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

Илья Ключников

Суперкомпилятор HOSC 1.0: внутреннее устройство

Москва
2009

Ilya G. Klyuchnikov. Supercompiler HOSC 1.0: under the hood

The paper describes the internal structure of HOSC, an experimental supercompiler dealing with programs written in a higher-order functional language. A detailed and formal account is given of the concepts and algorithms the supercompiler is based upon.

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

Илья Г. Ключников. Суперкомпилятор HOSC 1.0: внутренняя структура

В работе описана внутренняя структура экспериментального суперкомпилятора HOSC. Дано полное описание всех существенных понятий и алгоритмов суперкомпилятора, работающего с функциональным языком высшего порядка.

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

Содержание

1 Введение

Суперкомпиляция – метод преобразования программ, предложенный В.Ф. Турчиным в 70-х годах прошлого века [19, 20].

В работе рассматривается специализатор HOSC – экспериментальный суперкомпилятор для простого функционального языка с функциями высших порядков.

Несмотря на то, что алгоритмы суперкомпиляции программ на языках с функциями высших порядков уже описывались ранее [6, 8, 9, 10, 13, 12], эти описания не являются полными в том смысле, что некоторые существенные детали в них опущены или описаны неформально.

Целью данной работы является *полное* и *формальное* описание алгоритмов суперкомпилятора HOSC. Исходный код суперкомпилятора HOSC публично доступен в сети Интернет по адресу `http://hosc.googlecode.com`.

2 Обозначения

Для краткости и удобочитаемости мы используем в формулах надчеркивания для обозначения повторяющихся конструкций: $\overline{A_i}$ означает $A_0 A_1 \ldots A_n$, если A_0 определено, или $A_1 \ldots A_n$ в противном случае (n определяется контекстом). Таким образом надчеркнутая конструкция должна содержать части с нижним индексом и похожа на генератор списков (list comprehension). Дважды надчеркнутый текст разворачивается “снаружи внутрь” по надчеркиваниям и слева направо по двойным индексам. Если надчеркнутый текст встречается внутри скобок, обозначающих множество или список ({}

$tDef$ ::= data $tCon = \overline{dCon_i}$;	type definition
$tCon$ ::= $tn\ \overline{tv_i}$	type constructor
$dCon$ ::= $c\ \overline{type_i}$	data constructor
$type$ ::= $tv \mid tCon \mid type \to type \mid (type)$	type expression
$prog$ ::= $\overline{tDef_i}\ e$ **where** $\overline{f_i = e_i}$;	program
e ::= v	variable
$\mid c\ \overline{e_i}$	constructor
$\mid f$	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 function
$\mid (e)$	parenthesized expression
p ::= $c\ \overline{v_i}$	pattern

Рис. 1: Грамматика HLL

или [] соответственно), мы считаем, что при разворачивании разделители элементов (запятые) вставляются автоматически. Например:

$$
\begin{array}{l}
let\ \overline{v_i = e_i};\ in\ e \Rightarrow let\ v_1 = e_1;\ v_2 = e_2;\ \ldots;\ v_n = e_n;\ in\ e \\
[\gamma,\ \overline{\gamma_i}] \Rightarrow [\gamma,\ \gamma_1,\ \gamma_2,\ \ldots,\ \gamma_n] \\
case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} \Rightarrow \\
case\ e_0\ of\ \{c_1\ \overline{v_{1k}} \to e_1;\ \ldots;\ c_n\ \overline{v_{nk}} \to e_n;\} \Rightarrow \\
case\ e_0\ of\ \{c_1\ v_{11}\ \ldots\ v_{1m_1} \to e_1;\ \ldots;\ c_n\ v_{n1}\ \ldots\ v_{nm_n} \to e_n;\}
\end{array}
$$

3 Входной язык

Входным языком суперкомпилятора HOSC является HLL – простой функциональный язык с функциями высших порядков. Семантика вычислений на языке HLL – ленивая. Язык HLL типизирован по Хиндли-Милнеру [2].

Программа на языке HLL состоит из определений типов данных, целевого выражения и определений глобальных функций.

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 $\overline{dCon_i} \Rightarrow dCon_1 \mid \ldots \mid dCon_n$).

Грамматика HLL показана на Рис. 1. Выражение – это переменная, конструктор, λ-абстракция, аппликация, case-выражение, определение ло-

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

(compose (map f)(map g)) xs where

compose = λf g x → f (g x);

map = λf xs →
    case xs of {
        Nil → Nil;
        Cons x1 xs1 → Cons (f x1) (map f xs1);
    };
```

Рис. 2: Пример программы на языке HLL

кальной функции или выражение в скобках. Определение функции связывает *глобальную* переменную с λ-абстракцией.

Целевое выражение e в программе *prog* на языке HLL может содержать свободные переменные. В определениях глобальных функций все переменные должны быть связанными.

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 (корректно типизированным) выражением. Во втором случае мы требуем, чтобы список аргументов $\overline{e_i}$ был непустым, а выражение e_0 не было аппликацией.

Пример программы на языке HLL приведен на Рис. 2.

3.1 Основные определения и соглашения

Следующие определения будут использоваться на протяжении всей статьи.

3.1.1 Связанные, глобальные и свободные переменные

Переменная является *связанной* переменной, если либо (1) она является аргументом объемлющей λ-абстракции или (2) определяется в образце объемлющей ветви `case`-выражения. В противном случае переменная считается *свободной*. Чтобы избежать возможного конфликта имен, мы требуем, чтобы имена переменных были уникальными в следующем смысле: для любых v и e *все* вхождения переменной v в выражение e либо свободные, либо связанные.

3.1.2 Равенство выражений

Два выражения e_1 и e_2 считаются равными, если они различаются только именами связанных переменных и есть взаимно однозначное соответствие между связанными переменными e_1 и e_2. Равенство выражений e_1 и e_2 записывается как $e_1 \equiv e_2$

3.1.3 Свободные переменные

Множество свободных переменных $fv[\![e]\!]$ выражения e определяется индуктивно следующим образом.

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

3.1.4 Связанные переменные

Множество связанных переменных $bv[\![e]\!]$ выражения e определяется индуктивно следующим образом.

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

3.1.5 Подстановка

Подстановкой будем называть конечный список пар вида

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

каждая пара в котором связывает переменную v_i с ее значением e_i.

Область определения подстановки обозначается $domain(\theta)$ и определяется следующим образом:

$$domain(\{v_1 := e_1, v_2 := e_2, \ldots, v_n := e_n\}) = \{v_1,\ v_2,\ \ldots,\ v_n\}$$

3.1.6 Применение подстановки

Результат применения подстановки θ к выражению e записывается в постфиксной форме $e\ \theta$. Применение подстановки θ к выражению e состоит в следующем

- Все свободные переменные выражения e, входящие в $domain(\theta)$, заменяются на их значения из θ
- Все остальные части выражения e остаются неизменными

Чтобы избежать проблемы конфликта имен переменных, мы требуем, чтобы область определения θ не содержала связанных переменных из e:

$$domain(\theta) \cap bv[\![e]\!] = \emptyset$$

Формально применение подстановки определяется так:

$$\begin{array}{lcll}
v\theta & = & e & \text{if } v := e \in \theta \\
 & = & v & \text{otherwise} \\
(c\ \overline{e_i})\theta & = & c\ \overline{(e_i\theta)} & \\
(\lambda v \to e)\theta & = & \lambda v \to (e\theta) & \\
(e_1\ e_2)\theta & = & (e_1\theta)\ (e_2\theta) & \\
(case\ e_0\ of\ \{\overline{p_i\ \to e_i;}\})\theta & = & case\ (e_0\theta)\ of\ \{\overline{p_i\ \to (e_i\theta);}\} & \\
(letrec\ f = e_1\ in\ e_2)\theta & = & letrec\ f = (e_1\theta)\ in\ (e_2\theta) &
\end{array}$$

3.1.7 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 $\theta = e_1 \circledcirc e_2$.

3.1.8 Переименование выражения

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

3.1.9 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 $e_1 \frown e_2$.

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

Рис. 3: Наблюдаемые выражения, контексты, редексы

3.1.10 Наиболее тесное обобщение

Обобщение $(e_g, \theta_1, \theta_2)$ называется *наиболее тесным обобщением* выражений e_1 и e_2, если для любого обобщения $(e'_g, \theta'_1, \theta'_2) \in e_1 \frown e_2$ верно, что $e'_g \lessdot e_g$, т.е. e_g является частным случаем e'_g. Мы предполагаем, что определена операция $\sqcap$ такая , что $e_1 \sqcap e_2 = (e_g, \theta_1, \theta_2)$

3.2 λ-лифтинг

Отметим, что конструкция `where` является всего лишь синтаксическим сахаром, так как глобальные определения функций всегда можно трансформировать в `letrec`-и и переместить в целевое выражение. Таким образом, любую программу на языке HLL всегда можно заменить на эквивалентную, состоящую из определений типов данных и одного *самодостаточного* целевого выражения. В некоторых случаях удобнее иметь дело именно с такими программами. Например, задача распознавания эквивалентности программ сводится к распознаванию эквивалентности выражений.

Однако, в процессе суперкомпиляции удобнее работать не с `letrec`-ами, а с глобальными функциями. Поэтому на первом этапе преобразований мы изгоняем все `letrec`-и из программы широко известным методом λ-лифтинга [7].

В дальнейшем мы будем считать, что исходные программы не содержат `letrec`-выражений. (А остаточные программы – наоборот, не будут содержать глобальных определений по построению.)

3.3 Наблюдаемые выражения, контексты и редексы

Синтаксис наблюдаемых выражений, редексов и контекстов на Рис. 3.

К *наблюдаемым выражениям* относятся:

- локальная переменная,
- аппликация локальной переменной,
- конструктор,
- λ-абстракция.

$$
\begin{array}{lcl}
\mathcal{E}[\![c\ \overline{e_i}]\!] & \Rightarrow & c\ \overline{e_i} \\
\mathcal{E}[\![\lambda v_0 \to e_0]\!] & \Rightarrow & \lambda v_0 \to e_0 \\
\mathcal{E}[\![con\langle f_g \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle unfold(f_g) \rangle]\!] \\
\mathcal{E}[\![con\langle (\lambda v \to e_0)\ e_1 \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle e_0\{v\ :=\ e_1\} \rangle]\!] \\
\mathcal{E}[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\} \rangle]\!] & \Rightarrow & \mathcal{E}[\![con\langle e_j\{\overline{v_{jk}\ :=\ e'_k}\} \rangle]\!]
\end{array}
$$

Рис. 4: Операционная семантика языка HLL (вызов по имени)

К *редексам* относятся:

- глобальная переменная
- аппликация λ-абстракции
- case-выражение, где селектором является наблюдаемое выражение

Контекст – выражение с дырой. Контекстом является:

- дыра (пустой контекст) $\langle\rangle$
- аппликация контекста
- case-выражение, где селектором является контекст

Если con – контекст, то $con\langle e \rangle$ обозначает результат замены дыры внутри con на e. По сути дела, дыра $\langle\rangle$ используется как метапеременная.

Можно показать, что любое выражение e на языке HLL является либо наблюдаемым выражением, либо представимо в виде контекста con с помещенным в дыру редексом r, то есть $e = con\langle r \rangle$. Этот факт известен как *единственность декомпозиции*.

3.4 Операционная семантика языка HLL

Операционная семантика языка определяется как редукция с нормальным порядком вычислений к слабой головной нормальной форме [1] (Рис. 4). Обратите внимание, что операционная семантика определяется *для выражений без свободных переменных*.

4 Гомеоморфное вложение

Традиционно гомеоморфное вложение описывается для выражений без связанных переменных [11, 4, 15, 16], – все переменные считаются свободными и вкладывающимися друг в друга. Однако, в выражениях языка HLL

могут встречаться связанные переменные, поэтому для суперкомпилятора HOSC вложение переменных определяется более утонченно:

- любые свободные переменные вкладываются
- вкладываются *соответствующие* связанные переменные
- глобальная переменная вкладывается *только* сама в себя

Для запоминания соответствия связанных переменных мы будем использовать таблицу соответствия ρ:

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

В случае вложения двух λ-абстракций через сцепление, в ρ помещается пара переменных-аргументов этих абстракций. В случае вложения двух case-выражений через сцепление, в ρ помещаются пары соответствующих друг другу образцовых переменных.

При проверке вложено ли через погружение выражения e в тело λ-абстракции $\lambda v \to e_1$, связанной переменной λ-абстракции v не соответствует никакая из переменных e. Поэтому помещаем в ρ пару $(\bullet, v)$, где $\bullet$ – "пустышка", заменяющая отсутствующую переменную.

При проверке вложено ли через погружение выражение e в ветвь case-выражения, мы поступаем аналогичным образом – запоминаем, что образцовым переменным не соответствуют никакие переменные из e.

4.1 Расширенное гомеоморфное вложение

Пусть ρ – таблица соответствия связанных переменных. Тогда отношение *расширенного гомеоморфного вложения* $\trianglelefteq|_\rho$ для выражений e' и e'' (при таблице соответствий ρ) определяется по индукции следующим образом:

$$e' \trianglelefteq e'' \,|_\rho \qquad \text{if } e' \trianglelefteq_v e'' \,|_\rho \text{ or } e' \trianglelefteq_d e'' \,|_\rho \text{ or } e' \trianglelefteq_c e'' \,|_\rho$$

Таким образом, мы различаем следующие типы вложений:

- $e' \trianglelefteq_v e'' \,|_\rho$ – вложение переменных
- $e' \trianglelefteq_d e'' \,|_\rho$ – погружение
- $e' \trianglelefteq_c e'' \,|_\rho$ – сцепление

Мы будем также пользоваться следующими сокращениями

$$\begin{array}{lcl} e' \trianglelefteq e'' & \stackrel{\text{def}}{=} & e' \trianglelefteq e'' \,|_{\{\}} \\ e' \trianglelefteq_v e'' & \stackrel{\text{def}}{=} & e' \trianglelefteq_v e'' \,|_{\{\}} \\ e' \trianglelefteq_d e'' & \stackrel{\text{def}}{=} & e' \trianglelefteq_d e'' \,|_{\{\}} \\ e' \trianglelefteq_c e'' & \stackrel{\text{def}}{=} & e' \trianglelefteq_c e'' \,|_{\{\}} \end{array}$$

4.2 Вложение переменных

$$
\begin{array}{rl}
f \trianglelefteq_v f \mid_\rho & \\
v_1 \trianglelefteq_v v_2 \mid_\rho & \text{if } (v_1,\ v_2) \in \rho \\
v_1 \trianglelefteq_v v_2 \mid_\rho & \text{if } v_1 \notin \mathit{domain}(\rho) \text{ and } v_2 \notin \mathit{range}(\rho)
\end{array}
$$

4.3 Вложение через погружение

Для вложения выражения e через погружение в другое выражение при таблице соответствий ρ мы требуем:

$$\forall v \in fv(e) : v \notin \mathit{domain}(\rho)$$

Обоснование разумности этого требования будет дано в следующем разделе. Вложение через погружение определяется следующим образом:

$$
\begin{array}{rl}
e \trianglelefteq_d c\ \overline{e_i} \mid_\rho & \text{if } e \trianglelefteq e_i \mid_\rho \text{ for some } i \\
e \trianglelefteq_d \lambda v_0 \to e_0 \mid_\rho & \text{if } e \trianglelefteq e_0 \mid_{\rho \cup \{(\bullet, v_0)\}} \\
e \trianglelefteq_d \overline{e_i} \mid_\rho & \text{if } e \trianglelefteq e_i \mid_\rho \text{ for some } i \\
e \trianglelefteq_d \mathit{case}\ e_0\ \mathit{of}\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} \mid_\rho & \text{if } e \trianglelefteq e_0 \mid_\rho \\
e \trianglelefteq_d \mathit{case}\ e_0\ \mathit{of}\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\} \mid_\rho & \text{if } e \trianglelefteq e_i \mid_{\rho \cup \{\overline{(\bullet, v_{ik})}\}} \text{ for some } i
\end{array}
$$

4.4 Вложение через сцепление

$$
\begin{array}{ll}
c\ \overline{e'_i} \trianglelefteq_c c\ \overline{e''_i} \mid_\rho & \text{if } \forall i : e'_i \trianglelefteq e''_i \mid_\rho \\
\lambda v_1 \to e_1 \trianglelefteq_c \lambda v_2 \to e_2 \mid_\rho & \text{if } e_1 \trianglelefteq e_1 \mid_{\rho \cup \{(v_1, v_2)\}} \\
e'\ \overline{e'_i} \trianglelefteq_c e''\ \overline{e''_i} \mid_\rho & \text{if } e' \trianglelefteq e'' \mid_\rho \text{ and } \forall i : e'_i \trianglelefteq e''_i \mid_\rho \\
\mathit{case}\ e'\ \mathit{of}\ \{\overline{c_i\ \overline{v'_{ik}} \to e'_i;}\} \trianglelefteq_c \mathit{case}\ e''\ \mathit{of}\ \{\overline{c_i\ \overline{v''_{ik}} \to e''_i;}\} \mid_\rho & \\
\qquad \text{if } e' \trianglelefteq e'' \mid_\rho \text{ and } \forall i : e'_i \trianglelefteq e''_i \mid_{\rho \cup \{\overline{(v'_{ik}, v''_{ik})}\}}
\end{array}
$$

Важно, что в третьем правиле (сцепление аппликаций) e' и e'' сами не являются аппликациями, – это гарантирует наличие нетривиального обобщения для любых сцепленных выражений.

4.5 Примеры

Приведем примеры вкладывающихся и не вкладывающихся выражений (в примерах используются конструкторы и глобальные переменные, определенные в программах на Рис. 2 и Рис. 9):

- $x \trianglelefteq y$
- $\mathit{map} \not\trianglelefteq \mathit{compose}$

- $map\ f \not\trianglelefteq compose\ map\ f$
- $map\ f \trianglelefteq_c map\ (compose\ f\ g)$
- $\lambda x \to Nil \trianglelefteq_c \lambda x \to Cons\ a\ Nil$
- $\lambda x \to Nil \not\trianglelefteq \lambda x \to Cons\ x\ Nil$
- $case\ x\ of\ \{Z \to x;\ S\ b \to f\ b;\} \trianglelefteq_c case\ x\ of\ \{Z \to x;\ S\ b \to g\ b;\}$
- $case\ x\ of\ \{Z \to x;\ S\ b \to f\ b;\} \not\trianglelefteq case\ x\ of\ \{Z \to x;\ S\ b \to (f\ g)\ b;\}$
- $case\ x\ of\ \{Z \to x;\ S\ b \to f\ b;\} \not\trianglelefteq case\ x\ of\ \{Z \to x;\ S\ b \to f\ (g\ b);\}$

5 Обобщение

Специализатор HOSC обобщает два выражения e_1 и e_2 только тогда, когда e_1 вложено через сцепление в e_2. Поэтому мы описываем алгоритм нахождения тесного обобщения выражений e_1 и e_2 только для случая, когда $e_1 \trianglelefteq_c e_2$.

5.1 Тесное обобщение сцепленных выражений

Тесного обобщение выражений e_1 и e_2, таких, что $e_1 \trianglelefteq_c e_2$, вычисляется путем последовательного применений следующих правил переписывания (правила общего функтора и правила общего подвыражения) к начальному тривиальному обобщению $(v, \{v := e_1\}, \{v := e_2\})$. Требование раздела 4.3 исходит из того, что θ_1 и θ_2 должны быть корректными подстановками – то есть подстановками над свободными переменными. Это требование гарантирует, что подстановки будут корректными, если $e_1 \trianglelefteq_c e_2$.

5.2 Правило общего функтора

5.2.1 Переменная

$$\begin{pmatrix} e \\ \{v_1 := v_2\} \cup \theta_1 \\ \{v_1 := v_2\} \cup \theta_2 \end{pmatrix} \Rightarrow \begin{pmatrix} e\{v_1 := v_2\} \\ \theta_1 \\ \theta_2 \end{pmatrix}$$

5.2.2 Конструктор

$$\begin{pmatrix} e \\ \{v := c\ \overline{e'_1}\} \cup \theta_1 \\ \{v := c\ \overline{e''_1}\} \cup \theta_2 \end{pmatrix} \Rightarrow \begin{pmatrix} e\{v := c\ v_1\ \ldots\ v_n\} \\ \{\overline{v_i := e'_1}\} \cup \theta_1 \\ \{\overline{v_i := e''_1}\} \cup \theta_2 \end{pmatrix}$$

5.2.3 λ-абстракция

$$\left(\begin{array}{c} e \\ \{v := \lambda v' \to e'\} \cup \theta_1 \\ \{v := \lambda v'' \to e''\} \cup \theta_2 \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v := \lambda v_0 \to v_1\} \\ \{v_1 := e'\{v' := v_0\}\} \cup \theta_1 \\ \{v_1 := e''\{v'' := v_0\}\} \cup \theta_2 \end{array}\right)$$

5.2.4 Аппликация

$$\left(\begin{array}{c} e \\ \{v := e'_0\ \overline{e'_i}\} \cup \theta_1 \\ \{v := e''_0\ \overline{e''_i}\} \cup \theta_2 \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v := v_0\ \overline{v_i}\} \\ \{v_0 := e'_0,\ \overline{v_i := e'_i}\} \cup \theta_1 \\ \{v_0 := e''_0,\ \overline{v_i := e''_i}\} \cup \theta_2 \end{array}\right)$$

5.2.5 Case-выражение

$$\left(\begin{array}{c} e \\ \{v := case\ e'_0\ of\ \overline{\{c_i\ \overline{v'_{ik}} \to e'_i;\}}\} \cup \theta_1 \\ \{v := case\ e''_0\ of\ \overline{\{c_i\ \overline{v''_{ik}} \to e''_i;\}}\} \cup \theta_2 \end{array}\right)$$
$$\Downarrow$$
$$\left(\begin{array}{c} e\{v := case\ v_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to v_i;}\}\} \\ \{v_0 := e'_0,\ \overline{v_i := e'_i\{\overline{v'_{ik} := v_{ik}}\}}\} \cup \theta_1 \\ \{v_0 := e''_0,\ \overline{v_i := e''_i\{\overline{v''_{ik} := v_{ik}}\}}\} \cup \theta_2 \end{array}\right)$$

5.3 Правило общего подвыражения

$$\left(\begin{array}{c} e \\ \{v_1 := e',\ v_2 := e'\} \cup \theta_1 \\ \{v_1 := e'',\ v_2 := e''\} \cup \theta_2 \end{array}\right) \Rightarrow \left(\begin{array}{c} e\{v_1 := v_2\} \\ \{v_2 := e'\} \cup \theta_1 \\ \{v_2 := e''\} \cup \theta_2 \end{array}\right)$$

5.4 Let-выражение

Let-выражение используется для представления для представления результата обобщения выражений. Синтаксис let-выражения имеет вид:

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

где v_i может входить в e_g, значением v_i является вычисление e_i. Поскольку HLL - ленивый язык, то let-выражение по смыслу эквивалентно $e_g\{\overline{v_i := e_i}\}$

Let-выражения появляются при построении частичного дерева процессов и не допускаются во входной программе.

5.5 Примеры

Приведем примеры обобщения вложенных через сцепление выражений из раздела 4.5:

- $map\ f \sqcap map\ (compose\ f\ g) =$
  $(map\ h, \{h := f\}, \{h := (compose\ f\ g)\})$
- $\lambda x \to Nil \sqcap \lambda x \to Cons\ a\ Nil =$
  $(\lambda x \to y, \{y := Nil\}, \{y := Cons\ a\ Nil\})$
- $case\ x\ of\ \{Z \to x;\ S\ b \to f\ b;\} \sqcap case\ x\ of\ \{Z \to x;\ S\ b \to (f\ g)\ b;\} =$
  $(case\ x\ of\ \{Z \to x;\ S\ b \to h\ b;\}, \{h := f\}, \{h := f\ g\})$

6 Метавычисления

Дерево процессов представляет собой ориентированное (возможно – бесконечное) дерево, в узлах которого находятся HLL-выражения. Деревья процессов строятся при *метавычислениях* [3, 4, 22, 23], – символическом вычислении выражений, которые могут содержать (в отличие от случая обычных вычислений) свободные переменные.

Мы будем обозначать выражение, помещенное в узел n, как $n.expr$.

Построение дерева процессов для программы начинается с того, что создается дерево, состоящее из одного узла, в который помещается целевое выражение. Затем к листьям дерева постепенно подвешиваются новые потомки в соответствии с правилами *прогонки*, описанными ниже.

6.1 Шаг прогонки

Оператор $\mathcal{D}$ (Рис 5) принимает в качестве аргумента выражение, находящееся в листе, и возвращает ноль или более выражений $e_1, \ldots, e_k$. Затем суперкомпилятор подвешивает к листу k дочерних узлов и помещает в них выражения $e_1, \ldots, e_k$.

Последнее правило относится к let-выражениям, которые не присутствуют в исходных программах, но могут появляться в результате обобщения.

7 Частичное дерево процессов

В результате метавычислений в общем случае строится бесконечное дерево процессов. Задача суперкомпиляции – превратить его в конечный (возможно, циклический) граф, который называется *частичным деревом процессов* и отличается от обычного дерева процессов следующими свойствами:

$$\mathcal{D}[\![v\ \overline{e_i}]\!] \Rightarrow [v,\ \overline{e_i}] \qquad (D_1)$$

$$\mathcal{D}[\![c\ \overline{e_i}]\!] \Rightarrow [\overline{e_i}] \qquad (D_2)$$

$$\mathcal{D}[\![\lambda v_0 \to e_0]\!] \Rightarrow [e_0] \qquad (D_3)$$

$$\mathcal{D}[\![con\langle f_0\rangle]\!] \Rightarrow [con\langle unfold(f_0)\rangle] \qquad (D_4)$$

$$\mathcal{D}[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] \Rightarrow [con\langle e_0\{v_0 := e_1\}\rangle] \qquad (D_5)$$

$$\mathcal{D}[\![con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}\rangle]\!] \Rightarrow \left[con\langle e_j\{\overline{v_{jk} := e'_k}\}\rangle\right] \qquad (D_6)$$

$$\mathcal{D}[\![con\langle case\ v\ \overline{e'_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!] \Rightarrow \left[v\ \overline{e'_j},\ \overline{con\langle e_i\{v\ \overline{e'_j} := p_i\}\rangle}\right] \qquad (D_7)$$

$$\mathcal{D}[\![let\ \overline{v_i = e_i;}\ in\ e]\!] \Rightarrow [e,\ \overline{e_i}] \qquad (D_8)$$

Рис. 5: Шаг прогонки

- В узлах частичного дерева процессов могут быть let-выражения.
- В частичном дереве процессов могут быть *циклы*: пусть узел 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.

Таким образом, частичное дерево процессов есть ориентированное дерево (дуги которого обозначаются $\to$), дополненное “повторными"дугами (обозначаемыми $\rightrightarrows$), которые превращают его в ориентированный граф.

Целью суперкомпиляции является построение частичного дерева процессов, имеющего размеры “в пределах разумного”.

На Рис. 6 описаны операции над частичным деревом процессов, которые используются суперкомпилятором HOSC при построении частичного дерева процессов и при генерации остаточной программы.

Узел β считается *обработанным* в следующих случаях:

- Узел уже “зациклен”: из узла выходит двойная дуга $\beta \rightrightarrows \alpha$
- $\beta.expr$ является нуль-арным конструктором
- $\beta.expr$ является локальной переменной

Узел β считается *нетривиальным*, если $\beta.expr$ представимо в виде:

$$con\langle f\rangle \qquad \text{or} \qquad con\langle case\ v\ \overline{e_j}\ or\ \{\overline{p_i \to e_i;}\}\rangle$$

Во всех остальных случаях узел β считается *тривиальным*.

$processed(node)$	Возвращает $true$ или $false$ в зависимости от того, обработан данный узел или нет.
$trivial(node)$	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.
$ancestor(t,\ \beta,\ \simeq)$	Возвращает узел α - ближайшего предка β такого, что $\alpha.expr \simeq \beta.expr$, или $\bullet$, если β не имеет такого предка.
$ancestor(t,\ \beta,\ \lessdot)$	Возвращает узел α - ближайшего предка β такого, что $\alpha.expr \lessdot \beta.expr$, или $\bullet$, если β не имеет такого предка.
$ancestor(t,\ \beta,\ \trianglelefteq_c)$	Возвращает узел α - ближайшего предка β такого, что $\alpha.expr \trianglelefteq_c \beta.expr$, или $\bullet$, если β не имеет такого предка.
$fold(t,\ \alpha,\ \beta)$	"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
$unprocessedLeaves(t)$	Возвращает список всех необработанных листьев дерева t, или $\bullet$, если все листья обработаны.

Рис. 6: Операции над частичным деревом процессов

8 Построение частичного дерева процессов

Настало время описать центральную часть суперкомпилятора HOSC – алгоритм построения частичного дерева процессов (Рис. 7).

Пусть дана программа, построение частичного дерева процессов начинается с того, что в корень дерева помещается исходное (целевое) выражение. Затем, пока есть хотя бы один необработанный лист β:

1. Если β является тривиальными узлом, "метавычисляем" $\beta.expr$ – делаем один шаг прогонки.

2. Если у β есть предок α такой, что $\alpha.expr \simeq \beta.expr$, то делаем узел α базовым узлом β – проводим специальную дугу из β в α: $\beta \rightrightarrows \alpha$

while $unprocessedLeaves(t) \neq \bullet$ **do**
  $\beta = unprocessedLeaves(t).head$
  if $trivial(\beta)$ **then**
    $t = drive(t, \beta)$
  else if $ancestor(t,\ \beta,\ \simeq) \neq \bullet$ **then**
    $\alpha = ancestor(t,\ \beta,\ \simeq)$
    $t = fold(t,\ \alpha,\ \beta)$
  else if $ancestor(t,\ \beta,\ \lessdot) \neq \bullet$ **then**
    $\alpha = ancestor(t,\ \beta,\ \lessdot)$
    $t = abstract(t, \beta, \alpha)$
  else if $ancestor(t,\ \beta,\ \trianglelefteq_c) \neq \bullet$ **then**
    $\alpha = ancestor(t,\ \beta,\ \trianglelefteq_c)$
    $t = abstract(t, \alpha, \beta)$
  else
    $t = drive(t, \beta)$
  end
end

Рис. 7: Алгоритм построения частичного дерева процессов

3. Если у β есть предок α такой, что $\alpha.expr \lessdot \beta.expr$, то обобщаем $\beta.expr$
4. Если у β есть предок α такой, что $\alpha.expr \trianglelefteq_c \beta.expr$, то обобщаем $\alpha.expr$
5. В противном случае "метавычисляем" $\beta.expr$ – делаем шаг прогонки.

Наиболее существенный момент алгоритма – четвертый пункт. Если из алгоритма этот пункт исключить, то в большинстве случаев работа суперкомпилятора никогда не завершится.

Однако, отношение гомеоморфного вложения является своего рода свистком, который сигнализирует о том, что два выражения "опасно похожи" друг на друга, но обратную дугу все же нельзя провести. Поэтому суперкомпилятор обобщает выражение в функциональном (базовом узле), предотвращая таким образом бесконечное разрастание дерева процессов.

9 Построение остаточной программы

В данном разделе описывается алгоритм преобразования частичного дерева процессов t в программу на языке HLL.

Алгоритм (представленный на Рис. 8) определяется через два взаимно рекурсивных оператора (функции): $\mathcal{C}$ и $\mathcal{C}'$. Остаточная программа извле-

$$\mathcal{C}\ [\![\alpha]\!]_{t,\Sigma} = \mathcal{C}'[\![\alpha.expr]\!]_{t,\alpha,\Sigma} \qquad (C_0)$$

$$\mathcal{C}'[\![let\ \overline{v_i = e_i;}\ in\ e]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]\{\overline{v_i = \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma}}\} \qquad (C_1)$$

$$\mathcal{C}'[\![v\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow v\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} \qquad (C_2)$$

$$\mathcal{C}'[\![c\ \overline{e_i}]\!]_{t,\alpha,\Sigma} \Rightarrow c\ \overline{\mathcal{C}[\![\gamma_i]\!]_{t,\Sigma}} \qquad (C_3)$$

$$\mathcal{C}'[\![\lambda v_0 \to e_0]\!]_{t,\alpha,\Sigma} \Rightarrow \lambda v_0 \to \mathcal{C}[\![\gamma']\!]_{t,\Sigma} \qquad (C_4)$$

$$\mathcal{C}'[\![con\langle(\lambda v_0 \to e_0)\ e_1\rangle]\!] \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} \qquad (C_5)$$

$$\mathcal{C}'[\![con\langle case\ c\ \overline{e_j'}\ of\ \{\overline{p_i \to e_i;}\}\rangle]\!]_{t,\alpha,\Sigma} \Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} \qquad (C_6)$$

$$\mathcal{C}'[\![con\langle f_g\rangle]\!]_{t,\alpha,\Sigma}$$

$$\Rightarrow letrec\ f' = \lambda\overline{v_i} \to \quad \text{if } \lceil \alpha \between t \rceil \neq \bullet \qquad (C_7^1)$$
$$(\mathcal{C}[\![\gamma']\!]_{t,\Sigma'})\theta'$$
$$in\ f'\overline{v_i'}$$
where
$$[\overline{\beta_i}] = \lceil \alpha \between t \rceil,\ \theta_i = \alpha.expr \otimes \beta_i.expr,$$
$$\overline{v_i'} = domain(\bigcup \overline{\theta_i}),\ \theta' = \{\overline{v_i' := v_i}\},$$
$$\Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i}),\ f' \text{ and } \overline{v_i} \text{ are fresh}$$

$$\Rightarrow f'_{sig}\theta \quad \text{if } \lceil \alpha \Uparrow t \rceil \neq \bullet \qquad (C_7^2)$$
where
$$f'_{sig} = \Sigma(\lceil \alpha \Uparrow t \rceil),\ \theta = \lceil \alpha \Uparrow t \rceil.expr \otimes \alpha.expr$$

$$\Rightarrow \mathcal{C}[\![\gamma']\!]_{t,\Sigma} \quad \text{otherwise} \qquad (C_7^3)$$

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

$$\Rightarrow letrec\ f' = \lambda\overline{v_i} \to \quad \text{if } \lceil \alpha \between t \rceil \neq \bullet \qquad (C_8^1)$$
$$(case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'} of \{\overline{p_i \to \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma'};}\})\theta'$$
$$in\ f'\overline{v_i'}$$
where
$$[\overline{\beta_i}] = \lceil \alpha \between t \rceil,\ \theta_i = \alpha.expr \otimes \beta_i.expr,$$
$$\overline{v_i'} = domain(\bigcup \overline{\theta_i}),\ \theta' = \{\overline{v_i' := v_i}\},$$
$$\Sigma' = \Sigma \cup (\alpha,\ f'\ \overline{v_i}),\ f' \text{ and } \overline{v_i} \text{ are fresh}$$

$$\Rightarrow f'_{sig}\theta \quad \text{if } \lceil \alpha \Uparrow t \rceil \neq \bullet \qquad (C_8^2)$$
where
$$f'_{sig} = \Sigma(\lceil \alpha \Uparrow t \rceil),\ \theta = \lceil \alpha \Uparrow t \rceil.expr \otimes \alpha.expr$$

$$\Rightarrow case\ \mathcal{C}[\![\gamma']\!]_{t,\Sigma'} of \{\overline{p_i \to \mathcal{C}[\![\gamma_i']\!]_{t,\Sigma'};}\} \quad \text{otherwise} \qquad (C_8^3)$$

Чтобы уменьшить громоздкость правил, принято следующее соглашение. Если в правой части правила используется γ_i, считается, что $[\overline{\gamma_i}] = children(t,\ \alpha)$ и 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 или же первый дочерний узел требует “особого рассмотрения”.

Рис. 8: Правила генерации остаточной программы

кается из частичного дерева процессов t вычислением

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

Для построения остаточной программы мы обходим частичное дерево сверху вниз, начиная с корня. При обходе функционального узла, генерируется определение рекурсивной функции. Соответствие между функциональным узлом и сигнатурой новой функции заносится в Σ. Впоследствии Σ используется для генерации рекурсивного вызова.

Остановимся на наиболее существенных деталях этого алгоритма.

- Правило C_0 связывает $\mathcal{C}$ и $\mathcal{C}'$.
- Правила C_1, C_2 и C_3 определяют, как нужно собрать обработанные части *let*-выражения, вызова неизвестной функции и конструктора, чтобы получить соответствующее остаточное выражение.
- Правило C_4 описывает конструирование остаточной λ-абстракции из обработанного тела.
- Случаям $con\langle(\lambda v_0 \to e_0)\ e_1\rangle$ и $con\langle case\ c\ \overline{e_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$ соответствуют правила C_5 и C_6. Шаги редукции, полностью выполненные во время прогонки не попадают в остаточную программу.
- Правила C_7 и C_8 используются при обходе нетривиальных узлов:
  - При обходе базового узла генерируется letrec – рекурсивное определение функции (правила C_7^1 и C_8^1). Причем аргументами λ-абстракции становятся только те свободные переменные, которые “не сохранились” в выражениях в повторных узлах. Этот прием позволяет понижать арность функций. Соответствие между базовый узлом и сигнатурой заносится в Σ и передается вниз по дереву.
  - При обходе повторного узла генерируется рекурсивный выход функции, определяемой базовым узлом (правила C_7^2 и C_8^2).
  - Правила C_7^3 и C_8^3 используются в случае, когда нетривиальный узел не является ни базовым, ни повторным.

10 Обзор литературы

Первоначально суперкомпиляция была сформулирована для языка Рефал [18]. В настоящее время, наиболее разработанным суперкомпилятором для языка Рефал является SCP4 [14].

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

Первое *полное* и *формальное* описание алгоритмов, являющихся существенными для любого суперкомпилятора (прогонка, обобщение, генерация остаточной программы), встречается в магистерской работе Сёренсена [15] и в последующих совместных статьях Глюка и Сёренсена [4, 17], где описан суперкомпилятор простейшего функционального языка первого порядка (семантика вычислений – вызов по имени).

Полное описание алгоритмов для суперкомпилятора языка TSG (функциональный язык первого порядка, вызов по имени) дано Абрамовым [23].

Суперкомпиляторы, описанные Абрамовым, Глюком и Сёренсеном строят частичное дерево процессов, которое затем преобразуется в остаточную программу.

В последние несколько лет возрос интерес к суперкомпиляции функциональных языков с функциями высших порядков [6, 8, 9, 10, 13, 12].

Джонссон [8, 9] и Митчелл [13, 12] используют алгоритмы близкие по духу к дефорестации [21, 5],~- у них отсутствует отдельная фаза построения частичного дерева процессов и прогонка перемежается с генерацией остаточной программы.

Гамильтон [6] и Кабир [10] строят частичное дерево процессов.

В работах Гамильтона, Джонссона, Кабира и Митчела обстоятельно изложены прогонка и построение остаточной программы. К сожалению, используемые ими отношение гомеоморфного вложения и алгоритм нахождения тесного обобщения не описаны должным образом.

Отношение гомеоморфного вложения (используемого для обобщения) для языка первого порядка без case-выражений подробно описано в [4, 15, 16]. Case-выражения рассматриваются в [17], однако там отсутствует формальное описание вложения для case-выражений. Такая же ситуация и для языка высшего порядка с case-выражениями в [6, 8, 9, 10, 13, 12].

В уже упоминавшихся работах [4, 15, 16] описывается алгоритм нахождения тесного обобщения для языка первого порядка без case-выражений, то есть без связанных переменных. В работах [17, 8, 6, 13] рассматриваются связанные переменные, но алгоритм нахождения тесного обобщения не приведен полностью – лишь говорится, что он является модификацией алгоритмов из [4, 15, 16].

11 Заключение

В работе описана внутренняя структура экспериментального суперкомпилятора HOSC, работающего с функциональным языком высшего порядка.

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

Благодарности

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

Список литературы

[1] S. Abramsky. The lazy lambda calculus. *Research topics in functional programming*, pages 65–116, 1990.

[2] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM New York, NY, USA, 1982.

[3] R. Glück and A.V. Klimov. Occam's razor in metacompuation: the notion of a perfect process tree. In *WSA '93: Proceedings of the Third International Workshop on Static Analysis*, volume 724 of *LNCS*, pages 112–123. Springer, 1993.

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

[5] G. W. Hamilton. Higher order deforestation. *Fundamenta Informaticae*, 69(1-2):39–61, 2006.

[6] G. W. Hamilton. Distillation: extracting the essence of programs. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 61–70. ACM Press New York, NY, USA, 2007.

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

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

[9] P. A. Jonsson and Johan Nordlander. Supercompiling overloaded functions. Submitted to ICFP 2009, 2009.

[10] M.H. Kabir. *Automatic Inductive Theorem Proving and Program Construction Methods Using Program Transformation.* PhD thesis, Dublin City University, Faculty of Engineering and Computing, School of Computing, 2007.

[11] M. Leuschel. Homeomorphic embedding for online termination of symbolic methods. In *The essence of computation*, volume 2566 of *LNCS*, pages 379–403. Springer, 2002.

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

[13] N. Mitchell and C. Runciman. A supercompiler for core Haskell. In *Implementation and Application of Functional Languages*, volume 5083 of *LNCS*, pages 147–164. Springer, 2008.

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

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

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

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

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

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

[20] V. F. Turchin. Metacomputation: Metasystem transitions plus supercompilation. In *Partial Evaluation*, volume 1110 of *LNCS*, pages 481–509. Springer, 1996.

[21] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88*, volume 300 of *LNCS*, pages 344–358. Springer, 1988.

[22] С.М. Абрамов. *Метавычисления и их применение.* Изд-во "Наука", Москва, 1995.

[23] С.М. Абрамов and Л.В. Пармёнова. *Метавычисления и их применение. Суперкомпиляция.* Университет города Переславля, 2006.

A Примеры

В приложении рассматривается два примера суперкомпиляции. В обоих примерах используются функции высших порядков.

A.1 Итератор

```
data List a = Nil | Cons a (List a);
data Nat = Z | S Nat;

iterate (λn → S n) Z where

iterate = λf x → Cons x (iterate f (f x));
```

Рис. 9: Итератор: входная программа

На языке HLL можно обрабатывать бесконечные структуры данных. Если даны функция `f` и начальное значение `x`, функция `iterate` строит бесконечный список результатов повторного применения функции `f` к `x`:

```
iterate f x = Cons x (Cons (f x) (Cons (f (f x)) (Cons ...)))
```

Целевое выражение программы на Рис. 9 задает бесконечную последовательность натуральных чисел при помощи функции `iterate`.

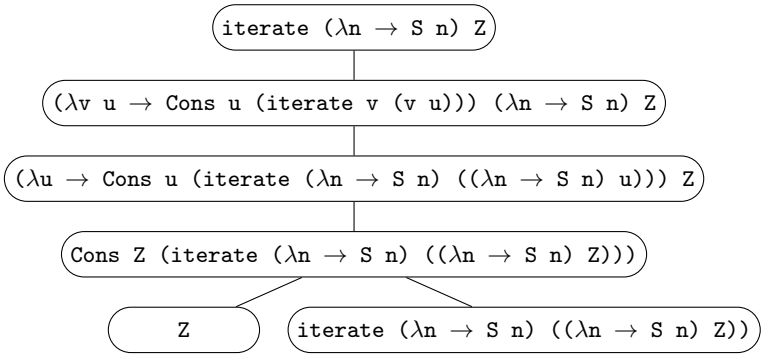

Рис. 10: Итератор: дерево процессов перед обобщением

После нескольких шагов прогонки этого выражения суперкомпилятором HOSC, получается дерево процессов, показанное на Рис. 10. На следующем шаге обнаруживается, что выражение в корне дерева вложено через сцепление в выражение, находящееся в правом листе дерева:

`iterate (λn → S n) Z` $\trianglelefteq_c$ `iterate (λn → S n) ((λn → S n) Z)`

Однако второе выражение не является частным случаем первого. Поэтому делается обобщение сверху: выражение в корне дерева

`iterate (λn → S n) Z`

заменяется на let-выражение

`let z = Z in iterate (λn → S n) z`

и уничтожается соответствующее поддерево. После нескольких шагов дальнейшей прогонки получается дерево, показанное на Рис. 11:

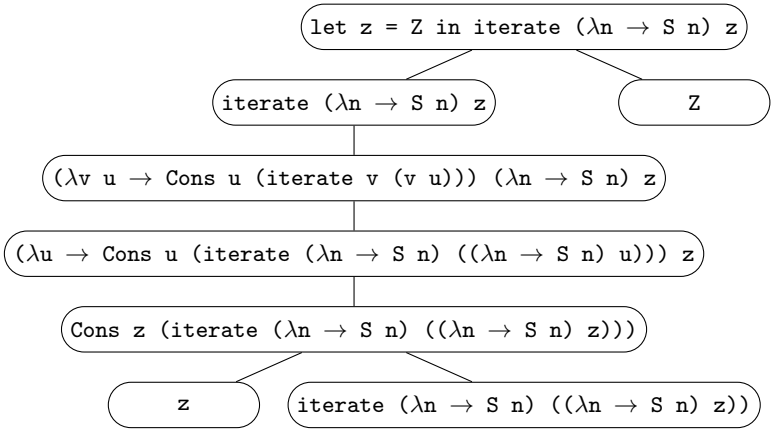

Рис. 11: Итератор: дерево процессов после первого обобщения

На этом шаге распознается вложение:

`iterate (λn → S n) z` $\trianglelefteq_c$ `iterate (λn → S n) ((λn → S n) z)`

На этот раз второе выражение уже является частным случаем первого:

`iterate (λn → S n) z` $\lhd$ `iterate (λn → S n) ((λn → S n) z)`

Таким образом, обобщается второе выражение:

`iterate (λn → S n) ((λn → S n) z)`

заменяется на let-выражение

`let y = (λn → S n) z in iterate (λn → S n) y`

В результате дальнейших метавычислений получается частичное дерево процессов, показанное на Рис. 12. По этому дереву строится остаточная программа, показанная на Рис. 13. Стоит отметить, что функция

`(λn → S n)`

была подставлена в тело новой рекурсивной функции `f`.

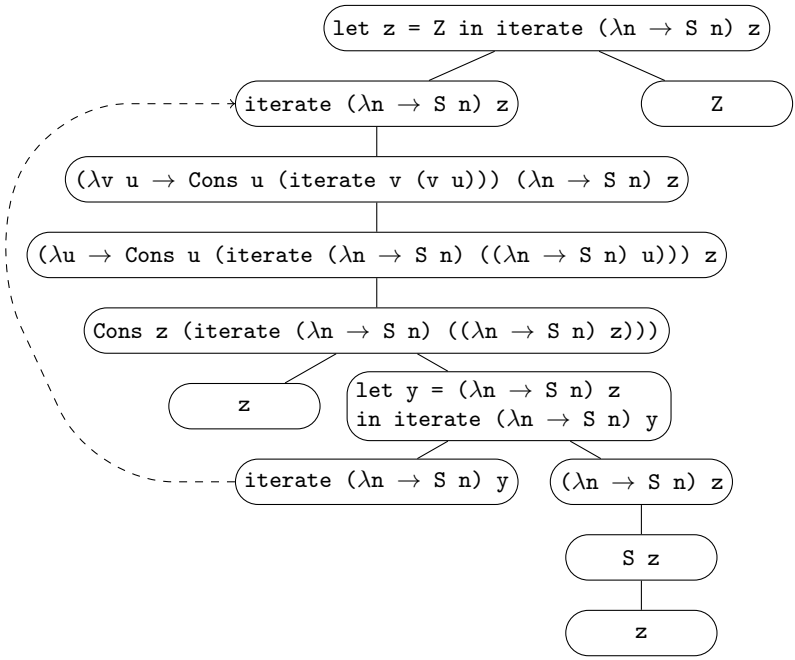

Рис. 12: Итератор: дерево процессов после второго обобщения

```
data List a = Nil | Cons a (List a);
data Nat = Z | S Nat;

letrec
    f = λw → Cons w (f (S w))
in f Z
```

Рис. 13: Итератор: остаточная программа

A.2 Числа Черча

В кодировке Черча натуральное число n представляется в виде функции высшего порядка, которая для любой функции f выдает функцию, повторно применяющую n раз функцию f. Проще говоря, "значением" такого числа является глубина погружения аргумента в данной аппликации:

$$f^n = f \circ f \circ f \circ \ldots \circ f$$

Числа 0, 1, 2, ..., n записываются в кодировке Черча следующим образом:

```
data Nat = Z | S Nat;
data Boolean = False | True;

eq (add x y) (unchurch(churchAdd (church x) (church y))) where

eq = λm y → case m of {
  Z → case n of {Z → True; S n1 → False; } ;
  S m1 → case n of {Z → False; S n1 → eq m1 n1;} ;
};
church = λn → case n of {
  Z → λf t → t;
  S n1 → λf t → f (church n1 f t);
};
unchurch = λn → n (λt → S t) Z;
churchAdd = λm n → (λf t → m f (n f t));
add = λm n → case m of {
  Z → n;
  S m1 → S (add m1 n);
};
```

Рис. 14: Числа Черча: входная программа

$0 = \lambda f\ x \to x$
$1 = \lambda f\ x \to f\ x$
$2 = \lambda f\ x \to f\ (f\ x)$
$\ldots$
$n = \lambda f\ x \to f^n\ x$

Функция сложения двух чисел Черча $churchAdd\ m\ n = m + n$ основана на тождестве $f^{(m+n)}\ x = f^m\ (f^n\ x)$.

В программе на Рис. 14 определяются типы данных `Boolean` и `Nat`. Тип `Nat` используется для представления натуральных чисел в кодировке Пеано с помощью конструкторов `Z` (zero) и `S` (successor). Функция `eq` проверяет два числа Пеано на равенство. Функции `church` и `unchurch` делают перевод чисел из кодировки Пеано в кодировку Черча и наоборот. Функции `add` и `churchAdd` осуществляют сложения натуральных чисел в кодировке Пеано и Черча соответственно.

В целевом выражении проверяется верно ли, что сумма двух чисел Пеано `x` и `y` равна декодированной сумме соответствующих чисел Черча `church x` и `church x`. Стоит отметить, что результатом вычисления целевого выражения в принципе может быть `False`, поскольку `False` присутствует в теле функции `eq`.

Результат преобразования этой программы суперкомпилятором HOSC приведен на Рис. 15. Ясно, что `False` не может быть результатом вычисле-

```
data Nat = Z | S Nat;
data Boolean = False | True ;

case x of {
  Z → case y of {
     Z → True;
     S y1 → letrec f= λz →
               case z of { Z → True; S z1 → f z1;}
             in f y1;
     };
  S x1 →
    letrec g = λv →
        case v of {
          Z → case y of {
              Z → True;
              S y2 → letrec h= λw →
                        case w of { Z → True; S w1 → h w1; }
                     in h y2;
          };
          S v1 → g v1;
        }
    in g x1;
}
```

Рис. 15: Числа Черча: остаточная программа

ния остаточной программы, так как `False` отсутствует в теле остаточной программы.

Структура исходной программы упростилась: практически очевидно, что остаточная программа просто возвращает `True` после деконструкции `x` and `y`.

Несложно показать, что остаточная программа возвращает `True`, если `x` и `y` являются “строгими” значениями (конечными числами Пеано), поскольку результат вычислений зависит только от успешной деконструкции параметров `x` и `y`.