighteq e'_i$, то $f_i \trianglerighteq g_i$.*

Следствие 15. *Пусть* $\{g_i = e'_i\{\overline{f_i = g_i;}\};\}$ – *преобразование* $\{f_i = e_i;\}$ *такое, что* $e_i \trianglerighteq_{\lesssim_s} e'_i$, *то* $f_i \trianglerighteq_{\lesssim_s} g_i$.

Теорема 16 (Теорема о локальном улучшении [23])**.** *Если переменные* h *и* $\overline{x_i}$ – *это все свободные переменые выражений* e_0 *и* e_1 *и*

$$letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e_0 \trianglerighteq_{\lesssim_s} letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e_1$$

то для всех выражений e *справедливо*

$$letrec\ h = \lambda\overline{x_i} \to e_0\ in\ e \trianglerighteq_{\lesssim_s} letrec\ h = \lambda\overline{x_i} \to e_1\ in\ e$$

Утверждение 17 (Леммы улучшения [23])**.** *Справедливы следующие утверждения:*

1. *Если* $e \trianglerighteq_{\lesssim_s} e'$, *то* $C[e] \trianglerighteq_{\lesssim_s} C[e']$
2. $con\langle case\ e\ of\ \{\overline{p_i \to e_i;}\}\rangle \lhd\!\rhd case\ e\ of\ \{\overline{p_i \to con\langle e_i\rangle;}\}$
3. $con\langle case\ e\ of\ \{\overline{p_i \to e_i\{z := e\};}\}\rangle \trianglerighteq_{\lesssim_s} con\langle case\ e\ of\ \{\overline{p_i \to e_i\{z := p_i\};}\}\rangle$
4. *Если* $e \mapsto e'$, *то* $C[e] \trianglerighteq_{\lesssim_s} C[e']$
5. *Для всех выражений* e *и подстановок* θ *таких, что* $h \notin domain(\theta)$, *если* $e_0 \mapsto e_1$, *то*

   $letrec\ h = \lambda\overline{y_i} \to e_1\ in\ e\{z := e_0\theta\} \lhd\!\rhd letrec\ h = \lambda\overline{y_i} \to e_1\ in\ e\{z := (h\ \overline{y_i})\theta\}$

5 Доказательство корректности

В дальнейшем используется следующее соглашение: будем писать

$$prog \trianglerighteq_{\lesssim_s} prog',\ prog \cong prog'$$

если $e \trianglerighteq_{\lesssim_s} e'$, где e и e' – целевые выражения программ $prog$ и $prog'$ соответственно.

Под корректностью отношения трансформации T мы понимаем следующее:

$$prog\ T\ prog' \Rightarrow prog \cong prog'$$

5.1 Отношение $HOSC_0$

Докажем вначале корректность отношения трансформации $HOSC_0$, позволяющего осуществлять свертку только узлов вида $con\langle f\rangle$. Это отношение представлено на Рис. 8 и Рис. 7. При генерации остаточной программы по правилам на Рис. 7 осуществляется рекурсивный обход частичного

$t = (e \to)$
while $unprocessedLeaf(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaf(t)$
  $t = choice\{drive(t, \beta), generalize(t, \beta), fold_0(t, \beta)\}$
end

Рис. 8: $HOSC_0$: построение частичного дерева процессов

дерева процессов. При этом обходе в таблицу Σ заносятся соответствия базовых узлов и сигнатур новых рекурсивных функций.

Будем считать, что $\Sigma(\bullet) = \{\}$ (пустая сигнатура). Определим подстановку

$$\rho_\Sigma = range(\Sigma)$$

Теорема 18. *Для любого дерева процесса t, построенного по правилам на Рис. 8, при генерации остаточной программы по правилам на Рис. 7 для любого состояния обхода дерева верно:*

$$\alpha.expr \trianglerighteq_s (\mathcal{C}[\![\alpha.expr]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

Доказательство. Доказываем по индукции по структуре выражения $\alpha.expr$. Рассмотрим вначале правило (C_3), – необходимо доказать, что

$$\alpha.expr \trianglerighteq_s (\mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma})\rho_\Sigma$$

- (C'_1) Требуется показать, что

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

  По правилу (C'_1):

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

  В силу построения ρ_Σ:

  $$(\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\{\overline{v_i = \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}}\})\rho_\Sigma = (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)(\{\overline{v_i = (\mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma)}\})$$

  Таким образом, необходимо показать, что:

  $$e\{\overline{v_i = e_i;}\} \trianglerighteq_s (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)(\{\overline{v_i = (\mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma)}\})$$

  По построению дерева (правило D_8 4): $e = \gamma'.expr,\ e_i = \gamma'_i.expr$. По гипотезе индукции:

  $$e \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma, \quad e_i \trianglerighteq_s \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma$$

  Отсюда по лемме 17(1) следует:

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

- (C'_2) Требуется показать, что

$$v\ \overline{e_i} \trianglerighteq_s (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}})\rho_\Sigma = (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma})$$

  По гипотезе индукции:

$$e_i \trianglerighteq_s \mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma$$

  Отсюда по лемме 17(1) следует требуемое.

- (C'_3) Требуется показать, что

$$c\ \overline{e_i} \trianglerighteq_s (v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}})\rho_\Sigma = (c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma})$$

  По гипотезе индукции:

$$e_i \trianglerighteq_s \mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}\rho_\Sigma$$

  Отсюда по лемме 17(1) следует требуемое.

- (C'_4) Требуется показать, что

$$\lambda v_0 \to e_0 \trianglerighteq_s (\lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma})\rho_\Sigma = \lambda v_0 \to (\mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma)$$

  По гипотезе индукции:

$$e_0 \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma$$

  Отсюда по лемме 17(1) следует требуемое.

- (C'_5) Требуется показать, что

$$con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle \trianglerighteq_s (case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma} of \{\overline{p_i \to \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma};}\})\rho_\Sigma =$$

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

  По построению дерева (правило D_7 4):

$$\gamma'.expr = v\ \overline{e'_j}, \quad \gamma'_i.expr = con\langle e_i\{v\ \overline{e'_j} := p_i\}\rangle$$

  По гипотезе индукции:

$$v\ \overline{e'_j} \trianglerighteq_s \mathcal{C}[\![\gamma']\!]_{t,\Sigma}\rho_\Sigma, \quad con\langle e_i\{v\ \overline{e'_j} := p_i\}\rangle \trianglerighteq_s \mathcal{C}[\![\gamma'_i]\!]_{t,\Sigma}\rho_\Sigma$$

  Отсюда по леммам 17(1, 2, 3) следует требуемое.

- $(C'_6), (C'_7), (C'_8)$. В этих случаях

$$\alpha.expr \mapsto \gamma'.expr$$

  По гипотезе индукции и лемме 17(4) следует требуемое.

Лемма 19.

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'} \lessdot\!\!\gtrdot letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

Доказательство. Рассмотрим выражение

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'}$$

В соответствии с правилом (C_2) все свободные вхождения f' находятся в подвыражениях вида $f'\overline{v_i'}$, Предположим что таких вхождений n штук – $\overline{f'\overline{v_i}\theta_k}$, где θ_k – просто переименование переменных $\overline{v_i}$. Тогда

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'}) = e'\{\overline{z_k := f'\overline{v_i}\theta_k}\}$$

где $f' \notin fv(e')$. Значит,

$$(\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'} \equiv (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\{f' := \lambda\overline{v_i} \to \alpha.expr\}$$

$$\lessdot\!\!\gtrdot e'\{\overline{z_k := f'\overline{v_i}\theta_k}\}\rho_\Sigma\{f' := \lambda\overline{v_i} \to \alpha.expr\} \lessdot\!\!\gtrdot e'\{\overline{z_k := (\alpha.expr)\theta_k}\}\rho_\Sigma$$

Так как $\alpha.expr \mapsto \gamma'.expr$, то по лемме 17(5)

$$e'\{\overline{z_k := (\alpha.expr)\theta_k}\}\rho_\Sigma$$

$$\lessdot\!\!\gtrdot letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ e'\{\overline{z_k := f'\overline{v_i}\theta_k}\}\rho_\Sigma$$

$$\equiv letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

□

Рис. 9: Вспомогательная лемма

Рассмотрим теперь правило (C_2). Требуется показать, что

$$\alpha.expr \trianglerighteq_{\sim_s} (\mathcal{C}[\![\alpha.expr]\!]_{t,\Sigma})\rho_\Sigma = (f'\overline{v_i})\rho_\Sigma$$

По построению (операция $fold_0$ из на Рис. 6)

$$(f'\overline{v_i})\rho_\Sigma = (\lambda\overline{v_i} \to \alpha.expr)\overline{v_i}$$

Так как

$$(\lambda\overline{v_i} \to \alpha.expr)\overline{v_i} \lessdot\!\!\gtrdot \alpha.expr$$

следует требуемое.

Осталось рассмотреть правило (C_1). По построению надо показать, что

$$\alpha.expr \trianglerighteq_{\sim_s} (letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\ in\ f'\overline{v_i})\rho_\Sigma$$

Что равносильно

$$\alpha.expr \trianglerighteq_{\sim_s} letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\ in\ f'\overline{v_i}$$

Так как

$$\alpha.expr \mapsto \gamma'.expr$$

то по лемме 17(5)

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \alpha.expr \lhd\!\rhd letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i}$$

Так как $f' \notin fv(\alpha.expr)$, то

$$\alpha.expr \lhd\!\rhd letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i}$$

Таким образом, достаточно показать, что

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ f'\overline{v_i} \trianglerighteq_s$$

$$\trianglerighteq_s letrec\ f' = \lambda\overline{v_i} \to (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma\ in\ f'\overline{v_i}$$

По теореме 16 достаточно показать, что

$$letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \gamma'.expr \trianglerighteq_s$$

$$\trianglerighteq_s letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_\Sigma$$

Исходя из $letrec\ f' = \lambda\overline{v_i} \to \gamma'.expr\ in\ \gamma'.expr = \gamma'.expr$ и леммы 19, достаточно показать, что

$$\gamma'.expr \trianglerighteq_s (\mathcal{C}[\![\gamma'.expr]\!]_{t,\Sigma'})\rho_{\Sigma'}$$

что соответствует гипотезе индукции. □

Следствие 20 (Корректность $HOSC_0$)**.** $e \trianglerighteq_s \mathcal{SC}_0[\![e]\!]$

Доказательство. Следует из теоремы 18 и того, что $\mathcal{SC}_0[\![e]\!] = (\mathcal{C}[\![t.root]\!]_{t,\{\}})\{\}$, где t – дерево, сконструированное по алгоритму 8. □

5.2 Отношение $HOSC_{1/2}$

Трансформация $HOSC_0$ позволяет делать свертку только конфигураций вида $con\langle f\rangle$. В данном разделе покажем, что отношение трансформации $HOSC_{1/2}$, позволяющее делать свертку *любых* конфигураций, корректно.

Теорема 21 (Корректность $HOSC_{1/2}$)**.** *Для любого дерева процесса t, построенного по правилам на Рис. 5, при генерации остаточной программы по правилам на Рис. 7 верно: $e \cong \mathcal{SC}_{1/2}[\![e]\!]$*

Доказательство. Для доказательства корректности отношения $HOSC_{1/2}$ применяется следующий прием: рассмотрим в частичном дереве процессов t, допускаемом трансформацией $HOSC_{1/2}$, свернутые конфигурации c_1, c_2, ..., декомпозиция которых не соответствует виду $con\langle f \rangle$. Мы вставим непосредственно сверху узлов с конфигурациями c_1, c_2, ... узлы с конфигурациями c'_1, c'_2, ... вида $con\langle g \rangle$, где g - некоторая новая функция и $c'_i \to c_i$ и перенесем свертку со старых конфигураций на новые конфигурации. Мы анализируем историю появления конфигураций c_1, c_2, ..., и находим подвыражения в исходной программе e_1, e_2, ..., которые при прогонке трансформировались в конфигурации c_1, c_2, ... Заменяем найденные выражения e_i на $g\ \overline{e'_i}$. Следует отметить, что для этого возможно потребуется привести части исходной программы к суперкомбинаторному виду [20] с абстракцией максимально свободных выражений (см. пример в разделе 5.2.1).

Повторяем это преобразование, пока в частичном дереве процессов остаются свернутые конфигурации, не допускаемые $HOSC_0$. В итоге получаем частичное дерево процессов t' и новую программу $prog'$. В силу построения, остаточные программы, сгенерированные по деревьям t и t' будут совпадать. Обозначим эти программы $prog''$. Также в силу построения, дерево t' допустимо трансформацией $HOSC_0$. По определению e_i является строгим улучшением $g\ \overline{e'_i}$ (так как $g\ \overline{e'_i} \to e_i$), то есть $prog$ является строгим улучшением $prog'$ (по теореме об улучшениях). Имеем:

$$prog' \trianglerighteq_s prog,\ prog'' = \mathcal{SC}_0[\![prog']\!]$$

Отсюда (в силу того, что трансформация $HOSC_0$ – строго улучшающая) следует:

$$prog' \trianglerighteq_s prog''$$

По определению строгого улучшения

$$prog \cong prog''$$

Но с другой стороны

$$prog'' = \mathcal{SC}_{1/2}[\![prog]\!]$$

Следовательно

$$prog \cong \mathcal{SC}_{1/2}[\![prog]\!]$$

□

5.2.1 Пример

Рассмотрим простую программу $prog$ на Рис. 10. На Рис. 13 показано частичное дерево процессов t, построенное для этой программы трансформацией $HOSC_{1/2}$ (для уменьшения размеров дерева, тривиальная β-редукция пропущена). Отношение трансформации $HOSC_{1/2}$ позволяет

```
data Stream = S Stream;

case x of {S y1 → (S (id1 y1));} where

id1 = λx → case (id x) of { S y → S (id1 y);};
id = λx → x;
```

Рис. 10: Программа *prog*

```
data Stream = S Stream;

g x where

id1 = λx → g (id x);
g = λx → case x of {S y → S (id1 y);};
id = λx → x;
```

Рис. 11: Программа $prog'$

нам сделать свертку конфигураций с case-выражениями. Результат такой суперкомпиляции является программа $prog''$, показанная на Рис. 12.

Используя прием, описанный ранее, строим программу $prog'$. Программа *prog* является улучшением программы $prog'$, показанной на Рис. 11. Видно, что при построении $prog'$, нам пришлось абстрагировать подвыражение `id x` внутри определения функции `id1`. На Рис. 13 показано совмещение дерева t', построенного для $prog'$ трансформацией $HOSC_0$, и дерева t. Обратные дуги дерева t обозначены пунктиром, обратные дуги дерева t' обозначены сплошной линией. Очевидно, что остаточные программы для деревьев t и t' совпадают. Это является иллюстрацией того, что $prog'' \cong prog$.

5.3 Отношение $HOSC$

Правила генерации остаточной программы по правилам на Рис. 7 в некоторых случаях могут привести к излишнему увеличению арности рекурсивных функций (arity over-raising [21]), - см. пример в разделе 5.3.1.

По правилам на Рис. 7 количество аргументов рекурсивной функции, полученной при свертывании на конфигурации e определяется количеством свободных переменных этой конфигурации без учета того, как эти переменные влияют на дальнейшее вычисление.

Однако, количество аргументов рекурсивной функции можно уменьшить, если определять его количеством переменных в конфигурации e,

```
data Stream = S Stream;
letrec f = λp → case p of {S y → S (f y);} in f x
```

Рис. 12: Программа $prog''$

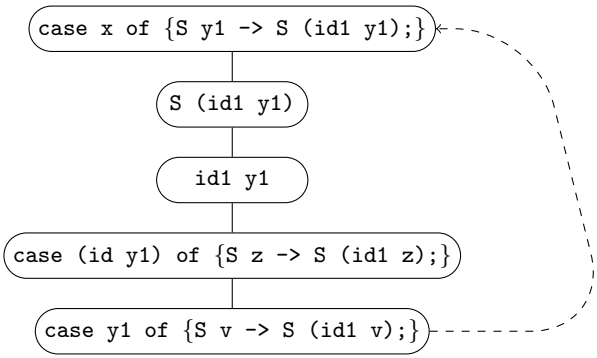

Рис. 13: Частичное дерево процессов для *prog* по $HOSC_{1/2}$

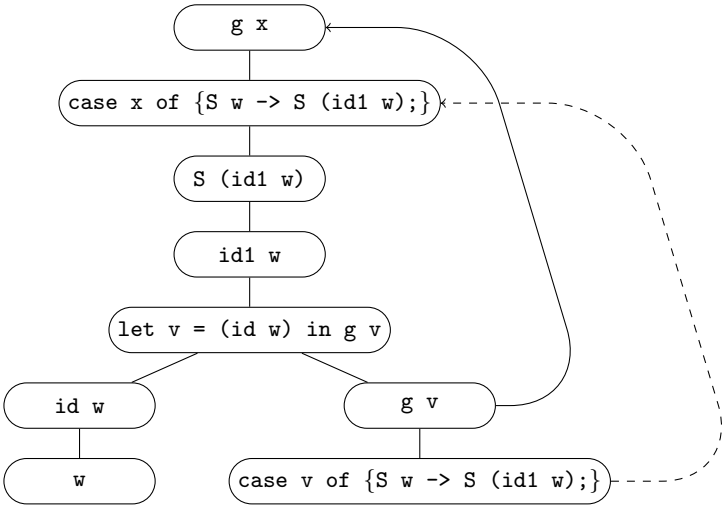

Рис. 14: Преобразование дерева $HOSC_{1/2}$ в дерево $HOSC_0$

которые изменились в повторных конфигурациях. Так и делается в правилах генерации остаточной программы 15.

Обозначим через $HOSC$ отношение трансформации, в котором ча-

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

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

where

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

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

$\Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i})$, f' and $\overline{v_i}$ are fresh

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

where

$f'_{sig} = \Sigma(\lceil \alpha \Uparrow t \rceil)$, $\theta = \lceil \alpha \Uparrow t \rceil.expr \otimes \alpha.expr$

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

Оператор $\mathcal{C}'$ совпадает с оператором $\mathcal{C}'$ на Рис. 7.

Рис. 15: $HOSC$: Генерация остаточной программы

стичное дерево процесса строится по правилам на Рис. 5, а программа генерируется по правилам на Рис. 15.

Отношение между $\mathcal{SC}_{1/2}[\![prog]\!]$ и $\mathcal{SC}[\![prog]\!]$ есть ничто иное, как частный случай λ-отбрасывания (λ-dropping [2]).

$$\mathcal{SC}_{1/2}[\![prog]\!] \xrightarrow{\lambda-dropping} \mathcal{SC}[\![prog]\!]$$

В [1, 26] показана корректность λ-отбрасывания. То есть,

$$prog \cong \mathcal{SC}_{1/2}[\![prog]\!] \quad \mathcal{SC}[\![prog]\!] \cong \mathcal{SC}_{1/2}[\![prog]\!]$$

Отсюда следует корректность отношения трансформации $HOSC$:

$$prog \cong \mathcal{SC}[\![prog]\!]$$

5.3.1 Пример

На Рис. 16 представлена простая программа, вычисляющая конкатенацию неизвестного списка `x` и непустого списка `Cons y z`. Частичное дерево процессов этой программы для трансформации $HOSC_{1/2}$ показано на Рис. 17 - сворачиваются конфигурации `app x (Cons y z)` и `app w (Cons y z)`. При построении остаточной программы при трансформации $HOSC_{1/2}$ получается рекурсивная функция от трех аргументов (Рис. 18), причем 2 из них передаются без изменений при рекурсивном вызове. Если суперкомпилировать выражение `app x (Cons y (Cons v w))` по отношению $HOSC_{1/2}$, то получится рекурсивная функция от 4 четырех аргументов и т.д. Если строить остаточную программа по дереву на Рис. 17 по отношению $HOSC$, то получится рекурсивная функция от одного аргумента (Рис 19).

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

app x (Cons y z)
where

app = λxs ys → case xs of {
    Nil → ys;
    Cons z zs → Cons z (app zs ys);
};
```

Рис. 16: `app x (Cons y z)`: исходная программа

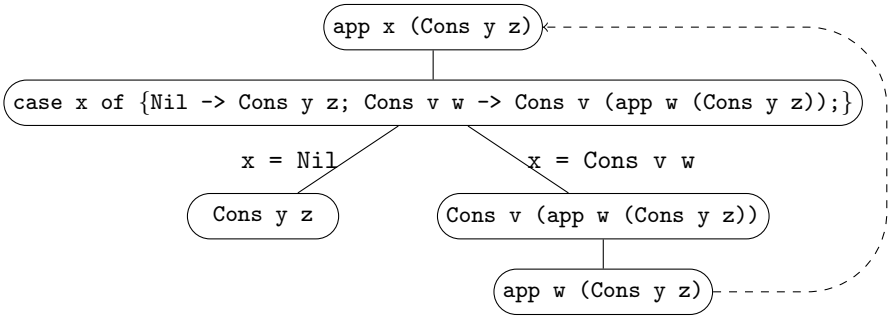

Рис. 17: `app x (Cons y z)`: частичное дерево процессов

6 Типизация и корректность

Во всех результатах, полученных до сих пор, язык HLL считался бестиповым (типизация игнорировалась): не было ошибок типизации, – были ошибки только времени выполнения.

Рассмотрим язык HLL (Рис. 20), в котором есть определения типов. Язык HLL типизирован по Хиндли-Милнеру и допускает явно приписать тип любому подвыражению.

Семантика типизированного HLL такая же как и у языка бестипового HLL (Рис. 3), за одним исключением – рассматриваются только корректно типизированные программы. Перед выполнением программы, интерпретатор проверяет, что программа корректно типизирована. Если программа содержит ошибку типизации, считаем, что интерпретатор сразу возвращает ошибку. Если рассматривать только неявно типизированные программы, то отношение трансформации HOSC не является корректным.

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

letrec f = λxs v zs → case xs of {
    Nil → Cons v zs;
    Cons x1 xs1 → Cons x1 (f xs1 v zs);
}
in f x y z
```

Рис. 18: `app x (Cons y z)`: остаточная программа

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

letrec f = λxs → case xs of {
    Nil → Cons y z;
    Cons x1 xs1 → Cons x1 (f xs1);
}
in f x
```

Рис. 19: `app x (Cons y z)`: остаточная программа с λ-отбрасыванием

Выведенные типы для остаточной программы могут быть более общими, нежели для исходной. Таким образом, суперкомпиляция может *расширять* область определения (в смысле типизации) при неявной типизации по Хиндли-Милнеру.

Рассмотрим программу на Рис. 21. Выведенный тип ее целевого выражения

```
(russel (MkU russel)) :: Bool
```

Самый простой вариант суперкомпиляции этой программы приведен на Рис. 22. Выведенный тип для остаточного выражения:

```
(letrec f=f in f) :: a
```

Как видно, выведенный тип остаточного выражения является более общим, чем тип исходного выражения, а значит, существуют контексты, в которых второе выражение может находится без ошибок типизации, а первое – нет.

То, что выводимая типизация по Хиндли-Милнеру не сохраняется при β-редукции, является известным фактом ([18]).

Эквивалентность двух выражений подразумевает, что эти два выражения в программах можно безопасно взаимозаменять.

Чтобы достичь безопасной взаимозаменяемости исходной и остаточной программы для типового варианта HLL, необходимо, чтобы типиза-

$tDef ::= \text{data } tCon = \overline{dCon_i};$	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
$prog ::= \overline{tDef_i}\ e\ \textbf{where}\ \overline{f_i = e_i};$	program
$e ::= e'$	implicitly typed
$\quad \mid e' :: type$	explicitly typed
$e' ::= v$	variable
$\quad \mid c\ \overline{e_i}$	constructor
$\quad \mid f$	function
$\quad \mid \lambda\ \overline{v_i} \to e$	λ-abstraction
$\quad \mid e_1\ e_2$	application
$\quad \mid \textbf{case}\ e_0\ \textbf{of}\ \{\overline{p_i \to e_i;}\}$	case-expression
$\quad \mid \textbf{letrec}\ f = e_0\ \textbf{in}\ e_1$	local definition
$\quad \mid \textbf{let}\ \overline{v_i = e_i;}\ \textbf{in}\ e$	let-expression
$\quad \mid (e)$	parenthesized expression
$p :: = c\ \overline{v_i}$	pattern

Рис. 20: Язык HLL (типизированный)

ция остаточной программы совпадала с типизацией исходной программы.

Чтобы исходное выражение и остаточное выражение e' оставались эквивалентными с учетом ошибок типизации, необходимо, чтобы в любом контексте C либо $C[e]$ и $C[e']$ корректно типизируемы, либо оба выражения $C[e]$ и $C[e']$ содержат ошибку типизации.

Можно достичь этого достаточно простым способом. Приписываем явно выведенные типы отсуперкомпилированному выражению (совпадает с типом исходного выражения) и каждой свободной переменной в остаточном выражении.

Остается тонкий случай, когда в остаточной программе нет некоторых переменных из исходной программы, – для полной корректности достаточно добавить искусственные аппликации λ-абстракций только для того, чтобы передать в остаточную программу информацию об устраненных переменных.

Возвращаясь к примеру суперкомпиляции программы на на Рис. 21. На Рис. 23 показано явное приписывание типов в остаточной программе.

```
data Bool = True | False;
data U = MkU (U → Bool);

russel (MkU russel) where

russel = λu → case u of {MkU p → p u;};
```

Рис. 21: `russel (MkU russel)`: исходная программа

```
data Bool = True | False;
data U = MkU (U → Bool);
(letrec f=f in f)
```

Рис. 22: `russel (MkU russel)`: остаточная программа

7 Обсуждение

В дальнейшем под *суперкомпилятором* H будем иметь в виду суперкомпилятор, описанный в [9, 10]. Соответственно, суперкомпилятор $H_{1/2}$ – это суперкомпилятор H, не делающий λ-отбрасывание. Суперкомпилятор H_0 – это суперкомпилятор $H_{1/2}$, делающий свертку только узлов вида $con\langle f \rangle$.

Суперкомпилятор $H_{1/2}$ имеет большую склонность к нормализации по сравнению с суперкомпилятором H_0. Рассмотрим программу $prog_1$ на Рис. 24. Через два шага редукции целевое выражение программы $prog_1$ совпадет с целевым выражением программы $prog$ (Рис. 10). Результат суперкомпиляции этих двух программ суперкомпилятором $HOSC_{1/2}$ один и тот же – $prog''$ (Рис. 12). Суперкомпилятор $HOSC_0$ выдаст разные результаты: $prog''$ (Рис. 12) для $prog_1$ и $prog_1'$ (Рис. 25) для $prog$.

Важно отметить, что в общем случае $\mathcal{H}_{1/2}[\![e]\!]$ и $\mathcal{H}[\![e]\!]$ не являются улучшением e, что может быть плохо с точки зрения оптимизации программ, но может оказаться полезным в случае анализа программ [11].

Суперкомпилятор H имеет большую склонность к сохранению "изоморфизма" между программами, нежели суперкомпилятор $H_{1/2}$.

Рассмотрим выражения `app x ys` и `app x (Cons y z)` (функция app определена на Рис. 16). Эти выражения соотносятся через подстановку:

$$app\ x\ (Cons\ y\ z) = (app\ x\ ys)\{ys := (Cons\ y\ z)\}$$

При преобразовании суперкомпилятором H также соотносятся и соответствующие остаточные программы (Рис. 19 и Рис. 27 соответственно):

$$\mathcal{H}[\![app\ x\ (Cons\ y\ z)]\!] = \mathcal{H}[\![app\ x\ ys]\!]\{ys := (Cons\ y\ z)\}$$

```
data Bool = True | False;
data U = MkU (U → Bool);
(letrec f=f in f) :: Bool
```

Рис. 23: `russel (MkU russel)`: остаточная программа с явной типизацией

```
data Stream = S Stream;

case (id x) of {S y1 → (S (id1 y1));} where

id1 = λx → case (id x) of { S y → S (id1 y);};
id = λx → x;
```

Рис. 24: Программа $prog_1$

Что неверно для суперкомпилятора $H_{1/2}$:

$$\mathcal{H}_{1/2}[\![app\ x\ (Cons\ y\ z)]\!] \neq \mathcal{H}_{1/2}[\![app\ x\ ys]\!]\{ys := (Cons\ y\ z)\}$$

Два перечисленных свойства (склонность к нормализации близких выражений и склонность к сохранению отношений) суперкомпилятора $HOSC$ хороши для использования его в качестве суперкомпилятора нулевого уровня при многоуровневой суперкомпиляции [12].

8 Обзор литературы

Основополагающая часть суперкомпиляции (прогонка) была изначально описана Турчиным как *эквивалентные* преобразования рекурсивных функций, описанных на языке РЕФАЛ [28]. Позднее были сформулированы другие составляющие суперкомпиляции [29, 30]. На сегодняшний день наиболее продвинутым суперкомпилятором языка РЕФАЛ является SCP4 [19]. Ради оптимизации SCP4 может жертвовать эквивалентностью преобразований - остаточная программа может успешно завершаться (и выдавать значение) в случаях, когда исходная программа зацикливается или завершается с ошибкой. Главная причина заключается в том, что РЕФАЛ – язык с вызовом по значению, в то время как прогонка “лениво” интерпретирует выражение. Доказательство частичной корректности суперкомпилятора SCP4 не опубликовано.

Затем суперкомпиляция была переформулирована в терминах более простого функционального языка *первого* порядка с ленивой семантикой. В магистерской работе Сёренсена [27] была показана корректность предложенного суперкомпилятора.

```
data Stream = S Stream;

case x of {
    S z → S (letrec f = λp → case p of {S y → S (f y);} in f z);
}
```

Рис. 25: Программа $prog'_1$

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

letrec f = λxs vs → case xs of {
    Nil → vs;
    Cons x1 xs1 → Cons x1 (f xs1 vs);
}
in f x ys
```

Рис. 26: $\mathcal{H}_{1/2}[\![app\ x\ ys]\!]$

Митчел описал суперкомпилятор Supero для подмножества языка Haskell [17, 16]. Доказательство корректности Supero не опубликовано.

Джонсон и Нордландер описали суперкомпилятор для функционального языка с функциями высших порядков с семантикой вызовов по значению [4] и показали его корректность [6].

Совсем недавно Мендель-Глисон и Гамильтон описали суперкомпиляцию для варианта системы F и показали её корректность [15] через бисимуляцию [3].

Крустев верифицировал модельный суперкомпилятор полностью механически [13].

Новизна данной работы заключается в следующем:

- Суперкомпилятор *HOSC* описан в общем виде - в виде отношения трансформации. Показана корректность трансформации. В работах [17, 4, 15] описаны конкретные детерминированные реализации суперкомпиляции.
- *HOSC*, в отличие от суперкомпиляторов [17, 4] осуществляет свертку любых узлов – показана корректность такой свертки. Благодаря сворачиванию любых узлов увеличивается способность суперкомпилятора к нормализации выражений, что полезно для применения суперкомпиляции как средства анализа программ.
- В [7] указано, что преобразование статического аргумента (генерация *letrec*-выражений со свободными переменными, [25]) может быть

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

letrec f = λxs → case xs of {
    Nil → ys;
    Cons x1 xs1 → Cons x1 (f xs1);
}
in f x
```

Рис. 27: $\mathcal{H}[\![app\ x\ ys]\!]$

иногда полезно для оптимизации программ, но не показана корректность такого преобразования. Мы показали, что генерация *letrec*-выражений со свободными переменными играет существенную роль при использовании суперкомпиляции в качестве инструмента анализа и показали корректность этого для отношения трансформации $HOSC$.

- В работах [17, 5] предполагается типизация по Хиндли-Милнеру (в стиле Карри), но не рассматривается вопрос о корректности суперкомпиляции в смысле типизации. Мы показали, что суперкомпиляция может расширять область определения выражения, выводимые типы в остаточном выражении могут быть более общими, нежели в исходном. Через явное приписывание типов в остаточной программе можно сузить область определения до исходной.
- Мы показали корректность отношения трансформации $HOSC$ с помощью теории операционных улучшений, что позволило показать, что трансформация $HOSC_0$ дает на выходе более эффективную программу. В [15] доказательство корректности проведено с помощью бисимуляции, что не позволяет сказать как относятся исходная и остаточная программа в смысле производительности. Мы работали с полной (sound) типизацией по Хиндли-Милнеру. В [15] рассматривается потенциально неполный (unsound) вариант системы F.

Список литературы

[1] O. Danvy. An extensional characterization of lambda-lifting and lambda-dropping. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *LNCS*, pages 241–250, London, UK, 1999. Springer-Verlag.

[2] O. Danvy and U. P. Schultz. Lambda-dropping: transforming recursive equations into programs with block structure. *Theor. Comput. Sci.*, 248(1-2):243–287, 2000.

[3] A. D. Gordon. Bisimilarity as a theory of functional programming. Minicourse. Technical Report NS-95-3, BRICS, University of Cambridge Computer Laboratory, 1995.

[4] P. Jonsson and J. Nordlander. Positive Supercompilation for a higher order call-by-value language. *ACM SIGPLAN Notices*, 44(1):277–288, 2009.

[5] P. A. Jonsson. Positive supercompilation for a higher-order call-by-value language. Master's thesis, Luleå University of Technology, 2008.

[6] P. A. Jonsson and J. Nordlander. Positive supercompilation for a higher order call-by-value language. extended proofs. Technical Report 17, Luleå University of Technology, 2008.

[7] P. A. Jonsson and J. Nordlander. Supercompiling overloaded functions. submitted to ICFP 2009, 2009.

[8] A. V. Klimov. A program specialization relation based on supercompilation and its properties. In *First International Workshop on Metacomputation in Russia*, 2008.

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

[10] I. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination. Preprint 21, 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] D. Krustev. A simple supercompiler formally verified in Coq. In *Second International Workshop on Metacomputation in Russia*, 2010.

[14] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 154–165. Springer, 1992.

[15] G. E. Mendel-Gleason and G. W. Hamilton. Equivalence in supercompilation and normalisation by evaluation. In *Second International Workshop on Metacomputation in Russia*, 2010.

[16] N. Mitchell. *Transformation and Analysis of Functional Programs*. PhD thesis, University of York, 2008.

[17] 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.

[18] B. Monsuez. Polymorphic typing for call-by-name semantics. In *Proceedings of the International Conference on Formal Methods in Programming and Their Applications*, pages 156–169, London, UK, 1993. Springer-Verlag.

[19] A. P. Nemytykh. The supercompiler SCP4: General structure. In *PSI 2003*, volume 2890 of *LNCS*, pages 162–170. Springer, 2003.

[20] S. L. Peyton Jones. An introduction to fully-lazy supercombinators. In *Combinators and Functional Programming Languages*, volume 242 of *LNCS*, pages 175–206. Springer, 1986.

[21] S. A. Romanenko. Arity raiser and its use in program specialization. In *ESOP '90*, volume 432 of *LNCS*, pages 341–360. Springer, 1990.

[22] D. Sands. Operational theories of improvement in functional languages. In *Proceedings of the Fourth Glasgow Workshop on Functional Programming*, 1991.

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

[24] D. Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. Program. Lang. Syst.*, 18(2):175–234, 1996.

[25] A. Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, Glasgow University, Department of Computing Science, 1995.

[26] U. P. Schultz. Implicit and explicit aspects of scope and block structure. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, 1997.

[27] M. H. Sørensen. Turchin's supercompiler revisited: an operational theory of positive information propagation. Master's thesis, Københavns Universitet, Datalogisk Institut, 1996.

[28] V. Turchin. Equivalent transformations of recursive functions defined in Refal. In *Proceedings of the symposium on Theory of Languages and Methods of Constructing of Programming Systems*, pages 31–42, 1972.

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

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