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

Препринты ИПМ • Препринт № 21 за 2010 г.

Ключников И.Г.

Суперкомпилятор HOSC 1.1: доказательство завершаемости

Рекомендуемая форма библиографической ссылки: Ключников И.Г. Суперкомпилятор HOSC 1.1: доказательство завершаемости // Препринты ИПМ им. М.В.Келдыша. 2010. № 21. 27 с. URL: http://library.keldysh.ru/preprint.asp?id=2010-21

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

Илья Ключников

Суперкомпилятор HOSC 1.1: доказательство завершаемости

Москва
2010

Ilya G. Klyuchnikov. Supercompiler HOSC 1.1: proof of termination

The paper contributes the proof of termination 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 1.1: доказательство завершаемости

В работе приводится доказательство завершаемости 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

Цель данной работы – показать, что суперкомпилятор HOSC [6, 7] завершается для любой входной программы.

В [6] была изложена внутренняя структура экспериментального суперкомпилятора HOSC 1.0, работающего с функциональным языком высшего порядка[1].

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

Попытка доказать завершаемость суперкомпилятора HOSC 1.0 закончилась неудачей и привела к построению контрпримера (см. 6). Однако, источник потенциального зацикливания был найден и в суперкомпилятор были внесены необходимые изменения (см. Приложение B). Модифицированный суперкомпилятор HOSC 1.1 гарантированно завершается. В дальнейшем под суперкомпилятором HOSC мы будем понимать HOSC 1.1.

Мы используем инструментарий для доказательства завершаемости *абстрактных преобразователей программ*, разработанный Сёренсеном [14], поскольку суперкомпилятор HOSC можно рассматривать как абстрактный преобразователь программ (частичных деревьев процессов).

Построение для данной программы частичного дерева процессов суперкомпилятором HOSC можно неформально описать так [6]. Построение начинается с того, что в корень дерева помещается исходное (целевое) выражение. Затем HOSC добавляет дочерние узлы к листьям дерева, пока все листья не станут *обработанными*.

Пусть β - необработанный лист строящегося дерева:

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: $\beta \rightrightarrows \alpha$, тем самым лист β становится обработанным.

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.

Результатом шага прогонки является подвешивание к узлу β дочерних узлов, и помещения в них результата метавычисления $\beta.expr$.

Теорема 1 (Завершаемость суперкомпилятора HOSC)**.** *Суперкомпилятор HOSC 1.1 завершается на любой входной программе.*

Идея доказательства состоит в следующем:

- Шаг 1 может выполниться подряд только конечное число раз, так как при прогонке тривиального узла, выражения в дочерних узлах будут *строго меньше* по размеру.

- Шаг 2 не влияет на завершаемость, так как не изменяет узлы дерева и для конечного дерева может выполниться конечное число раз.

- Таким образом, любая последовательность шагов 1 и 2 конечна. Предположим, что HOSC выполняет шаг 3 или шаг 4, обобщая выражение в узле. Мы покажем, что обобщения не могут продолжаться бесконечно, так как обобщение уменьшает "размер выражения".

- Остается показать, что выполнение шага 5 не может повторяться бесконечно. Это следует из того, что $\trianglelefteq|_\rho$ (взятое за основу для $\trianglelefteq_c$) – вполне-квазиупорядочение (см. ниже). Поэтому любая бесконечная последовательность шагов 5, включает шаг 4.

Дальнейшая часть работы структурирована следующим образом:

В главе 2 изложена часть инструментария Сёренсена, *необходимая* для построения доказательства завершаемости HOSC.

Глава 3 объясняет, почему на гомеоморфное вложение $\trianglelefteq|_\rho$ (используемое как свисток) наложены дополнительные требования, и почему эти требования имеют значение для языка с функциями высших порядков.

В главе 4 показывается, что $\trianglelefteq|_\rho$ является вполне-квазиупорядочением *на множестве выражений в узлах дерева процесса.* (Отметим, что это неверно для произвольного множества выражений.)

В главе 5 показывается, что HOSC завершается на любой входной программе, так как для него выполняются условия Сёренсена, *достаточные* для завершаемости абстрактного преобразователя программ.

2 Абстрактные преобразователи программ

В общем, мы используем обозначения из [6]. Однако, нам потребуется различать HLL выражения (присутствующие как в текстах входных программ на языке HLL, так и в узлах деревьев процессов) и let-выражения (которые могут появляться только в узлах дерева процессов).

- $\mathcal{E}$ обозначает множество HLL выражений,

- $\mathcal{L}$ обозначает множество $\mathcal{E}$, дополненное множеством let-выражений.

Любое HLL выражение $e \in \mathcal{E}$ считается эквивалентным *let in e*.

Определение 2 (Квазиупорядочение)**.** *Пусть на множестве S задано отношение $\leq$. $(S, \leq)$ называется квазиупорядочением (quasi-order), если $\leq$ транзитивно и рефлексивно. Мы пишем $s < s'$, если $s \leq s'$ и $s' \not\leq s$.*

Определение 3 (Вполне фундированное квазиупорядочение)**.** *Пусть $(S, \leq)$ - квазиупорядочение. $(S, \leq)$ является вполне фундированным квазиупорядочением (well-founded quasi-order), если не существует бесконечной последовательности $s_0, s_1, \ldots \in S$ такой, что $s_0 > s_1 > \ldots$.*

Определение 4 (Вполне-квазиупорядочение). *Пусть $(S, \leq)$ - квазиупорядочение. $(S, \leq)$ является вполне-квазиупорядочением (well-quasi-order), если для любой бесконечной последовательности $s_0, s_1, \ldots \in S$ существуют $i < j$ такие, что $s_i \leq s_j$.*

Определение 5 (Дерево над множеством выражений). *Пусть $\mathcal{L}$ - некоторое множество выражений. (Частичное) дерево процессов t из [6] является деревом над множеством $\mathcal{L}$ если $\forall \gamma \in t : \gamma.expr \in \mathcal{L}$.*

Множество всех деревьев над $\mathcal{L}$ будем обозначать как $T(\mathcal{L})$

Определение 6 (Абстрактный преобразователь программ). *Абстрактный преобразователь программ на $\mathcal{L}$ – отображение $M: T(\mathcal{L}) \to T(\mathcal{L})$.*

Суперкомпилятор HOSC является абстрактным преобразователем программ на $\mathcal{L}$, где $\mathcal{L}$ – множество всех HLL выражений, расширенное let-выражениями.

Определение 7 (Завершаемость преобразователя программ). *(1) Абстрактный преобразователь программ M на $\mathcal{L}$ завершается на $t \in T(\mathcal{L})$, если $M^i(t) = M^{i+1}(t)$ для некоторого i. (2) Преобразователь M на $\mathcal{L}$ завершается, если он завершается на всех деревьях $t \in T(\mathcal{L})$, состоящих из одного узла.*

Для удобства будем обозначать t_0 – начальное преобразуемое дерево, t_i – дерево после i шагов работы преобразователя.

Утверждение 8 (Преобразователь Коши). *Пусть $(\mathcal{L}, \leq)$ – вполне фундированное квазиупорядочение. Абстрактный преобразователь программ M на $\mathcal{L}$ является преобразователем Коши, если в ходе его работы:*

$$t_{i+1} = t_i\{\gamma := t'\}$$

для некоторого узла γ и выполняется одно из следующих условий:

- $\gamma \in leaves(t_i)$ *и* $\gamma.expr = t'.root.expr$
- $\gamma.expr > t'.root.expr$

Утверждение 9 (Конечный непрерывный предикат). *Пусть $\{\mathcal{L}_1, \mathcal{L}_2\}$ - разбиение $\mathcal{L}$, $(\mathcal{L}_1, \leq_1)$ – вполне-квазиупорядочение, $(\mathcal{L}_2, \leq_2)$ – вполне фундированное квазиупорядочение. Тогда $p: T(\mathcal{L}) \to \mathbb{B}$, определенное как*

$$p(t) = \begin{cases} 0 & \text{если } \exists \alpha, \beta : \alpha \text{ - предок } \beta, \\ & \alpha.expr, \beta.expr \in \mathcal{L}_1 \text{ и } \alpha.expr \leq_1 \beta.expr \\ 0 & \text{если } \exists \alpha, \beta : \alpha \to \beta,\ \alpha.expr, \beta.expr \in \mathcal{L}_2 \text{ и } \alpha.expr \not> _2 \beta.expr \\ 1 & \text{в противном случае} \end{cases}$$

является конечным непрерывным предикатом.

Определение 10 (Внутренность дерева)**.** *Корневой узел дерева t и все его узлы за исключением листьев – внутренность дерева t.*

Внутренность t обозначается t^0

Утверждение 11. *Пусть $p : T(\mathcal{L}) \to \mathbb{B}$ – конечный непрерывный предикат, тогда q, определяемое как $q(t) = p(t^0)$, – также непрерывный конечный предикат.*

Теорема 12 (Сёренсен)**.** *Пусть абстрактный преобразователь программ $M : T(\mathcal{L}) \to T(\mathcal{L})$ сохраняет предикат $p : T(\mathcal{L}) \to \mathbb{B}$. Если*

1. *M – преобразователь Коши, и*
2. *p – непрерывный конечный предикат,*

то M завершается.

3 Гомеоморфное вложение $\trianglelefteq\big|_\rho$

Самой сложной частью любого суперкомпилятора является алгоритм обобщения, который гарантирует завершаемость суперкомпилятора на любой программе, предотвращая построение бесконечного дерева процессов. Самой сложной задачей является принятие решения о том, какое выражение нужно обобщить. В суперкомпиляции эта часть алгоритма обобщения исторически называется *свистком* [15, 16].

Не только в суперкомпиляции, но и в других методах преобразования программ в качестве свистка хорошо себя зарекомендовало отношение гомеоморфного вложения [8, 9]. Свисток, основанный на гомеоморфном вложении, сравнивает выражение в текущем узле с выражениями в предках. Если свисток обнаруживает, что два выражения *синтаксически* похожи, то суперкомпилятор обобщает одно из двух выражений чтобы предотвратить появление бесконечных ветвей в частичном дереве процессов. Таким образом, *существенным* свойством свистка является то, что он будет срабатывает на последовательности выражений, *порождаемых прогонкой*. Говоря формально, свисток основан на *вполне квазиупорядочении*.

3.1 Простейший случай: язык первого порядка

Рассмотрим язык первого порядка [13, 14], синтаксис которого представлен на Рис. 1. Арность функторов (функций и конструкторов) фиксирована и конечна. Отношение гомеоморфного вложения для такого языка определено на Рис. 2. Тонкость в том, что все переменные являются свободными и не различаются.

Обозначим E_0 – множество выражений, определяемых грамматикой 1.

$$e ::= v \mid c(e_1, \ldots, e_n) \mid f(e_1, \ldots, e_n)$$

Рис. 1: Язык первого порядка: синтаксис выражений

Переменные	Погружение	Сцепление
$v_1 \trianglelefteq v_2$	$\dfrac{\exists i : e \trianglelefteq e'_i}{e \trianglelefteq \phi(e'_1, \ldots, e'_k)}$	$\dfrac{\forall i : e_i \trianglelefteq e'_i}{\phi(e_1, \ldots, e_k) \trianglelefteq \phi(e'_1, \ldots, e'_k)}$

Рис. 2: Язык первого порядка: вложение

Теорема 13 (Крускал)**.** $(E_0, \trianglelefteq)$ – *вполне-квазиупорядочение при условии, что конструкторов и функций – конечное число.*

Доказательство. Сворачиваем все переменные в одну константу (конструктор без аргументов), отличную от остальных функторов, получаем выражения без переменных и используем доказательство из [3]. □

Вложение на Рис. 2 обладает свойством, существенным для суперкомпиляции: если два выражения вложены через сцепление, то существует *нетривиальное обобщение* этих выражений.

3.2 Связанные переменные

В отличие от языка первого порядка, рассматриваемого Сёренсеном, суперкомпилятор HOSC работает с языком HLL (с функциями высшего порядка). В HLL выражениях присутствуют связанные переменные в в λ-абстракциях и case-выражениях. Концептуально связанные переменные в case-выражениях не отличаются от связанных переменных в λ-абстракциях. Поэтому в дальнейшем проблемы, относящиеся к связанным переменным, рассматриваются только для λ-абстракций.

λ-абстракцию можно рассматривать как "особый" конструктор при проверке на гомеоморфное вложение. Однако, при упрощенном подходе во время суперкомпиляции могут возникнуть трудности, так как вложение через сцепление не подразумевает наличие нетривиального обобщения. Действительно, если не различать связанные переменные, то следующие выражения "наивно" сцеплены, но нет нетривиального обобщения:

```
λx y → Pair x y
λx y → Pair y x
```

Однако, если различать связанные переменные, данные выражения не вкладываются. Несмотря на синтаксическое сходство двух выражений,

соответствующие безымянные функции различаются *по смыслу*. Поэтому в определении расширенного вложения в [6] используется таблица ρ для записи соответствия связанных переменных. Пусть $\trianglelefteq^1$ – вложение, учитывающее соответствие связанных переменных.

Однако, $\trianglelefteq^1$ еще достаточно грубо сравнивает выражения, поскольку существуют выражения, сцепленные через $\trianglelefteq^1$, для которых отсутствует нетривиальное обобщение. Например, следующие выражения:

```
λx → x
λx → (S x)
```

Поэтому в пункте 4.3 в [6] вводится дополнительное требование: выражение, свободные переменные которого уже есть в таблице соответствия ρ, не может быть погружено в другое выражение. Обозначим через $\trianglelefteq^2$ – вложение $\trianglelefteq^1$, расширенное требованием пункта 4.3.

3.3 Высший порядок и арность

Теорема Крускала применима к языку первого порядка (Рис. 1), так как арность функторов (функций и конструкторов) фиксирована, множество функторов в исходной программе конечно, и во время прогонки не могут появиться функторы, отсутствующие в исходной программе.

Однако, в случае языка высшего порядка, функция может быть частично применена, поэтому, потенциально, во время прогонки могут появляться вызовы функций произвольной "арности": `f`, `f x`, `f x y` и т.д. Вдобавок, аргументами функций могут быть значения-функции. Поэтому логично рассматривать имена функций как переменные, а не функторы, и ввести специальный конструктор для представления аппликации. Это можно сделать двумя способами:

1. Ввести только один бинарный конструктор `App`, первый аргумент которого – голова аппликации (которая может быть аппликацией), а второй – аргумент аппликации. Обозначим такую кодировку $\mathcal{A}_2$.

2. Не допускать аппликации быть в голове аппликации - ввести множество *различных* конструкторов `App2`, `App3` ... разной арности. Обозначим такую кодировку $\mathcal{A}_*$.

Выражение `f x y` будет представимо следующими способами:

1. `f x y = App(App(f, x), y)`

2. `f x y = App3(f, x, y)`

В обоих случаях если выражения e_1 и e_2 вложены через сцепление $\trianglelefteq^2$, то существует их нетривиальное обобщение. Также ясно, что при кодировке $\mathcal{A}_2$ свисток будет свистеть чаще, чем при $\mathcal{A}_*$.

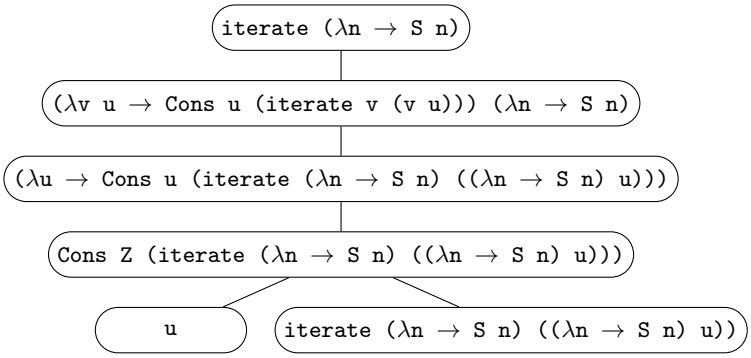

Рис. 3: Прогонка выражения `iterate (λn →S n)`

В принципе, в суперкомпиляторе может быть использован любая из кодировок. Однако первый подход неприменим для суперкомпилятора HOSC из-за того, что если во время прогонки встречается λ-абстракция, то HOSC начинает преобразовывать тело это абстракции.

Рассмотрим прогонку выражения `iterate (λn →S n)`. После нескольких шагов получится дерево, показанное на Рис. 3.

При использовании кодировки $\mathcal{A}_2$ верхнее выражение будет вложено через сцепление в правое нижнее выражение:

`App(iterate, λn→ S n)` $\leq_c$ `App(App(iterate, λn→ S n), App(λn→ S n, u))`

Верхнее выражение будет обобщено до

```
let f = iterate in f (λn→ S n)
```

и функция `iterate` не будет проспециализирована относительно своего первого аргумента.

Использование кодировки $\mathcal{A}_*$ приведет к специализации вызова функции `iterate`, и в общем случае предоставляет больше возможностей для специализации. Именно поэтому определение вложения в [6] основано (неявно) на использовании кодировки $\mathcal{A}_*$. Будем обозначать через $\trianglelefteq^3$ вложение из [6].

4 Вполне-квазиупорядочение $\trianglelefteq|_\rho$

Цель данной секций – показать, что отношение $\trianglelefteq|_\rho$ (оно же – $\trianglelefteq^3$) является вполне-квазиупорядочением на любой последовательности выражений получаемых при прогонке во время суперкомпиляции. Напомним, что отношение $\trianglelefteq^3$ – это отношение $\trianglelefteq$, с наложенными на него ограничениями. В следующих разделах мы покажем, что несмотря на дополнительные

ограничения, $\trianglelefteq^3$ остается вполне-квазиупорядочением последовательности выражений, возникающих в процессе *метавычислений*.

Определение 14 (Выражения, достижимые метавычислениями)**.** *Пусть дана программа prog. Выражение e достижимо для метавычислений, если оно находится в дереве процессов, полученном при прогонке программы prog. Множество выражений, достижимых метавычисление метавычисления программы prog, обозначается* $\mathcal{M}(prog)$.

В следующих разделах мы покажем что отношение $\trianglelefteq^3$ является квазиупорядочением на $\mathcal{M}(prog)$. Это достигается заменой представления связанных переменных таким образом, что теорема Крускала будет применима.

Вначале избавимся от связанных переменных в λ-абстракциях и case-выражениях.

4.1 Замена case-выражений на конструкторы

Избавимся от связанных переменных в case-выражениях. Это можно сделать, представив case-выражение в виде композиции специального конструктора и λ-абстракцй. Имя конструктора определяется типом селектора, первый аргумент конструктора – селектор, остальные – ветви, представленные λ-абстракциями. Ветви упорядочены в соответствии с порядком перечисления конструкторов в декларации типа.

Например, следующее выражение

```
case x of {Z → Z; S y → S y;}
```

кодируется так:

```
CaseNat(x, Z, λy → S y)
```

Пусть дана программа *prog*. Поскольку *prog* конечна, количество деклараций типов данный в *prog* также конечно. Таким образом, количество возможных case-конструкторов тоже конечно. В процессе прогонки не могут возникать новые case-конструкторы, так как case-выражения в $\mathcal{M}(prog)$ либо устраняются в результате шага редукции, либо вводятся при разворачивании определения функции. Таким образом, множество case-конструкторов, присутствующих в $\mathcal{M}(prog)$, конечно.

4.2 Замена имен переменных на индексы де Брюина

К сожалению, в результате прогонки может возникнуть (потенциально) неограниченное количество связанных переменных в λ-абстракциях. Однако, от них можно избавиться с помощью индексов де Брюина [2], – натуральных чисел, где число k означает “переменная, связанная k-й охватывающей λ-абстракцией”.

При проверке на гомеоморфное вложение, можно считать индексы де Брюина специальными константами. Будет показано, что, в отличие от имен переменных, множество индексов де Брюина на $\mathcal{M}(prog)$ конечно.

После замены именованных переменных на индексы де Брюина, предыдущее выражение принимает вид:

```
CaseNat(x, Z, λS 1)
```

Результат кодирование выражения e с помощью case-конструкторов и индексов де Брюина обозначается $\mathcal{E}_1[\![e]\!]$, результат кодирования программы $prog$ – $\mathcal{E}_1[\![prog]\!]$

Лемма 15 ($\mathcal{E}_1$, $\trianglelefteq$, $\trianglelefteq^1$)**.** $e_1 \trianglelefteq^1 e_2 \Leftrightarrow \mathcal{E}_1[\![e_1]\!] \trianglelefteq \mathcal{E}_1[\![e_2]\!]$

Доказательство. Структурная индукция по парам исходных выражений и их кодировкам. □

Лемма 16. *Индексы де Брюина ограничены на* $\mathcal{M}(prog)$*.*

Доказательство. Индексы де Брюина, встречающиеся в $\mathcal{E}_1[\![prog]\!]$ (а следовательно, и в закодированном целевом выражении), ограничены в силу конечности $\mathcal{E}_1[\![prog]\!]$. Индекс в $\mathcal{M}_{\mathcal{E}_1}(prog)$ ограничен наименьшей верхней гранью индексов в $\mathcal{E}_1[\![prog]\!]$, поскольку, если выражение e' возникло на шаге прогонки выражения e, то индексы в $\mathcal{E}_1[\![e']\!]$ не могут быть больше индексов в $\mathcal{E}_1[\![e']\!]$ и $\mathcal{E}_1[\![prog]\!]$. Действительно, рассмотрим правила прогонки (Приложение C)

- D_1, D_2, D_8 - шаг прогонки не затрагивает индексы де Брюина.
- D_3. Индекс де Брюина, соответствующий примененной λ-абстракции пропадает (становится свободной переменной), остальные индексы не изменяются
- D_4 - индексы в контексте сохраняются. Индексы в подставленном определении функции ограничены в силу конечности программы.
- D_5 - В силу *нормального порядка* β-редукции, осуществляемой при прогонке, после шага редукции индекс де Брюина, соответствующий v_0, исчезнет, все остальные индексы останутся неизменными.
- D_6, D_7 – аналогично D_5.

□

Лемма 17. $(\mathcal{M}(prog), \trianglelefteq^1)$ – *вполне-квазиупорядочение.*

Доказательство. Число конструкторов в $\mathcal{M}_{\mathcal{E}_1}(prog)$ ограниченно в силу леммы 16. Следовательно, $(\mathcal{M}_{\mathcal{E}_1}(prog), \trianglelefteq)$ – вполне-квазиупорядочение. Из леммы 15 следует, что $(\mathcal{M}(prog), \trianglelefteq^1)$ – вполне-квазиупорядочение. □

Важно отметить, что при шагах прогонки по правилам D_5, D_6, D_7 максимальный индекс не увеличивается из-за *нормального порядка* β-редукции. Однако, при полной β-редукции (которую HOSC не делает), максимальный индекс может увеличиваться:

$(\lambda \mathtt{x} \to \underline{(\lambda \mathtt{y\ z} \to \mathtt{y})\ (\lambda \mathtt{v} \to \mathtt{x})})\ \mathtt{w} \Rightarrow (\lambda \mathtt{x\ z\ v} \to \mathtt{x})\ \mathtt{w}$

При кодировке с индексами де Брюина:

$(\ \lambda \to \underline{(\ \lambda\lambda \to 2)\ (\ \lambda \to 2)})\ \mathtt{w} \Rightarrow (\ \lambda\lambda\lambda \to 3)\ \mathtt{w}$

4.3 Расширенные индексы де Брюина

Пересмотрим кодировку $\mathcal{E}_1$ так, чтобы она учитывала дополнительное требование, накладываемое $\trianglelefteq^2$: выражение, свободные переменные которого уже зарегистрированы в таблице ρ не может быть погружено в другое выражение ([6], раздел 4.3).

Это требование может быть учтено с помощью добавления к каждому индексу де Брюина подиндекса k, такого, что k есть число конструкторов (включая λ-абстракций) между переменной и соответствующей связывающей конструкцией.

Индекс де Брюина отражает только структуру λ-абстракций, в то время, как подиндекс учитывает структуру λ-абстракций и структуру всех остальных конструкторов.

Обозначим через $\mathcal{E}_2$ кодировку $\mathcal{E}_1$, учитывающую подиндексы. Приведем 2 примера:

e	$\mathcal{E}_2[\![e]\!]$
$\lambda \mathtt{x} \to \mathtt{x}$	$\lambda \to 1_1$
$\lambda \mathtt{x} \to \mathtt{S\ x}$	$\lambda \to \mathtt{S}\ 1_2$

Равенство двух расширенных индексов естественным образом определяется как равенство индексов и подиндексов. При проверке на гомеоморфное вложение будем считать различные расширенные индексы де Брюина различными конструкторами.

Лемма 18 ($\mathcal{E}_2$, $\trianglelefteq$, $\trianglelefteq^2$)**.** $e_1 \trianglelefteq^2 e_2 \Leftrightarrow \mathcal{E}_2[\![e_1]\!] \trianglelefteq \mathcal{E}_2[\![e_2]\!]$

Доказательство. Структурная индукция по парам исходных выражений и их кодировкам (аналогично 15) □

Лемма 19. *Подиндексы де Брюина ограничены на* $\mathcal{M}_{\mathcal{E}_2}(prog)$.

Доказательство. Аналогично доказательству леммы 16. □

Лемма 20. $(\mathcal{M}(prog), \trianglelefteq^2)$ – *вполне-квазиупорядочение.*

Доказательство. Количество конструкторов в $\mathcal{M}_{\mathcal{E}_2}(prog)$ ограничено в силу лемм 16 и 19. Значит, $(\mathcal{M}_{\mathcal{E}_2}(prog), \trianglelefteq)$ – вполне-квазиупорядочение. Из леммы 18 следует, что $(\mathcal{M}(prog), \trianglelefteq^2)$ – вполне-квазиупорядочение. □

4.4 Проблема арности

В кодировке $\mathcal{E}_2$ аппликации были представлены с помощью бинарных конструкторов (представление $\mathcal{A}_2$ из пункта 3.3). Рассмотрим кодировку $\mathcal{E}_3$, в которой, в отличие от $\mathcal{E}_2$, аппликации представлены набором конструкторов различной арности (кодировка $\mathcal{A}_*$ из пункта 3.3).

Типизация по Хиндли-Милнеру находит *главные* типы (principal types) выражений в программе. Главный тип может содержать типовые переменные, которые в разных контекстах могут инстанциироваться различными *конкретными* типами. Рассмотрим простую функцию *id*:

```
id :: a → a;
id = λx → x;
```

Функция id имеет тип $id :: \forall a.a \to a$. Поскольку типовая переменная a под квантором инстанциируется в зависимости от контекста, мы ничего не можем сказать об арности символа *id*. В действительности арность символа *id* в конкретном контексте может быть насколько угодно большой. Например, *id* можно применить к скольки угодно аргументам, каждый из которых тоже является *id*!

```
id
id id
id id id
...
id id id id ...
```

Это можно сделать в силу того, что функция *id* - полиморфная.

То есть, если тип некоторого выражения e_0 (более узко - переменной) является полиморфным, то в принципе можно сконструировать *корректно типизированное* выражение $e = e_0\ e_1 \dots e_n$ с любой арностью n.

Однако, если тип t_0 выражения e_0 - не содержит типовых переменных (мономорфный), то легко показать, что возможны только такие *корректно типизированные* выражения $e = e_0\ e_1 \dots e_n$, где арность n ограничена.

Если рассмотреть idA, конкретизацию функции *id* для конкретного типа A, то арность символа idA не может превышать 1. Поскольку $idA\ e :: A$, и не является функцией.

Следующие примеры показывают, почему типизации по Хиндли-Милнеру не допускает появление выражений неограниченной арности:

```
f = λx → f f x;
h = λx → h x h;
```

$$
\begin{array}{rcll}
t & ::= & \tau & \text{(type variable)} \\
 & | & t \to t & \text{(arrow)} \\
 & | & TyCon\ \overline{t_i} & \text{(type constructor)}
\end{array}
$$

Рис. 4: Грамматика типов HLL

```
f1 = λx y → f1 (x y) y;
f2 = λx y → f2 x (y x);
```

Приведенные функции неограниченно увеличивают число аргументов аппликации, но не являются корректно типизированными.

Определение 21 (Арность аппликации)**.** *Арность аппликации $a[\![e]\!]$ выражения определяется по индукции следующим образом:*

$$
\begin{array}{lcl}
a[\![v]\!] & = & 0 \\
a[\![f]\!] & = & 0 \\
a[\![\lambda v_0 \to e_0]\!] & = & 0 \\
a[\![c\ \overline{e_i}]\!] & = & 0 \\
a[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] & = & 0 \\
a[\![e_0\ e_1\ \ldots\ e_n]\!] & = & n
\end{array}
$$

Лемма 22 ($\mathcal{E}_3$, $\trianglelefteq$, $\trianglelefteq^3$)**.** $e_1 \trianglelefteq^3 e_2 \Leftrightarrow \mathcal{E}_3[\![e_1]\!] \trianglelefteq \mathcal{E}_2[\![e_2]\!]$

Доказательство. Структурная индукция по парам исходных выражений и их кодировкам (аналогично доказательству лемм 15 и 18). □

Язык HLL типизован по Хиндли-Милнеру [1]. Синтаксис типов показан на Рис. 4.

Определение 23 (Мономорфный тип)**.** *Тип называется мономорфным, если в нем отсутствуют типовые переменные.*

Определение 24 (Арность типа)**.** *Арность типа $A(t)$ определяется по индукции следующим образом:*

$$
\begin{array}{lcl}
A[\![\alpha]\!] & = & 0 \\
A[\![t_1 \to t_2]\!] & = & 1 + A(t_2) \\
A[\![TyCon\ \overline{t_i}]\!] & = & 0
\end{array}
$$

$\mathcal{M}_{\mathcal{E}_3}(prog)$ обозначает $\{\mathcal{E}_3[\![e]\!] \mid e \in \mathcal{M}(prog)\}$.

Нам нужно показать, что теорема Крускала выполняется на $\mathcal{M}_{\mathcal{E}_3}(prog)$. Если мы покажем, что арность типов выражений $\mathcal{M}_{\mathcal{E}_3}(prog)$ ограничена,

из этого будет следовать ограниченность арности выражений $\mathcal{M}_{\mathcal{E}_3}(prog)$. То есть, все аппликации в $\mathcal{M}_{\mathcal{E}_3}(prog)$ кодируются с помощью конечного числа конструкторов. Следовательно, теорема Крускала применима.

К сожалению, арность полиморфных типов сложно оценивать, так как типовые переменные полиморфного типа могу инстанциироваться функциональными типами, увеличивая таким образом арность типа. С другой стороны, арность мономорфных типов задана раз и навсегда, поскольку в них отсутствуют типовые переменные.

Мы покажем, что для мономорфно типизированной программы арность выражений, возникающих во время прогонки, ограничена в силу ограниченности арности соответствующих типов.

Тонкость, однако, заключается в том, что алгоритм суперкомпиляции, реализованный в HOSC, учитывает, что входная программа корректно типизирована, но явным образом не рассматривает конкретные типы.

Таким образом, можно воспользоваться следующим приемом. Пусть дана программа $prog$ с полиморфными типами, мы можем заменить ее на мономорфно типизированную программу $prog_m$, такую, что HOSC строит одинаковые деревья процессов для $prog$ и $prog_m$. Значит, если арности аппликаций на $\mathcal{M}_{\mathcal{E}_3}(prog_m)$ ограничены, то они ограничены и на $\mathcal{M}_{\mathcal{E}_3}(prog)$, поскольку $\mathcal{M}_{\mathcal{E}_3}(prog_m) = \mathcal{M}_{\mathcal{E}_3}(prog)$.

В следующем разделе мы опишем процедуру *мономорфизации* исходной программы $prog$.

4.4.1 Мономорфизация

При типизации по Хиндли-Милнеру мы строим граф зависимостей вызовов функций [12]. В результате получаем направленный граф, состоящий из строго связанных компонент, отсортированный в обратном топологическом порядке. Внутри каждой компоненты (рекурсивные) функции, принадлежащей данной компоненте, являются *мономорфными* в том смысле, что каждое вхождение имени функции, определяемой в данной компоненте, имеет один и тот же тип (возможно, содержащий типовые переменные). Функции из компоненты SCC_i не зависят от функций из компонент SCC_j при $i < j$. Введем специальную связанную компоненту SCC_0, соответствующую целевому выражению.

Мономорфизация строит из программы $prog$ новую программу $prog_m$, *операционно эквивалентную исходной*. Вначале $prog_m$ совпадает с $prog$. Предполагаем, что каждому подвыражению и функции в $prog_m$ *явно приписан его тип*.

Пусть A - некоторый зафиксированный *базовый* тип, например:

```
data A = A;
```

Мономорфизация шаг за шагом строит новую программу, уменьшая на каждом шаге граф зависимостей строго связанных компонент.

1. Если в типе в целевом выражении или его подвыражениях встречается типовая переменная, то заменяем ее на базовый тип A, превращая таким образом целевое выражение в мономорфное. После этого компонента SCC_0 становится мономорфной.

2. Выбираем компоненту с SCC_i с минимальным $i > 0$. Если такой нет – мономорфизация завершена.

3. Если в SCC_0 нет ссылок на символы, определенных в SCC_i – удаляем SCC_i из графа зависимостей и переходим к шагу 2.

   (a) Встраиваем конкретизацию компоненты SCC_i в компоненту SCC_0: берем *одно* вхождение какой-либо функции f из SCC_i. Вхождение f в SCC_0 имеет мономорфный тип в силу того, что SCC_0 уже мономорфизована. Заменяем это вхождение на f_i, где f_i - новое имя. Копируем определения $f \ldots g$ из SCC_i в SCC_0 с переименованиями $f_i \ldots g_i$. Если в компоненте SCC_i есть имена функций, типы которых содержат типовые переменные, которые не зависят от контекста, то конкретизируем эти типовые переменные базовым типом A. После этого шага компонента SCC_0 останется мономорфной: вхождение f в SCC_0 было мономорфным, значит (в контексте данного вхождения) мы однозначно приписываем мономорфные типы всем (под)выражениям в скопированной компоненте.

   Если в расширенной компоненте SCC_0 остались вхождения символов из SCC_i, – повторяем шаг 3(a), иначе – переходим к шагу 2.

Лемма 25. *Мономорфизация программы завершается за конечное число шагов.*

Доказательство. Количество вершин (строго связанных компонент) и дуг (зависимостей вызовов) в изначальном графе конечно. Шаг 1 выполняется один раз и не изменяет количество вершин и дуг в графе. Шаг 2 также не меняет количество вершин и дуг. На шаге 3 удаляется вершина графа - уменьшается количество вершин. Для SCC_i шаг 3(a) выполняется пока есть вхождение символов из SCC_i в SCC_0. Но число таких вхождений конечно и каждый шага 3(a) устраняет одно из них. □

Другими словами, мономорфизация программы завершается, так как в графе зависимостей нет циклов, выходящих за пределы одной компоненты, и в пределах одной компоненты каждое вхождение символа, определяемого в данной компоненте имеет один и тот же тип.

Лемма 26. *Прогонка мономорфизованной программы строит дерево процессов, совпадающее с деревом процессов, построенным прогонкой исходной программы, с точностью до индексов, возникших в результате мономорфизации.*

Доказательство. Определения мономорфизованных функций $f_1, \ldots, f_i$ совпадают с исходным определением f (с точностью до индексов). С другой стороны, алгоритм прогонки HOSC не зависит от конкретных имен функций. Поэтому если стереть индексы в дереве процессов, порожденном прогонкой мономорфизованной программы $prog_m$, то получится дерево, порожденном прогонкой исходной программы. □

4.4.2 Конечность арности

Лемма 27. *Арность аппликации любого подвыражения, возникающего при прогонке программы prog ограничена максимальной арностью типа подвыражения в мономорфизованной программе* $prog_m$.

Доказательство. Рассмотрим дерево, порождаемое прогонкой $prog_m$. Выражение в любом узле дерева является мономорфным. Поэтому достаточно показать, что ограничена арность типа любого подвыражения, порождаемого прогонкой. Это верно для выражения в корне дерева, являющимся целевым выражением программы $prog_m$. Покажем, что каждый шаг прогонки сохраняет ограниченность арности аппликаций.

- D_1, D_2, D_3, D_8 - при переходе к подвыражениям максимальная арность типа подвыражения не может увеличиться.
- D_4 - максимальная арность после завершения шага не больше максимальной арности до шага и максимальной арности подвыражений в определении f_0.
- D_5, D_6, D_7 - максимальная арность типа подвыражения не увеличивается так как $type(v_0) = type(e_0)$, $type(v_{ik}) = type(e'_k)$, $type(v\ \overline{e'_j}) = type(p_i)$ соответственно.

Поскольку арность аппликации во время прогонки $prog_m$ ограничена, то по лемме 26 она ограничена и при прогонке $prog$. □

Из леммы следует, что в кодировке $\mathcal{M}_{\mathcal{E}_3}(prog)$ количество конструкторов $App_2, \ldots, App_n$ ограничено.

Следствие 28. *Множество конструкторов* $App_2, \ldots, App_n$, *появляющихся в* $\mathcal{M}_{\mathcal{E}_3}(prog)$, *конечно.*

Теорема 29. $(\mathcal{M}(prog), \trianglelefteq^3)$ – *вполне-квазиупорядочение.*

e	$\mathcal{E}_3[\![e]\!]$
`map f`	`App2(map, f)`
`map (compose f g)`	`App2(map, App3(compose, f, g))`
λ`x` $\rightarrow$ `Cons x Nil`	$\lambda\rightarrow$ `Cons(`1_2`, Nil)`
`case x of` `{Z` $\rightarrow$ `x; S b` $\rightarrow$ `f (g b);}`	`CaseNat(x, x,` $\lambda\rightarrow$ `App2(f, App2(g,` 1_3`)))`

Рис. 5: Примеры кодирования

Доказательство. Множество конструкторов в $\mathcal{M}_{\mathcal{E}_3}(prog)$ ограничено в силу лемм 16, 19 и 27. Значит, $(\mathcal{M}_{\mathcal{E}_3}(prog), \trianglelefteq)$ – вполне-квазиупорядочение. По лемме 22 $(\mathcal{M}(prog), \trianglelefteq^3)$ – вполне-квазиупорядочение. □

Следствие 30. *Так как $\trianglelefteq^3$ совпадает с $\trianglelefteq|_\rho$, мы показали, что отношение $\trianglelefteq|_\rho$ – вполне-квазиупорядочение на $\mathcal{M}(prog)$.*

Следствие 31. *Отношение $\trianglelefteq_c|_\rho$ – вполне-квазиупорядочение на $\mathcal{M}(prog)$.*

Доказательство. Следует из теоремы 29 и леммы Хигмана [3]. □

4.5 Кодировка $\mathcal{E}_3$

В разделе 4.5 в [6] примедены примеры вкладывающихся и не вкладывающихся по $\trianglelefteq|_\rho$ выражений. Некоторые выражения из этих примеров в кодировке $\mathcal{E}_3$ приведены на Рис. 5.

Близкая к $\mathcal{E}_1$ кодировка использовалась [10] для решении проблемы конфликта имен при дефорестации.

Если кодировать выражения через $\mathcal{E}_3$, то нет нужды в использовании таблицы соответствия ρ при проверке на расширенное вложение $\trianglelefteq|_\rho$.

5 Завершаемость суперкомпилятора HOSC 1.1

Все теоремы и доказательства этой секции повторяют (с незначительными модификациями) материал из [14].

Идея – показать, что выполняются условия теоремы 12.

Мы не будем явно рассматривать операцию $fold$, так как в [6] она использовалась только для удобства записи алгоритма построения остаточной программы, – просто будем считать, что узел β является обработанным, если выполняется условие для совершения операции $fold$.

Лемма 32. *Суперкомпилятор HOSC - преобразователь Коши.*

Доказательство. Определим отношение $\succeq$ на множестве $\mathcal{L}$ как:

$$let\ x'_1 = e'_1, \ldots, x'_m = e'_m\ in\ e' \succeq let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \Leftrightarrow m = 0\ \&\ n \geq 0$$

Просто показать, что $\succeq$ – вполне фундированное квазиупорядочение. Покажем, что для на любом шаге построения частичного дерева процессов

$$t_{i+1} = t_i\{\gamma := t'\}$$

для некоторого узла γ и выполняется одно из следующих условий:

- $\gamma \in leaves(t_i)$ и $\gamma.expr = t'.root.expr$ (прогонка),
- $\gamma.expr \succ t'.root.expr$ (обобщение).

Достаточно проанализировать возможные операции на каждом шаге:

1. $t_{i+1} = drive(t_i, \beta) = t_i\{\beta := t'\}$, где $\beta \in leaves(t_i)$ и $t' = \beta.expr \rightarrow e_1, \ldots, e_n$. Значит:
$$\beta.expr = t'.root.expr$$

2. $t_{i+1} = abstract(t_i, \beta, \alpha) = t_i\{\beta := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \rightarrow\}$, где $\alpha = ancestor(t,\ \beta,\ \lessdot)$, $\alpha.expr \not\simeq \beta.expr$, β – нетривиальный узел, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \{\}, \theta) = \alpha.expr \sqcap \beta.expr$, $\beta.expr = e\theta$. Тогда $\alpha.expr \equiv e$ и $\beta.expr \equiv \alpha.expr\theta$, но $\alpha.expr \not\simeq \beta.expr$, то есть $n > 0$. Таким образом:
$$\beta.expr \succ let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e = t'.root.expr$$

3. $t_{i+1} = abstract(t_i, \alpha, \beta) = t_i\{\alpha := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \rightarrow\}$, где $\alpha = ancestor(t,\ \beta,\ \trianglelefteq_c)$, $\alpha.expr\ /\!\lessdot \beta.expr$, β – нетривиальный узел, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $\alpha.expr = e\theta_1$. Тогда $\alpha.expr \not\equiv e$, но $\alpha.expr = e\theta_1$, то есть $n > 0$. Таким образом:
$$\alpha.expr \succ let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e = t'.root.expr$$

□

Определение 33 (Множество выражений шага)**.** *Множество $\mathcal{M}^i_{\mathcal{SC}}(prog)$ определяется как множество выражений, находящихся в узлах дерева t_i на i-м шаге построения частичного дерева процессов.*

Определение 34 (Множество суперкомпиляции)**.** *Множество $\mathcal{M}_{\mathcal{SC}}(prog)$ определяется как объединение множеств:*

$$\mathcal{M}_{\mathcal{SC}}(prog) = \bigcup \mathcal{M}^i_{\mathcal{SC}}(prog)$$

Лемма 35. $(\mathcal{E} \cap \mathcal{M}_{\mathcal{SC}}(prog), \trianglelefteq^3)$ – *вполне-квазиупорядочение.*

Доказательство. Рассмотрим $\mathcal{M}_{\mathcal{E}_3}(prog)$. Легко показать, что при обобщении $e = let\ \overline{v_i = e_i;}\ in\ e_g$ индексы де Брюина, подиндексы де Брюина и максимальная арность аппликации не увеличиваются. Отсюда по лемме 29 следует, что $(\mathcal{E} \cap \mathcal{M}_{\mathcal{SC}}(prog), \trianglelefteq^3)$ – вполне-квазиупорядочение. □

Следствие 36. $\trianglelefteq|_\rho$ – *вполне-квазиупорядочение на* $\mathcal{E} \cap \mathcal{M}_{\mathcal{SC}}(prog)$.

Доказательство. Аналогично следствию 30. □

Следствие 37. $\trianglelefteq_c|_\rho$ – *вполне-квазиупорядочение на* $\mathcal{E} \cap \mathcal{M}_{\mathcal{SC}}(prog)$.

Доказательство. Аналогично следствию 31. □

Лемма 38. *Суперкомпилятор HOSC при построении частичного дерева процессов сохраняет конечный непрерывный предикат.*

Доказательство. Определим $l : \mathcal{L} \to \mathcal{E}$:

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

Определим $\sqsupseteq$ на множестве $\mathcal{L}$:

$$e_1 \sqsupseteq e_2 \Leftrightarrow (\mathcal{S}[\![l(e_1)]\!] > \mathcal{S}[\![l(e_2)]\!]) \vee (\mathcal{S}[\![l(e_1)]\!] = \mathcal{S}[\![l(e_2)]\!] \wedge l(e_1) \gtrdot l(e_2))$$

Так как $\lessdot$ – вполне фундированное квазиупорядочение, то $\sqsubseteq$ – вполне фундированное квазиупорядочение. Рассмотрим предикат $q : T(l) \to \mathbb{B}$, определенный как

$$q(t) = p(t^0)$$

где t^0 - внутренность t, а предикат $p : T(l) \to \mathbb{B}$:

$$p(t) = \begin{cases} 0 & \text{если } \exists \alpha, \beta : \alpha = ancestor(t,\ \beta,\ \trianglelefteq_c) \text{ и } \alpha, \beta \text{ - нетривиальные} \\ 0 & \text{если } \exists \alpha, \beta : \alpha \to \beta,\ \alpha, \beta \text{ - тривиальные и } \alpha.expr \not\trianglerighteq \beta.expr \\ 1 & \text{в противном случае} \end{cases}$$

Множества тривиальных и нетривиальных выражений представляют разбиение множества $\mathcal{M}_{\mathcal{SC}}(prog)$. $\trianglelefteq_c|_\rho$ – вполне-квазиупорядочение на нетривиальных выражениях $\mathcal{M}_{\mathcal{SC}}(prog)$ и $\sqsubseteq$ – вполне фундированное квазиупорядочение на тривиальных выражениях $\mathcal{M}_{\mathcal{SC}}(prog)$. Следовательно (см. утверждение 11), p и q – конечные непрерывные предикаты. Осталось показать, что HOSC сохраняет предикат q.

Пусть дано дерево t и узел β, будем считать β *корректным* в дереве t, если выполняются оба следующих условия:

(i) β – нетривиальный узел, β не является листом дерева $t \Rightarrow$ $ancestor(t, \beta, \trianglelefteq_c) = \bullet$

(ii) $\exists \alpha : \alpha \to \beta$ и α – тривиальный узел $\Rightarrow \alpha.expr \sqsupset \beta.expr$

Дерево считается корректным, если все его узлы – корректные.

Легко видеть, что $q(t) = 1$, если t – корректное. Достаточно показать, что на каждом шаге i t_i - корректное. Доказываем по индукции по i.

Для $i = 0$, (i)-(ii) выполняются.

Для $i > 0$, предполагаем, что t_i - корректное. Для t_{i+1} рассмотрим различные операции этого шага:

1. $t_{i+1} = drive(t_i, \beta) = t_i\{\beta := t'\}$, где $\beta \in leaves(t_i)$ и $t' = \beta.expr \to e_1, \ldots, e_n$ и $e_1, \ldots, e_n = \mathcal{D}[\![\beta.expr]\!]$. Необходимо показать, что β и $children(t_{i+1}, \beta)$ – корректные в t_{i+1}.

   Рассмотрим β: (1) если β – нетривиальный узел, алгоритм гарантирует, что для β выполнено условие (i), условие (ii) верно по гипотезе индукции, (2) если β – нетривиальный узел, то для β условие (i) выполнено тривиальным образом, и условие (ii) верно по гипотезе.

   Рассмотрим $children(t_{i+1}, \beta)$: (1) если β – нетривиальный узел, условия (i) и (ii) выполнены тривиальным образом для $children(t_{i+1}, \beta)$ (2) если β – тривиальный узел, то условие (i) выполнено тривиальным образом, а условие (ii) выполняется т. к. $\forall i : \beta.expr \sqsupset e_i$, – здесь очень важно, что при прогонке тривиального выражения, размер выражения уменьшается (см. 6).

2. $t_{i+1} = abstract(t_i, \beta, \alpha) = t_i\{\beta := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \to\}$, где $\alpha = ancestor(t,\ \beta,\ \lessdot)$, $\alpha.expr \not\simeq \beta.expr$, β – нетривиальный узел, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \{\}, \theta) = \alpha.expr \sqcap \beta.expr$, $\beta.expr = e\theta$. Необходимо показать, что β – корректен в t_{i+1}.

   Условие (i) выполняется тривиальным образом. Из гипотезы индукции и факта, что $l(\beta_i.expr) = l(let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e) = l(\beta_{i+1}.expr)$, следует выполнение условия (ii).

3. $t_{i+1} = abstract(t_i, \alpha, \beta) = t_i\{\alpha := let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e \to\}$, где $\alpha = ancestor(t,\ \beta,\ \trianglelefteq_c)$, $\alpha.expr\ /\!\lessdot \beta.expr$, β – нетривиальный узел, $\alpha.expr, \beta.expr \in \mathcal{E}$, $\alpha.expr \lessdot \beta.expr$, $(e, \theta_1, \theta_2) = \alpha.expr \sqcap \beta.expr$, $\alpha.expr = e\theta_1$. Необходимо показать, что α – корректен в t_{i+1}.

   Условие (i) выполняется тривиальным образом. Из гипотезы индукции и факта, что $l(\alpha_i.expr) = l(let\ x_1 = e_1, \ldots, x_n = e_n\ in\ e) = l(\alpha_{i+1}.expr)$, следует выполнение условия (ii).

□

Следствие 39. *Построение дерева частичных процессов суперкомпилятором HOSC завершается.*

Доказательство. Из лемм 32, 38 следует, что выполняются условия теоремы 12. □

```
data D = F (D → D);

λf → apply (F (λx → f (apply x x)))(F (λx → f (apply x x)))
where

apply = λx → case x of { F f → f; };
```

Рис. 6: Оператор неподвижной точки с глобальным определением

```
data D = F (D → D);

λf → (λy → case y of { F g → g; })
    (F (λx → f ((λy → case y of { F g → g; }) x x)))
       (F (λx → f ((λy → case y of { F g → g; }) x x)))
```

Рис. 7: Оператор неподвижной точки без глобального определения

6 HOSC 1.0 и завершаемость

Суперкомпилятор HOSC 1.0 мог зациклится. На Рис. 6 показано определение неподвижной точки без использования явной рекурсии. Суперкомпиляция данного выражения завершается, так как при построении дерева процессов встречаются нетривиальные выражения вида

```
apply (F (λx → f (apply x x)))(F (λx → f (apply x x)))
```

Однако, если заменить вызов функции `apply` на ее определение, то HOSC 1.0 не завершится! Причина заключается в том, что прогонка измененной программы (Рис. 7) будет порождать только тривиальные выражения, которые не проверяются на вложение.

Эта проблема решена в суперкомпиляторе HOSC 1.1 B.

7 Обзор литературы

Первое полное описание суперкомпилятора с доказательством завершаемости можно найти в магистерской работе Сёренсена [13]. Суперкомпилятор Сёренсена работает с ленивым функциональным языком первого порядка. Затем Сёренсеном был разработан инструментарий для доказательство завершаемости преобразователей программ, работающих с деревьями [14].

Митчел и Рансиман описали суперкомпилятор Supero для подмножества функционального языка с ленивой семантикой Haskell [11]. Доказа-

тельство завершаемости Supero не опубликовано.

Джонссон и Нордландер описали суперкомпилятор для функционального языка с функциями высших порядков с семантикой вызовов по значению [5] и показали его завершаемость [4].

Данная работа отличается от [11, 5] в следующем:

- При проверке на гомеоморфное вложение HOSC различает связанные переменные (вкладываются только соответствующие связанные переменные). Поэтому свисток HOSC срабатывает реже, чем свисток в [11, 5], и происходит меньше переобобщений.
- HOSC проверяет только вложение через сцепление – это гарантирует наличие наиболее тесного обобщение. HOSC однозначно строит обобщение. В [11, 5] свисток срабатывает и при погружении – в данном случае может отсутствовать наиболее тесное обобщение и возникает неоднозначность при построении обобщения.
- В [11, 5] глобальные функции имеют фиксированную арность и каждый вызов глобальной функции должна быть насыщенным по аргументам. Поэтому в [11, 5] иногда в программу приходится искусственно вставлять λ-абстракции, чтобы избежать частичного применения глобальной функции. HOSC не имеет такого ограничения и допускает любые (корректно типизированные) частичные аппликации.

Главной новизной данной работы является доказательство того, что расширенное гомеоморфное вложение, различающее связанные переменные все равно остается вполне-квазиупорядочением на множестве выражений возникающих во время суперкомпиляции (а не на всем множестве выражении). Однако, этого достаточно для завершаемости суперкомпилятора. Вдобавок, использование более утонченного сложения позволяет в некоторых случая избежать переобобщения.

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] 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.

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

[3] Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1-2):69–116, 1987.

[4] 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.

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

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

[7] 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. Springer, 2010.

[8] M. Leuschel. On the power of homeomorphic embedding for online termination. In *Static Analysis*, volume 1503 of *LNCS*, pages 230–245. Springer, 1998.

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

[10] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 154–165. Springer, 1993.

[11] 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.

[12] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

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

[14] 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. Springer, 1998.

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

[16] V. F. Turchin. The algorithm of generalization in the supercompiler. In *Partial Evaluation and Mixed Computation*, 1988.

A HOSC 1.0: работа над ошибками

В [6] есть неточности и опечатки.

Секция 5.2 должна начинаться:

Правила общего функтора имеют вид:

$$(v, \{v := e_1\} \cup \theta_1, \{v := e_2\} \cup \theta_2) \Rightarrow \ldots$$

Правила общего функтора применимы только тогда, когда

$$e_1 \trianglelefteq_c e_2 \; or \; e_1 \trianglelefteq_v e_2$$

7-е правило прогонки (глава 6, Рис. 5) должно читаться:

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

B HOSC 1.1

HOSC 1.1 несколько иначе чем HOSC 1.0 определяет разбиение узлов дерева на тривиальные и нетривиальные, учитывая размер выражения.

Размер выражения e $\mathcal{S}[\![e]\!]$ определяется по индукции следующим образом:

$$\begin{array}{ll}
\mathcal{S}[\![v]\!] & = 1 \\
\mathcal{S}[\![f]\!] & = 1 \\
\mathcal{S}[\![c\ \overline{e_i}]\!] & = 1 + \sum_i \mathcal{S}[\![e_i]\!] \\
\mathcal{S}[\![\lambda v \to e]\!] & = 1 + \mathcal{S}[\![e]\!] \\
\mathcal{S}[\![e_1\ e_2]\!] & = \mathcal{S}[\![e_1]\!] + \mathcal{S}[\![e_2]\!] \\
\mathcal{S}[\![case\ e_0\ of\ \{\overline{c_i\ \overline{v_{ik}} \to e_i;}\}]\!] & = 1 + \mathcal{S}[\![e_0]\!] + \sum_i \mathcal{S}[\![e_i]\!] \\
\mathcal{S}[\![letrec\ f = e_1\ in\ e_2]\!] & = 1 + \mathcal{S}[\![e_1]\!] + \mathcal{S}[\![e_2]\!]
\end{array}$$

Узел β считается *нетривиальным*, если выполняется одно из следующих условий:

1. $\beta.expr = con\langle f\rangle$,
2. $\beta.expr = con\langle case\ v\ \overline{e_j}\ of\ \{\overline{p_i \to e_i;}\}\rangle$,
3. $\beta.expr = con\langle(\lambda v \to e_0)\ e_1\rangle$ и $\mathcal{S}[\![(\lambda v \to e_0)\ e_1]\!] \leq \mathcal{S}[\![e_0\{v\ :=\ e_1\}]\!]$,
4. $\beta.expr = con\langle case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\}\rangle$ и
   $\mathcal{S}[\![case\ c_j\ \overline{e'_k}\ of\ \{\overline{c_i\ \overline{v_{ik}}\ \to e_i;}\}]\!] \leq \mathcal{S}[\![e_j\{\overline{v_{jk}\ :=\ e'_k}\}]\!]$

Во всех остальных случаях узел β считается *тривиальным*.

В отличие от HOSC 1.0, нетривиальные узлы 3-го и 4-го типа могут быть базовыми и повторными узлами в частичном дереве процессов. Алгоритм генерации остаточной программы изменяется соответствующим образом, – правила обхода нетривиальных узлов 3-го и 4-го типа полностью аналогичны правилам C_7^1, C_7^2 и C_7^3 из [6].

C Дополнительные обозначения

Для удобства ссылок на правила шага прогонки приведем их с нумерацией:

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

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

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

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

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

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

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

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