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

Препринты ИПМ • Препринт № 5 за 2017 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

Гречаник С.А.

Полипрограммы как представление множеств функциональных программ и преобразования над ними

Рекомендуемая форма библиографической ссылки: Гречаник С.А. Полипрограммы как представление множеств функциональных программ и преобразования над ними // Препринты ИПМ им. М.В.Келдыша. 2017. № 5. 31 с. doi:10.20948/prepr-2017-5
URL: http://library.keldysh.ru/preprint.asp?id=2017-5

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

Гречаник Сергей Александрович

Полипрограммы как представление множеств функциональных программ и преобразования над ними

Москва
2017

Sergei Grechanik. Polyprograms as a representation of sets of functional programs, and their transformations

In different program transformation methods there arise objects similar to programs but capable of having multiple definitions of the same function. We will call such objects polyprograms. For example, in the Burstall-Darlington framework such objects are simply sets of recursion equations, and in equality saturation of Tate et al. the corresponding structure is called E-PEG. An important property of polyprograms, which is used in these transformations, is their ability to represent sets of ordinary programs. In this paper we introduce the notion of polyprogram in a non-strict first-order functional language. We define a denotational semantics for polyprograms and describe some possible transformations of polyprograms. We also touch on the subject of extracting ordinary programs from polyprograms.

Supported by Russian Foundation for Basic Research grant No. 16-01-00813-a.

Сергей Гречаник. Полипрограммы как представление множеств функциональных программ и преобразования над ними

В различных методах преобразования программ возникают объекты, подобные программам, но способные содержать несколько определений одной и той же функции — мы будем называть такие объекты полипрограммами. Например, в системе Бёрстолла-Дарлингтона такими объектами являются просто множества рекурсивных уравнений, а в насыщении равенствами Тейта и др. аналогичная структура называется E-PEG. Важным свойством полипрограмм, используемым в этих преобразованиях, является их способность представлять множества обычных программ. В данной работе мы вводим понятие полипрограммы на нестрогом функциональном языке первого порядка, определяем денотационную семантику для полипрограмм и описываем некоторые возможные преобразования полипрограмм. Мы также касаемся темы выделения обычных программ из полипрограммы.

Работа выполнена при поддержке гранта РФФИ № 16-01-00813-а.

Содержание

1 Введение

Многие методы преобразования программ (в частности, суперкомпиляцию и насыщение равенствами) можно рассматривать в качестве частных случаев общей схемы преобразования программ Бёрстолла и Дарлингтона [3]. Данная схема состоит в том, что программа рассматривается как набор уравнений, после чего при помощи некоторых преобразований данный набор уравнений пополняется новыми уравнениями, причём новые уравнения могут как определять новые функции, так и быть дополнительными определениями старых функций. В результате какое-то подмножество этих уравнений выделяется в качестве новой программы.

Таким образом, набор уравнений, в отличие от программ, может обладать одной особенностью: у функций может быть по несколько определений. Такому объекту, однако, уделено не очень много внимания в литературе. Например, в работе Сэндса [12], решающей важную задачу обеспечения корректности всего метода (в оригинальной работе Бёрстолла-Дарлингтона проблема корректности не решалась), используется соглашение, что преобразования добавляют определения только новых функций, и таким образом поддерживается свойство единственности определений. И хотя для преобразований, таких как дефорестация [16] или суперкомпиляция [14; 15], такой подход приемлем, для насыщения равенствами [4] он уже не очень удобен.

В данной работе мы предлагаем уделить таким наборам уравнений больше внимания. Мы называем их *полипрограммами* (сокращение от “поливариантная программа”, термин предложен М.А. Бульонковым). Для полипрограмм, как мы увидим, довольно просто вводится денотационная семантика (хотя для

выполнения некоторых свойств мы рассматриваем не наименьшую неподвижную точку, а все неподвижные точки). С операционной семантикой, однако, не всё так просто.

Одно из применений полипрограмм — представление целого множества программ компактным образом. Действительно, если в полипрограмме n функций и у каждой функции есть по m определений, то полипрограмма представляет $O(m^n)$ различных программ (каждая программа получается путём выбора по одному определению для каждой функции — в общем случае так можно получить программы, не эквивалентные исходной, но пока проигнорируем эту проблему). При этом сама полипрограмма будет содержать всего mn определений. В суперкомпиляции и дефорестации этот факт не используется, но в насыщении равенствами и некоторых видах многорезультатной суперкомпиляции [7; 8] это свойство является ключевым, позволяя вместо применения эвристик по ходу преобразования использовать более точные методы выбора наилучшей программы из множества программ. Без компактного представления такой подход был бы обречён на неудачу из-за комбинаторного взрыва.

Основной вклад данной работы состоит в строгом определении семантики полипрограмм и описании системы преобразований над полипрограммами, похожей на метод насыщения равенствами. Описанная система преобразований имеет практическую реализацию в виде системы доказательства эквивалентностей, описанной в работе [5]. Эта работа является прямым продолжением работы [6].

Дальнейшее изложение построено следующим образом. В Разделе 2 даётся определение понятия полипрограммы для нестрогого функционального языка. В Разделе 3 вводится денотационная семантика полипрограмм на этом языке. В Разделе 4 описывается, как полипрограмма представляет множество программ: оказывается, что не все программы, полученные из полипрограммы наивным способом, семантически эквивалентны исходной полипрограмме, поэтому предлагается проверять выделяемые программы на соответствие признакам структурной и защищённой рекурсии. В Разделе 5 приводятся две системы преобразования полипрограмм: одна соответствует системе Бёрстолла-Дарлингтона и удобна для ручного преобразования полипрограмм, а вторая похожа на насыщение равенствами и хорошо подходит для программной реализации. Раздел 6 посвящён слиянию по бисимуляции — преобразованию, способному доказывать по индукции равенство двух функций. Раздел 7 является заключением.

1.1 Используемые обозначения

Далее будем использовать символ $\equiv$ для разделения левой и правой части определений функций программ и полипрограмм, чтобы отличить его от метатеоретического равенства $=$. Кроме того, часто будем использовать запись $\overline{e_i}$ для обозначения списков вида $e_1, e_2, \ldots, e_n$ для некоторого натурального n (определяется из контекста). Значение n будем также обозначать как $\overline{\max i}$. Также будем вкладывать такие конструкции друг в друга, например, $\overline{f_i(\overline{e_{ij}})}$

обозначает

$$f_1(e_{11}, \ldots, e_{1k_1}), \ldots, f_n(e_{n1}, \ldots, e_{nk_n})$$

для некоторого n и некоторых $k_1, \ldots, k_n$. То, какой индекс к какой верхней черте относится, определяется из контекста. Отметим, что нумерацию мы в таких конструкциях ведём всегда с 1.

Для выражения с подвыражениями $e_1, \ldots, e_n$ будем использовать запись $E\langle e_1, \ldots, e_n\rangle$ или $E\langle\overline{e_i}\rangle$. Для подстановки выражения e в выражение E вместо переменной x будем использовать запись $E\{x \mapsto e\}$.

Будем использовать обозначения $fv(e)$ для множества свободных переменных выражения e и $\overline{fv}(e)$ для списка свободных переменных (через запятую).

2 Понятие полипрограммы

В данной работе мы для определённости будем рассматривать нетипизированный функциональный язык первого порядка с нестрогой (non-strict) семантикой (примерно как в [14]). Под полипрограммами здесь будем понимать именно полипрограммы на этом языке.

Синтаксис определений нашего языка описан на Рис. 1. Будем использовать соглашение, что переменные (x) обозначаются строчными буквами, имена функций (f) — строчными буквами или словами, начинающимися со строчных букв (например, add), а конструкторы (C) — прописными буквами или словами, начинающимися с прописной буквы. Также будем считать, что все имена функций и конструкторов имеют некоторую арность, обозначенную $arity(f)$ или соответственно $arity(C)$. Также будем опускать пустые скобки у конструкторов и функций нулевой арности (C вместо $C()$) и точки с запятой после определений функций и ветвей case-выражений.

Для удобства в язык включена конструкция $\perp$ (“не определено”), но можно считать её синтаксическим сахаром для выражения $\mathbf{case}\ C\ \mathbf{of}\{D \to D\}$ (сопоставление с образцом конструктора, не упомянутого в образцах).

Дополнительно потребуем от определений нашего входного языка, чтобы:

- переменные в левых частях определений и в образцах не повторялись (т.е. в одном списке переменных все переменные разные, но перекрытие переменных во вложенных конструкциях разрешается);
- все переменные были связаны либо левыми частями, либо образцами;
- имена конструкторов в сопоставлениях с образцом не повторялись и ветви были отсортированы по именам конструкторов некоторым образом (главное, чтобы в одной программе использовался один порядок);
- использования имён конструкторов и функций были согласованы с их арностью.

Рис. 1 Синтаксис рассматриваемого языка

Определение:

$def ::= f(\overline{x_i}) \equiv e$

Выражение:

$e ::= x$	переменная
$\mid f(\overline{e_i})$	вызов функции
$\mid C(\overline{e_i})$	конструктор
$\mid$ **case** e_0 **of** $\{\ \overline{C_i(\overline{x_{ij}})\ \rightarrow\ e_i;}\ \}$	сопоставление с образцом
$\mid \bot$	ошибка

Определение 1. *Программой* (на нашем языке) назовём набор определений (как на Рис. 1), такой, что каждой функции f, входящей в программу (т.е. упомянутой в каком-либо из этих определений), соответствует ровно одно определение вида $f(\overline{x_i}) \equiv \ldots$ из этого набора.

Т.е. нельзя, чтобы у функции не было определений, и нельзя, чтобы было больше одного определения.

Теперь введём понятие полипрограммы.

Определение 2. *Полипрограммой* (на нашем языке) назовём набор определений (как на Рис. 1).

Заметим, что любая программа также является и полипрограммой.

В отличие от программ, в полипрограмме разрешены множественные определения одной и той же функции, а также функции без определений. Если рассматривать определение как уравнение (относительно неизвестных-функций), то полипрограмма является системой уравнений, т.е. семантические функции-решения должны удовлетворять всем определениям (более строго мы введём денотационную семантику в следующем разделе).

Возможность задавать функции без определений нужны для того, чтобы любые фрагменты полипрограмм (т.е. полипрограммы-подмножества некоторой другой полипрограммы) были также полипрограммами, например в следующей полипрограмме каждое определение можно рассматривать как отдельную полипрограмму:

$$f(x) \equiv A(g(x))$$
$$g(x) \equiv B(f(x))$$

Но, например, в полипрограмме $g(x) \equiv B(f(x))$, нет определений функции f, хотя она используется в правой части определения g.

Для удобства также введём понятие монопрограммы.

Определение 3. *Монопрограммой* назовём полипрограмму, в которой каждая функция имеет не более одного определения.

Монопрограмма — почти программа, чтобы достроить её до программы, надо всего лишь добавить определения $f(\overline{x_i}) \equiv f(\overline{x_i})$ для функций без определений.

3 Семантика полипрограмм

Множественные определения приводят к некоторым интересным последствиям. Во-первых, некоторые определения могут противоречить друг другу, например, как в этом случае:

$$f \equiv C$$
$$f \equiv D$$

Такие программы мы будем называть *несовместными.*

Во-вторых, некоторые программы, ведущие себя одинаково в операционном смысле и в смысле наименьшей неподвижной точки, могут вести себя по-разному в качестве фрагментов полипрограмм, например, рассмотрим следующие две программы:

$$f(x) \equiv f(x) \qquad \text{и} \qquad f(x) \equiv \bot.$$

(Напомним, что $\bot$ можно выразить как **case** C **of**$\{D \to \ldots\}$). С точки зрения наименьшей неподвижной точки эти программы эквивалентны, однако полипрограмма

$$f(x) \equiv f(x)$$
$$f(x) \equiv C$$

определяет функцию $f(x) = C$. Первое определение говорит, что f равна себе — это тавтология, и данный факт не противоречит второму определению. Однако если заменить первое определение на $f(x) \equiv \bot$, то получится несовместная полипрограмма:

$$f(x) \equiv \bot$$
$$f(x) \equiv C$$

Действительно, если бы данная программа была совместна, то выполнялось бы равенство $\bot = f(x) = C$, но $C \neq \bot$.

Отметим, что такие определения, как $f(x) \equiv f(x)$, могут появиться в полипрограмме в процессе интуитивно понятных преобразований. Иногда они являются не тавтологиями, а более сложными утверждениями. Например, определение $f(x) \equiv f(f(x))$ говорит, что функция f идемпотентна — такое

определение может быть полезно в качестве леммы при преобразовании полипрограммы.

По этим причинам мы отказываемся от стандартного подхода, предписывающего рассматривать наименьшую неподвижную точку [13], и считаем семантикой полипрограммы все возможные неподвижные точки (модели). Тогда семантика несовместной программы — это просто пустое множество моделей. Кроме того, этот подход делает семантику композиционной: семантика полипрограммы (множество моделей) является пересечением семантик её определений. В частности, это позволяет заменять фрагменты полипрограммы на эквивалентные им полипрограммы с сохранением семантики всей полипрограммы. Замена же определения $f(x) \equiv f(f(x))$ на $f(x) \equiv \bot$ является некорректной, потому что для первого определения моделями являются все идемпотентные функции, а у второго определения есть всего одна модель, приписывающая функции f значение $\bot$.

Теперь введём денотационную семантику полипрограмм более формально. Аналогично логике предикатов первого порядка мы введём понятие интерпретации и модели, однако интерпретации будут задавать только значения пользовательских функций (функциональных символов) нашей полипрограммы, смысл же конструкций языка будет фиксирован.

Интерпретация отображает каждый функциональный символ арности n из полипрограммы в функцию типа $\mathbb{A}^n \to \mathbb{A}$, где $\mathbb{A}$ — множество данных первого порядка, заданное как наибольшая неподвижная точка следующего определения:

$$\mathbb{A} = \{C(\overline{a_i}) \mid a_i \in \mathbb{A},\ C \text{ — конструктор}\} \cup \{\bot\}.$$

Таким образом, $\mathbb{A}$ является множеством конечных и бесконечных деревьев, построенных из конструкторов и $\bot$.

На $\mathbb{A}$ естественным образом вводится частичный порядок.

Определение 4. $a \sqsubseteq b$ для $a, b \in \mathbb{A}$ определяется как наименьшее отношение, удовлетворяющее следующим утверждениям:

$$\bot \sqsubseteq a$$
$$\overline{a_i \sqsubseteq b_i} \Rightarrow C(\overline{a_i}) \sqsubseteq C(\overline{b_i})$$

Более того, $\mathbb{A}$ — полурешётка с операцией взятия точной нижней грани $\sqcap$. Но это не решётка, т.к. операция взятия точной верхней грани $\sqcup$ является частичной.

Определение 5. Пусть A — частично упорядоченное множество. Тогда A^n — частично упорядоченное множество кортежей, причём

$$(a_1, \ldots, a_n) \sqsubseteq (b_1, \ldots, b_n) \Leftrightarrow \forall i\ a_i \sqsubseteq b_i.$$

Стандартным образом вводится понятие наименьшей верхней грани и понятие непрерывности по Скотту.

Определение 6. Пусть A — частично упорядоченное множество (в частности, $\mathbb{A}$). *Верхней гранью* множества $X \subseteq A$ называется элемент $a \in A$, такой, что $\forall x \in X \Rightarrow x \sqsubseteq a$. *Наименьшей верхней гранью* (супремумом) множества X называется такая верхняя грань a, что для любой другой верхней грани a' выполняется $a \sqsubseteq a'$. Обозначение: $a = \sup X$.

Определение 7. Пусть A, B — частично упорядоченные множества. Функция $f : A \to B$ является *непрерывной*, если для любой монотонной последовательности $a_1 \sqsubseteq a_2 \sqsubseteq \ldots$ элементов из A, такой, что $\sup\{a_i\} = a$ верно, что $f(a) = \sup\{f(a_i)\}$.

Через $\mathbb{D}$ обозначим множество всех непрерывных функций над $\mathbb{A}$ конечной арности, т.е. $\mathbb{D} = \bigcup_n \mathbb{A}^n \to \mathbb{A}$.

Определение 8. Пусть P — полипрограмма, а F — множество функциональных символов данной полипрограммы (имён функций, упомянутых в ней). *Интерпретацией* полипрограммы P будем называть функцию $\eta : F \to \mathbb{D}$, такую, что $arity(\eta(f)) = arity(f)$.

Моделью полипрограммы P будем называть такую интерпретацию μ, что $\mu \models P$, где $\mu \models P \stackrel{\text{def}}{=} \bigwedge_{d \in P} [\![d]\!]_\mu$, т.е. в μ выполняются все определения d полипрограммы P. $[\![d]\!]_\mu$ определено на Рис. 2.

Таким образом, если все определения полипрограммы истинны в некоторой интерпретации, то такая интерпретация называется моделью полипрограммы. Нам также понадобится понятие семантической эквивалентности полипрограмм.

Определение 9. Две полипрограммы P_1 и P_2 со множествами функциональных символов F_1 и F_2 будем называть *семантически эквивалентными* на подмножестве функциональных символов $F \subseteq F_1 \cap F_2$, если для любой модели $\mu_1 \models P_1$ существует модель $\mu_2 \models P_2$, такая, что $\forall f \in F \,.\, \mu_1(f) = \mu_2(f)$, и наоборот, для любой модели $\eta_2 \models P_2$ существует модель $\eta_1 \models P_1$, такая, что $\forall f \in F \,.\, \eta_1(f) = \eta_2(f)$.

Рассмотрим теперь некоторые особенности введённой таким образом семантики полипрограмм.

Во-первых, как уже было сказано, бывает так, что две полипрограммы (даже программы) имеют одинаковые наименьшие неподвижные точки (модели), однако разное множество всех неподвижных точек (моделей), и поэтому мы не считаем их эквивалентными, например, полипрограмма

$$f(x) \equiv f(f(x))$$

обладает такими моделями, что $\mu(f)$ — идемпотентная функция, а для полипрограммы

$$f(x) \equiv f(x)$$

Рис. 2 Семантика определений

$$(\!|f(\overline{e_j})|\!)_{\mu\sigma} = \mu(f)(\overline{(\!|e_j|\!)_{\mu\sigma}})$$
$$(\!|x|\!)_{\mu\sigma} = \sigma(x)$$
$$(\!|C(\overline{e_j})|\!)_{\mu\sigma} = C(\overline{(\!|e_j|\!)_{\mu\sigma}})$$
$$(\!|\bot|\!)_{\mu\sigma} = \bot$$
$$(\!|\textbf{case } e_0 \textbf{ of } \{\overline{C_j(\overline{x_i}) \to e_j}\}|\!)_{\mu\sigma} =$$
$$= \begin{cases} (\!|e_j|\!)_{\mu\tau} & \text{если } (\!|e_0|\!)_{\mu\sigma} = C_j(\overline{a_i}) \\ & \text{где } \tau(y) = a_i, \text{ если } y = x_i, \text{ и } \tau(y) = \sigma(y) \text{ иначе} \\ \bot & \text{если } (\!|e_0|\!)_{\mu\sigma} = C_l(\ldots) \neq C_j(\ldots) \forall j \\ \bot & \text{если } (\!|e_0|\!)_{\mu\sigma} = \bot \end{cases}$$

$$[\![t_1 \equiv t_2]\!]_\mu \Leftrightarrow \forall \sigma : V \to \mathbb{A} \;.\; (\!|t_1|\!)_{\mu\sigma} = (\!|t_2|\!)_{\mu\sigma}$$

(σ — контекст, отображающий имена переменных из V в значения)

вообще любая интерпретация является моделью.

Отметим, что полипрограмма, в отличие от программы, может не иметь наименьшей модели, даже если полипрограмма совместна. Например, следующая полипрограмма не допускает, чтобы g было равно $\bot$:

$$f \equiv \textbf{case } g \textbf{ of} \{A \to C; B \to C\}$$
$$f \equiv C$$

Все её модели — это $\{f \mapsto C, g \mapsto A\}$ и $\{f \mapsto C, g \mapsto B\}$, однако они несравнимы.

На практике рассмотрение всех неподвижных точек вместо наименьшей означает, что мы теряем некоторую полноту, например, нельзя преобразовать программу $f(x) \equiv f(f(x))$ в программу $f(x) \equiv f(x)$, поскольку они имеют разные множества моделей (хотя их наименьшие модели совпадают). Однако мы не теряем корректность, потому что если все модели совпадают, то наименьшие тем более. Вопрос о том, насколько серьёзным является данное ограничение на практике и как его обойти (например, введя другую семантику или используя предварительную обработку программы), оставлен для будущих исследований.

Интересно, что можно записать незавершающуюся программу (в операционном смысле, т.е. не просто возвращающую ошибку, а именно зацикливающуюся) так, чтобы она имела единственную модель как полипрограмма,

например, как в этом случае:

$$\begin{aligned}&\inf \equiv A(\inf)\\&\mathrm{eat}(x) \equiv \textbf{case } x \textbf{ of } A(x') \ \rightarrow \ \mathrm{eat}(x')\\&\mathrm{bottom} \equiv \mathrm{eat}(\inf)\end{aligned}$$

Здесь inf является бесконечной цепочкой $A(A(A(\ldots)))$, а функция eat просто разбирает свой аргумент, пока не упрётся в $\bot$ или в незнакомый конструктор. Значение функции bottom будет равно $\bot$ во всех моделях. Действительно, т.к. функция $\mu(\mathrm{eat})$ должна быть непрерывной, а на всех значениях вида $S^n(\bot)$ она равна $\bot$ по определению конструкций языка, то и на бесконечном $S(S(\ldots))$ она также равна $\bot$. Здесь крайне важно, что в качестве интерпретаций функциональных символов допускаются только непрерывные функции. Альтернативный способ доказательства единственности модели данной программы — использование признаков структурной и защищённой рекурсии, которые мы обсудим несколько позже.

4 Полипрограммы как множества программ

Если в полипрограмме для каждой функции, имеющей хотя бы одно определение, выбрать одно определение, то получится монопрограмма. Таким образом, наивно полипрограмму можно считать представлением множества монопрограмм. Однако такое наивное определение множества монопрограмм, представимых полипрограммой, было бы не очень удачным, поскольку среди таких монопрограмм могут быть монопрограммы, не эквивалентные исходной полипрограмме. Поэтому добавим ещё ограничение, что монопрограмма должна быть семантически эквивалентна исходной полипрограмме. Кроме того, будем учитывать, что нас могут интересовать не все функции в полипрограмме.

Определение 10. Будем говорить, что для некоторого набора функциональных символов F полипрограмма P *представляет* множество монопрограмм M, таких, что

- $M \subseteq P$ (M — фрагмент P);
- M семантически эквивалентна P на функциях из F.

Задачу поиска монопрограммы, принадлежащей множеству, представляемому полипрограммой, будем называть *выделением* монопрограммы из полипрограммы.

Пример 1. Рассмотрим следующую полипрограмму:

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case } x \textbf{ of } \{\ T \ \to\ F;\ F \ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \textbf{case } x \textbf{ of } \{\ Z \ \to\ T;\ S(y) \ \to\ \mathrm{odd}(y)\ \}\\
&\mathrm{even}(x) \equiv \mathrm{not}(\mathrm{odd}(x))\\
&\mathrm{odd}(x) \equiv \textbf{case } x \textbf{ of } \{\ Z \ \to\ F;\ S(y) \ \to\ \mathrm{even}(y)\ \}\\
&\mathrm{odd}(x) \equiv \mathrm{not}(\mathrm{even}(x))
\end{aligned}$$

Для простоты будем считать, что нас интересуют все функции полипрограммы, т.е. $F = \{\mathrm{even}, \mathrm{odd}, \mathrm{not}\}$. Мы можем выделить из этой полипрограммы четыре монопрограммы, выбирая по одному определению для каждой функции. Три из них будут эквивалентны исходной полипрограмме:

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case } x \textbf{ of } \{\ T \ \to\ F;\ F \ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \textbf{case } x \textbf{ of } \{\ Z \ \to\ T;\ S(y) \ \to\ \mathrm{odd}(y)\ \}\\
&\mathrm{odd}(x) \equiv \textbf{case } x \textbf{ of } \{\ Z \ \to\ F;\ S(y) \ \to\ \mathrm{even}(y)\ \}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case } x \textbf{ of } \{\ T \ \to\ F;\ F \ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \textbf{case } x \textbf{ of } \{\ Z \ \to\ T;\ S(y) \ \to\ \mathrm{odd}(y)\ \}\\
&\mathrm{odd}(x) \equiv \mathrm{not}(\mathrm{even}(x))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case } x \textbf{ of } \{\ T \ \to\ F;\ F \ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \mathrm{not}(\mathrm{odd}(x))\\
&\mathrm{odd}(x) \equiv \textbf{case } x \textbf{ of } \{\ Z \ \to\ F;\ S(y) \ \to\ \mathrm{even}(y)\ \}
\end{aligned}$$

Но одна не будет эквивалентна исходной полипрограмме (даже наименьшие неподвижные точки у них различны):

$$\begin{aligned}
&\mathrm{not}(x) \equiv \textbf{case } x \textbf{ of } \{\ T \ \to\ F;\ F \ \to\ T\ \}\\
&\mathrm{even}(x) \equiv \mathrm{not}(\mathrm{odd}(x))\\
&\mathrm{odd}(x) \equiv \mathrm{not}(\mathrm{even}(x))
\end{aligned}$$

Поэтому исходная полипрограмма представляет множество монопрограмм, состоящее из трёх вышеуказанных монопрограмм, не включающее четвёртую. $\triangle$

Отметим, что не всегда из полипрограммы можно выделить монопрограмму, даже если полипрограмма имеет наименьшую модель.

Пример 2. Бывают определения, которые не являются самодостаточными и могут работать только совместно, например:

$$\begin{aligned}
&f(x) \equiv \textbf{case } x \textbf{ of } \{\ A \ \to\ A;\ B \ \to\ f(x)\ \}\\
&f(x) \equiv \textbf{case } x \textbf{ of } \{\ A \ \to\ f(x);\ B \ \to\ B\ \}
\end{aligned}$$

Здесь эти два определения вместе однозначно задают функцию, такую, что $f(A) = A, f(B) = B$, но по отдельности они задают её только на какой-то своей части возможных входных данных (например, если использовать только первое определение, то мы знаем, что $f(A) = A$, а про $f(B)$ мы знаем только то, что оно равно самому себе — обычно в этом случае полагают, что $f(B) = \bot$). Если полипрограмма состоит только из этих двух определений, то из неё невозможно выделить программу, эквивалентную всей полипрограмме, и являющейся её фрагментом. Отметим, впрочем, что можно "развернуть" это определение, введя дополнительную функцию, однако такая программа не будет фрагментом исходной полипрограммы:

$$f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ A\ \to\ A;\ B\ \to\ g(x)\ \}$$
$$g(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{\ A\ \to\ f(x);\ B\ \to\ B\ \}$$

$\triangle$

Задача доказательства эквивалентности полипрограммы и выделенной из неё монопрограммы алгоритмически неразрешима, поэтому на практике мы обычно не можем построить всё множество монопрограмм, представимых полипрограммой. Однако можно приблизить его снизу, используя алгоритмически разрешимые признаки этой эквивалентности.

Один из простейших способов обеспечить эквивалентность — обеспечить единственность модели как у полипрограммы, так и у выделяемой из неё монопрограммы. Точнее, достаточно потребовать, чтобы у полипрограммы была хотя бы одна модель (это легко выполняется, если полипрограмма получается из обычной программы в результате преобразований), а у выделяемой монопрограммы было не более одной модели. В этом случае полипрограмма и монопрограмма будут эквивалентны на функциях монопрограммы. Таким образом, нам нужны лишь алгоритмически разрешимые признаки того, что монопрограмма имеет не более одной модели. Т.к. монопрограмма всегда имеет хотя бы одну модель, будем называть их просто признаками единственности. Отметим, что для некоторых применений, например для слияния по бисимуляции, которое мы рассмотрим позже, нужна не абсолютная единственность модели, а единственность модели для каждой интерпретации функций без определений.

В качестве признака единственности модели можно использовать признаки структурной и защищённой рекурсии [1; 2]. Обычно они используются для обеспечения завершаемости и продуктивности в тотальных языках программирования, однако их можно применять в нетотальном языке для обеспечения единственности модели (при этом программа с такой моделью не обязательно будет завершающейся или продуктивной). Доказательство этого факта мы оставим в качестве будущей работы, здесь приведём лишь упрощённую формулировку признаков.

Нам понадобится следующее отношение порядка на значениях из $\mathbb{A}$.

Определение 11. Пусть $a, b \in \mathbb{A}$. Тогда $a < b \Leftrightarrow (a \leq b \wedge a \neq b)$, а отношение $\leq$ определено как наименьшее отношение, удовлетворяющее следующим

утверждениям:

$a \leq a$;
$a \leq b \Rightarrow a \leq C_j(\ldots, b, \ldots)$;
$a \sqsubseteq b \Rightarrow a \leq b$;
$a \leq b, b \leq c \Rightarrow a \leq c$.

Пусть определение функции f имеет вид $f(\overline{x_i}) \equiv E\langle f(\overline{e_{1i}}), \ldots, f(\overline{e_{ni}})\rangle$, и в данной полипрограмме функция f не имеет других рекурсивных вызовов (в том числе косвенных). Тогда:

- Если существует j, такое, что для любой интерпретации μ и любого контекста σ, назначающего переменным конечные (но, возможно, частичные) значения, верно, что $(\!|E\langle\overline{f(\overline{e_{li}})}\rangle|\!)_{\mu\sigma}$ имеет вид $\mathcal{E}\langle\overline{\mu(f)(\overline{a_{li}})}\rangle$, и для любого l верно $a_{lj} < \sigma(x_j)$, то любые две модели данной полипрограммы, совпадающие на всех функциях, входящих в E, кроме f, совпадают и на f (т.е. $\mu_1(f) = \mu_2(f)$).

- Если для любой интерпретации μ и любого контекста σ верно, что значение $(\!|E\langle\overline{f(\overline{e_{li}})}\rangle|\!)_{\mu\sigma}$ имеет вид $C_{\mu\sigma}(\ldots)$, где $C_{\mu\sigma}$ — некоторый конструктор или $\bot$, то любые две модели данной полипрограммы, совпадающие на всех функциях, входящих в E, кроме f, совпадают и на f (т.е. $\mu_1(f) = \mu_2(f)$).

Первый случай соответствует структурной рекурсии, а второй — защищённой рекурсии, однако мы не будем формально вводить эти термины и будем использовать их исключительно неформально.

Пример 3. Снова рассмотрим следующую полипрограмму:

$\mathrm{not}(x) \equiv \mathbf{case}\ x\ \mathbf{of}\ \{\ T\ \to\ F;\ F\ \to\ T\ \}$
$\mathrm{even}(x) \equiv \mathbf{case}\ x\ \mathbf{of}\ \{\ Z\ \to\ T;\ S(y)\ \to\ \mathrm{odd}(y)\ \}$
$\mathrm{even}(x) \equiv \mathrm{not}(\mathrm{odd}(x))$
$\mathrm{odd}(x) \equiv \mathbf{case}\ x\ \mathbf{of}\ \{\ Z\ \to\ F;\ S(y)\ \to\ \mathrm{even}(y)\ \}$
$\mathrm{odd}(x) \equiv \mathrm{not}(\mathrm{even}(x))$

Возьмём следующую монопрограмму:

$\mathrm{not}(x) \equiv \mathbf{case}\ x\ \mathbf{of}\ \{\ T\ \to\ F;\ F\ \to\ T\ \}$
$\mathrm{even}(x) \equiv \mathbf{case}\ x\ \mathbf{of}\ \{\ Z\ \to\ T;\ S(y)\ \to\ \mathrm{odd}(y)\ \}$
$\mathrm{odd}(x) \equiv \mathrm{not}(\mathrm{even}(x))$

Рассмотрим определение функции even. Раскроем вызов функции odd, чтобы избавиться от косвенных рекурсивных вызовов:

$\mathrm{even}(x) \equiv \mathbf{case}\ x\ \mathbf{of}\ \{\ Z\ \to\ T;\ S(y)\ \to\ \mathrm{not}(\mathrm{even}(y))\ \}.$

Если обозначить правую часть за E, то

$$(\!|E|\!)_{\mu\sigma} = \textbf{case}\ \sigma(x)\ \textbf{of}\ \{\ Z\ \to\ T;\ S(y)\ \to\ \mu(\text{not})(\mu(\text{even})(y))\ \}.$$

Если $\sigma(x) = Z$, то $(\!|E|\!)_{\mu\sigma} = T$, если же $\sigma(x) = S(a)$, причём a — конечное, то $(\!|E|\!)_{\mu\sigma} = \mu(\text{not})(\mu(\text{even})(a))$, причём $a < S(a) = \sigma(x)$, во всех остальных случаях $(\!|E|\!)_{\mu\sigma} = \bot$. Таким образом, выполняется признак структурной рекурсии, а значит, все модели, совпадающие на функции not, совпадают и на even. Отметим, что крайне важно, что в условии требуется выполнение некоторых неравенств только для конечных аргументов, потому что для бесконечного $b = S(S(S(\ldots)))$ выполняется $b = S(b)$ и не выполняется $b < S(b)$.

Функция not не является рекурсивной, поэтому у её определения ровно одна модель. У функции odd не более одной модели для каждой интерпретации even. Значит и вся монопрограмма имеет не более одной модели.

Значит, если исходная полипрограмма совместна, то рассмотренная монопрограмма принадлежит множеству представляемых ей монопрограмм. $\triangle$

5 Правила преобразования полипрограмм

В соответствии со схемой Бёрстолла-Дарлингтона [3] полипрограммы преобразуются при помощи правил вывода, которые добавляют новые уравнения к уже существующим. В насыщении равенствами [4] происходит то же самое — E-PEG (структура, компактно представляющая множества программ) насыщается равенствами при помощи применения некоторых аксиом. Мы будем записывать правила преобразования полипрограмм в виде $P_1 \mapsto P_2$, где P_1 и P_2 — полипрограммы. Правила могут добавлять определения и функции, а также *удалять* определения (будем считать, что удалять функции правила не могут — эта операция проблемная, потому что удаляемая функция может использоваться в других местах полипрограммы). Записывать правила будем следующим образом:

$$\left\{\begin{array}{l}\text{удаляемое определение}\\ \text{определение}_1\\ \text{определение}_2\end{array}\right\} \mapsto \left\{\begin{array}{l}\text{новое определение}\\ \text{определение}_1\\ \text{определение}_2\end{array}\right\}$$

Применение такого правила к полипрограмме P заключается в замене фрагмента, соответствующего левой части правила (с точностью до переименования функций и переменных), на фрагмент, соответствующий правой части.

Также будем считать очевидным, что если в правиле левая часть семантически эквивалентна правой на функциях из левой части, то исходная полипрограмма также семантически эквивалентна преобразованной на функциях исходной полипрограммы.

Правила, удаляющие определения, мы будем называть *деструктивными*, а не удаляющие — *недеструктивными*. Отметим, что возможность удалять определения является оптимизацией, потому что каждое деструктивное пра-

вило можно переформулировать в недеструктивном виде, и полипрограмма, полученная в результате применения деструктивных правил, будет подмножеством полипрограммы, полученной в результате применения аналогичных недеструктивных правил. Т.е. недеструктивные правила могут дать больше определений. Однако деструктивные правила необходимы для того, чтобы избежать раннего наступления комбинаторного взрыва. В схеме Бёрстолла-Дарлингтона правила вывода не удаляют определения. В насыщении равенствами Тейта и др. аксиомы также недеструктивны, однако в случае насыщения равенствами некоторые базовые операции над E-PEG, не считающиеся деструктивными, соответствуют деструктивным правилам преобразования полипрограмм.

Далее мы приведём две системы правил преобразования полипрограмм: первая будет очень близка к исходной схеме Бёрстолла-Дарлингтона, а вторая будет моделировать насыщение равенствами в применении к нашему функциональному языку. Мы рассмотрим только самые основные правила (например, мы не будем рассматривать распространение позитивной информации и правила обобщения или абстракции).

5.1 Правила в стиле Бёрстолла-Дарлингтона

Раскрытие вызова функции (unfolding)

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv H \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv H \\ f(\overline{x_i}) \equiv E\langle H\{\overline{y_j \mapsto e_j}\}\rangle \end{array}\right\}$$

Данное правило подставляет тело функции в место вызова (одновременно подставляя выражения-аргументы в тело). Оно достаточно стандартно. Мы будем считать, что процесс подстановки интуитивно понятен.

Свёртка (folding)

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv H\langle E\{\overline{y_j \mapsto z_j}\}\rangle \\ g(\overline{y_j}) \equiv E \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv H\langle E\{\overline{y_j \mapsto z_j}\}\rangle \\ g(\overline{y_j}) \equiv E \\ f(\overline{x_i}) \equiv H\langle g(\overline{z_j})\rangle \end{array}\right\}$$

Данное правило обратно предыдущему и говорит, что правую часть определения можно заменить на левую, т.е. вместо выражения вставить вызов соответствующей функции. Данная формулировка менее мощная, чем в оригинальной работе Бёрстолла и Дарлингтона, потому что допускает замещение переменных выражения E не произвольными выражениями, а только переменными.

Далее идут менее важные правила, относящиеся к семантике нашего языка.

Редукция сопоставления с образцом

$$\left\{ f(\overline{x_i}) \equiv H\langle \textbf{case}\ C(\overline{e_j})\ \textbf{of}\ \{\ldots; C(\overline{y_j}) \to E; \ldots\}\rangle \right\} \mapsto \\ \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv H\langle \textbf{case}\ C(\overline{e_j})\ \textbf{of}\ \{\ldots; C(\overline{y_j}) \to E; \ldots\}\rangle \\ f(\overline{x_i}) \equiv H\langle E\{\overline{y_j \mapsto e_j}\}\rangle \end{array} \right\}$$

Сопоставление с образцом результата сопоставления с образцом

$$\left\{ f(\overline{x_i}) \equiv H\langle \textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to E_k}\})\ \textbf{of}\ \{\ldots\}\rangle \right\} \mapsto \\ \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv H\langle \textbf{case}\ (\textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to E_k}\})\ \textbf{of}\ \{\ldots\}\rangle \\ f(\overline{x_i}) \equiv H\langle \textbf{case}\ e\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to \textbf{case}\ E_k\ \textbf{of}\ \{\ldots\}}\}\rangle \end{array} \right\}$$

Перед применением данного правила переменные переименовываются так, чтобы не было конфликтов имён.

Пример 4. Рассмотрим классический пример доказательства того, что сложение ассоциативно

$$(1)\ \mathrm{add}(x, y) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to y; S(x') \to S(\mathrm{add}(x', y))\}$$
$$(2)\ f(x, y, z) \equiv \mathrm{add}(\mathrm{add}(x, y), z)$$
$$(3)\ g(x, y, z) \equiv \mathrm{add}(x, \mathrm{add}(y, z))$$

В этой полипрограмме определены функции f и g, и нужно доказать, что они эквивалентны. Начнём с преобразования функции f. Сначала раскроем внешний вызов add в определении (2):

$$(4)\ f(x, y, z) \equiv \\ \quad \textbf{case}\ \mathrm{add}(x, y)\ \textbf{of} \\ \quad\quad Z \to z \\ \quad\quad S(x') \to S(\mathrm{add}(x', z))$$

Теперь раскроем add в разбираемой позиции в определении (4):

$$(5)\ f(x, y, z) \equiv \\ \quad \textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{Z \to y; S(x') \to S(\mathrm{add}(x', y))\})\ \textbf{of} \\ \quad\quad Z \to z \\ \quad\quad S(x') \to S(\mathrm{add}(x', z))$$

Теперь применим сопоставление с образцом результата сопоставления с образ-

цом:

$$
\begin{array}{l}
(6)\, f(x,y,z) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\quad\quad Z \to \textbf{case}\ y\ \textbf{of}\{Z \to z; S(x') \to S(\mathrm{add}(x', z))\} \\
\quad\quad S(x') \to \textbf{case}\ S(\mathrm{add}(x', y))\ \textbf{of}\{Z \to z; S(x'') \to S(\mathrm{add}(x'', z))\}
\end{array}
$$

Теперь в определении (6) в ветви Z произведём свёртку, а в ветви S — редукцию сопоставления с образцом:

$$
\begin{array}{l}
(7)\, f(x,y,z) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\quad\quad Z \to \mathrm{add}(y, z) \\
\quad\quad S(x') \to S(\mathrm{add}(\mathrm{add}(x', y), z))
\end{array}
$$

Теперь уже в ветви S применим свёртку

$$
\begin{array}{l}
(8)\, f(x,y,z) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\quad\quad Z \to \mathrm{add}(y, z) \\
\quad\quad S(x') \to S(f(x', y, z))
\end{array}
$$

Теперь проделаем примерно то же самое с функцией g. Раскрываем внешний вызов add в определении (3):

$$
\begin{array}{l}
(9)\, g(x,y,z) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\quad\quad Z \to \mathrm{add}(y, z) \\
\quad\quad S(x') \to S(\mathrm{add}(x', \mathrm{add}(y, z)))
\end{array}
$$

Теперь производим свёртку в ветви S:

$$
\begin{array}{l}
(10)\, g(x,y,z) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\quad\quad Z \to \mathrm{add}(y, z) \\
\quad\quad S(x') \to S(g(x', y, z))
\end{array}
$$

Можно заметить, что определения (8) и (10) совпадают с точностью до переименования функций f на g, однако свёртка к ним неприменима. Чтобы вывести отсюда $f \equiv g$, нужно ещё одно преобразование, которое выводит равенство функций исходя из того, что они имеют одинаковые рекурсивные определения. Это преобразование мы рассмотрим в Разделе 6. $\triangle$

5.2 Правила в стиле насыщения равенствами

Насыщение равенствами заключается в применении недеструктивных преобразований к структуре данных, компактно описывающей целое множество программ, причём порядок применения правил не контролируется эвристиками и производится в ширину. Если применять таким образом правила из предыдущего раздела, то полипрограмма будет расти слишком быстро, причём в полипрограмме будет огромное количество дублирования. Действительно, рассмотрим следующую полипрограмму:

$$f(\overline{x_i}) \equiv E_1\langle H\rangle$$
$$g(\overline{x_i}) \equiv E_2\langle H\rangle$$

Выражение H (которое может быть довольно большим) входит в два определения как подвыражение, поэтому занимает в два раза больше памяти, чем могло бы, и, что хуже, в процессе преобразования, вероятно, будет преобразовано одним и тем же образом два раза. Чтобы устранить это дублирование, нужно выделить это подвыражение в отдельную вспомогательную функцию. Для этого нам потребуется правило расчленения, которое выносит любое подвыражение в качестве отдельной функции.

Расчленение

$$\left\{ f(\overline{x_i}) \equiv E\langle e'\rangle \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{fv}(e'))\rangle \\ g(\overline{fv}(e')) \equiv e' \end{array} \right\}$$

Кроме правила расчленения для устранения дублирования нам понадобятся ещё несколько правил, которые мы обсудим чуть позже. Все вместе они позволят преобразовать указанную выше полипрограмму к такому виду:

$$f(\overline{x_i}) \equiv E_1\langle h(\overline{fv}(H))\rangle$$
$$g(\overline{x_i}) \equiv E_2\langle h(\overline{fv}(H))\rangle$$
$$h(\overline{fv}(H)) \equiv H$$

Отсюда возникает следующая идея: выносить таким образом не только повторяющиеся подвыражения, но и вообще все подвыражения, кроме подвыражений вида $f(\overline{x_i})$, где x_i — различные переменные (такие подвыражения будем называть *элементарными вызовами функции*). Будем говорить, что если в полипрограмме вынесены все такие подвыражения, то она находится в расчленённой форме (или просто является расчленённой полипрограммой).

Определение 12. *Полипрограммой в расчленённой форме* назовём полипрограмму, в которой каждое определение имеет вид $f(\overline{x_i}) \equiv s$, где s — простейшее выражение, как на Рис. 3.

Правая часть любого определения полипрограммы в расчленённой форме состоит из элементарного вызова (обозначено r) или простейшего выражения

Рис. 3 Простейшие выражения

Элементарный вызов функции:

$r ::= f(\overline{x_{k_i}})$, где x_{k_i} — *различные* переменные

Простейшее выражение:

$$
\begin{aligned}
s ::= {} & r \\
| {} & x \\
| {} & f(\overline{r_i}) \\
| {} & C(\overline{r_i}) \\
| {} & \textbf{case } r_0 \textbf{ of } \{\ \overline{C_i(\overline{x_{ij}}) \ \rightarrow \ r_i;}\ \} \\
| {} & \bot
\end{aligned}
$$

(обозначено s), т.е. выражения, максимальные собственные подвыражения которого являются элементарными вызовами. Например $f(x, y)$ — элементарный вызов, а $f(x, x)$ — нет, т.к. дублируется переменная, $f(x, C())$ — тоже нет, т.к. второй аргумент — не переменная. Выражение

$$\textbf{case } x \textbf{ of } \{\ C(y) \ \rightarrow \ f(x, y)\ \}$$

не является простейшим выражением, т.к. его подвыражение x в разбираемой позиции не является элементарным вызовом, а выражение

$$\textbf{case } id(x) \textbf{ of } \{\ C(y) \ \rightarrow \ f(x, y)\ \}$$

уже является простейшим.

Любую полипрограмму можно привести к расчленённой форме за конечное число шагов.

Расчленённая форма интересна тем, что она приближает понятие полипрограммы к структуре данных, представляющей множества программ в насыщении равенствами (E-PEG). Действительно, E-PEG — это граф специального вида, вершины которого разбиты на классы эквивалентности. Вершины E-PEG представляют выражения как в AST, т.е. в вершине стоит метка, обозначающая конструкцию языка, а исходящие рёбра идут к вершинам, представляющим подвыражения. Если две вершины E-PEG находятся в одном классе эквивалентности, то это значит, что соответствующие выражения семантически эквивалентны. Упрощённо говоря, выделение программы из E-PEG происходит путём выбора из каждого класса эквивалентности по одной вершине.

Полипрограмма в расчленённой форме соответствует E-PEG в следующем смысле: вершины E-PEG соответствуют определениям расчленённой полипрограммы, а классы эквивалентности вершин E-PEG соответствуют функциям полипрограммы. Выделение монопрограммы из полипрограммы при этом пол-

ностью аналогично выделению программы из E-PEG. Таким образом, расчленённая полипрограмма и E-PEG оказываются очень близкими понятиями.

Правила преобразования полипрограмм, сформулированные в предыдущем подразделе, не подходят для расчленённой формы. Дело в том, что они зачастую просто не могут быть применены, потому что их левые части не находятся в расчленённой форме. Например, левая часть правила редукции сопоставления с образцом выглядит следующим образом:

$$\left\{ f(\overline{x_i}) \equiv H\langle \textbf{case } C(\overline{e_j}) \textbf{ of } \{\ldots; C(\overline{y_j}) \to E; \ldots\}\rangle \right\} \mapsto \ldots$$

Можно было бы перед применением таких правил временно раскрывать вызов функции, однако есть более изящное решение — переписать все правила так, чтобы обе части находились в расчленённой форме. У правила редукции сопоставления с образцом, например, левая часть будет выглядеть следующим образом:

$$\left\{ \begin{array}{l} f(\overline{x_i}) \equiv \textbf{case } g(\overline{z_l}) \textbf{ of } \{\ldots; C(\overline{y_j}) \to h(\overline{y_j}, \overline{x_i}); \ldots\} \\ g(\overline{z_l}) \equiv C(\overline{e_j(\overline{z_{jk}})}) \end{array} \right\} \mapsto \ldots$$

Обратите внимание, что неизвестное подвыражение E было заменено на элементарный вызов неизвестной функции h, потому что в расчленённой форме подвыражения могут иметь только такой вид. Кроме того, отпала необходимость в выражении-контексте H, потому что в расчленённой форме оно не может быть нетривиальным.

Если обе части всех правил находятся в расчленённой форме, то расчленённость будет поддерживаться автоматически и правило расчленения будет нужно только в самом начале для преобразования из программы в расчленённую полипрограмму.

Теперь рассмотрим преобразования, подходящие для расчленённой формы.

Транзитивность

$$\left\{ \begin{array}{l} f(\overline{x_i}) \equiv E \\ g(\overline{y_j}) \equiv E \end{array} \right\} \mapsto \left\{ \begin{array}{l} f(\overline{x_i}) \equiv E \\ g(\overline{y_j}) \equiv E \\ f(\overline{x_i}) \equiv g(\overline{y_j}) \end{array} \right\}$$

Это правило является аналогом свёртки из предыдущего раздела. Фактически оно позволяет вывести определение, обозначающее равенство функций исходя из того, что у этих функций есть определения с одинаковыми правыми частями.

Конгруэнтность

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle g(\overline{e_j})\rangle \\ g(\overline{y_j}) \equiv h(\overline{y_{\theta(j)}}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv E\langle h(\overline{e_{\theta(j)}})\rangle \\ g(\overline{y_j}) \equiv h(\overline{y_{\theta(j)}}) \end{array}\right\}$$

$$\left\{\begin{array}{l} g(\overline{x_i}) \equiv E \\ g(\overline{x_i}) \equiv h(\overline{x_{\theta(i)}}) \end{array}\right\} \mapsto \left\{\begin{array}{l} h(\overline{x_{\theta(i)}}) \equiv E \\ g(\overline{x_i}) \equiv h(\overline{x_{\theta(i)}}) \end{array}\right\}$$

Очень важное правило, которое позволяет распространить информацию о равенстве функций путём замены вхождений одного функционального символа на другой. Замену можно производить как слева, так и справа. Т.к. в равенстве двух функций переменные могут идти в разном порядке, правило параметризовано перестановкой θ. Данное правило следует применять сразу ко всем определениям, в которых встречается функция слева — тогда можно старую функцию вообще больше не рассматривать в дальнейших преобразованиях, считая её имя просто ссылкой на новую функцию.

По своей природе данное правило является частным случаем раскрытия вызова функции из предыдущего раздела.

Удаление дубликатов

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv E \\ f(\overline{x_i}) \equiv E \end{array}\right\} \mapsto \left\{ f(\overline{x_i}) \equiv E \right\}$$

Данное правило просто является оптимизацией, помогающей бороться с комбинаторным взрывом.

Вышеуказанные правила, работая вместе, позволяют устранять дублирование из расчленённой полипрограммы (фактически, они реализуют аналог конгруэнтного замыкания [9]).

Пример 5. Действительно, снова рассмотрим проблему дублирования подвыражений:

$$f(\overline{x_i}) \equiv E_1\langle H\rangle$$
$$g(\overline{x_i}) \equiv E_2\langle H\rangle$$

После применения расчленения данная полипрограмма примет следующий вид:

(1) $f(\overline{x_i}) \equiv E_1\langle h_1(\overline{y_j})\rangle$
(2) $g(\overline{x_i}) \equiv E_2\langle h_2(\overline{y_j})\rangle$
(3) $h_1(\overline{y_j}) \equiv H$
(4) $h_2(\overline{y_j}) \equiv H$

Теперь применим транзитивность, которая добавит новое определение:

$$(5)\ h_2(\overline{y_j}) \equiv h_1(\overline{y_j})$$

Теперь конгруэнтность, которая изменит определения (2) и (4):

$$(2')\ g(\overline{x_i}) \equiv E_2\langle h_1(\overline{y_j})\rangle$$
$$(4')\ h_1(\overline{y_j}) \equiv H$$

И, наконец, удаление дубликатов приведёт полипрограмму к окончательному виду:

$$f(\overline{x_i}) \equiv E_1\langle h_1(\overline{y_j})\rangle$$
$$g(\overline{x_i}) \equiv E_2\langle h_1(\overline{y_j})\rangle$$
$$h_1(\overline{y_j}) \equiv H$$
$$h_2(\overline{y_j}) \equiv h_1(\overline{y_j})$$

Определение $h_2(\overline{y_j}) \equiv h_1(\overline{y_j})$ можно удалить, потому что функция h_2 нигде не используется, однако для этого нужно отдельное преобразование — сборка мусора [11]. Вопрос сборки мусора находится вне рамок данной работы. $\triangle$

Самое интересное изменение происходит с правилом раскрытия вызова функции. Оно распадается на несколько простых случаев, соответствующих конструкциям языка, используемым в определении вызываемой функции. При этом оказывается, что отдельная операция подстановки в выражение ($E\{x \mapsto e\}$) не нужна, потому что неэлементарные вызовы играют роль явных подстановок, а правила раскрытия вызова функции становятся правилами проталкивания этих явных подстановок вниз.

Раскрытие вызова тождественной функции

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv y_k \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv y_k \\ f(\overline{x_i}) \equiv e_k(\overline{x_i}) \end{array}\right\}$$

Раскрытие вызова функции, состоящей из вызова функции

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv g(\overline{d_k(\overline{y_j})}) \end{array}\right\} \mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}) \\ h(\overline{y_j}) \equiv g(\overline{d_k(\overline{y_j})}) \\ f(\overline{x_i}) \equiv g(\overline{q_k(\overline{x_i})}) \\ \hline q_k(\overline{x_i}) \equiv d_k(\overline{e_j(\overline{x_i})}) \end{array}\right\}$$

Аналогичным образом записываются правила “**раскрытие вызова функции, состоящей из конструктора**” и “**раскрытие вызова функции, со-**

стоящей из сопоставления с образцом" (последнее несколько более громоздко).

Остались правила, описывающие поведение сопоставления с образцом.

Редукция сопоставления с образцом

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ g(\overline{x_i})\ \textbf{of}\ \{\ldots; C(\overline{y_j}) \to h(\overline{y_j}, \overline{x_i}); \ldots\} \\ g(\overline{x_i}) \equiv C(\overline{e_j}(\overline{x_i})) \end{array}\right\} \mapsto$$
$$\mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ g(\overline{x_i})\ \textbf{of}\ \{\ldots; C(\overline{y_j}) \to h(\overline{y_j}, \overline{x_i}); \ldots\} \\ g(\overline{x_i}) \equiv C(\overline{e_j}(\overline{x_i})) \\ f(\overline{x_i}) \equiv h(\overline{e_j(\overline{x_i})}, \overline{x_i}) \end{array}\right\}$$

Сопоставление с образцом результата сопоставления с образцом

$$\left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ g(\overline{x_i})\ \textbf{of}\ \{\ldots\} \\ g(\overline{x_i}) \equiv \textbf{case}\ e(\overline{x_i})\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to e_k(\overline{y_{jk}}, \overline{x_i})}\} \end{array}\right\} \mapsto$$
$$\mapsto \left\{\begin{array}{l} f(\overline{x_i}) \equiv \textbf{case}\ g(\overline{x_i})\ \textbf{of}\ \{\ldots\} \\ g(\overline{x_i}) \equiv \textbf{case}\ e(\overline{x_i})\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to e_k(\overline{y_{jk}}, \overline{x_i})}\} \\ f(\overline{x_i}) \equiv \textbf{case}\ e(\overline{x_i})\ \textbf{of}\ \{\overline{C_k(\overline{y_{jk}}) \to h_k(\overline{y_{jk}}, \overline{x_i})}\} \\ h_k(\overline{y_{jk}}, \overline{x_i}) \equiv \textbf{case}\ e_k(\overline{y_{jk}}, \overline{x_i})\ \textbf{of}\ \{\ldots\} \end{array}\right\}$$

Приведённая система правил может выполнять те же преобразования, что и система правил в стиле Бёрстолла-Дарлингтона из предыдущего подраздела, однако она более удобна для программной реализации и менее удобна для ручного преобразования полипрограмм, поэтому мы не будем рассматривать примеры её применения.

5.2.1 Слияние с точностью до перестановки параметров

Описанная система обладает одной приятной особенностью: благодаря тому, что применение правил происходит с точностью до переименования переменных, в некоторых случаях возможно слияние функций, отличающихся только порядком параметров. Например, рассмотрим следующую полипрограмму:

(1) $f(x, y) \equiv C(h(x), y)$
(2) $g(x, y) \equiv C(h(y), x)$

Правые части этих определений различны, однако транзитивность всё равно к ним применима, потому что правила применяются с точностью до переименования переменных. Действительно, если переименовать переменные в определении (2), то получится $g(y, x) \equiv C(h(x), y)$, и правая часть такого определения будет совпадать с правой частью определения (1), что даёт возможность применить правило транзитивности, получив $g(x, y) \equiv f(y, x)$.

После применения транзитивности, конгруэнтности и удаления дубликатов получится следующая полипрограмма (g становится просто ссылкой на f, и её можно, в принципе, удалить сборкой мусора):

$$f(x, y) \equiv C(h(x), y)$$
$$g(x, y) \equiv f(y, x)$$

Эта возможность полезна для работы с функциями высокой арности, потому что количество возможных вариаций одной и той же функции растёт как $n!$ (если учитывать только порядок параметров).

Описанная особенность является полезным побочным эффектом применения правил с точностью до переименования переменных, и она отсутствует, например, в насыщении равенствами Тейта и др. [4]. Отметим, что похожий приём применяется в [10] для слияния выражений, отличающихся константным целочисленным смещением.

6 Слияние по бисимуляции

При помощи правил раскрытия вызова и свёртки или транзитивности и конгруэнтности можно добиться вывода определения, утверждающего эквивалентность двух функций, представленных одинаковыми *ациклическими* фрагментами полипрограммы. Однако если функции представлены фрагментами, имеющими циклы, эта процедура не работает. Простейший пример — две бесконечные последовательности:

$$f \equiv A(f)$$
$$g \equiv A(g)$$

Этот факт мотивирует поиск некоторого преобразования полипрограмм, которое бы могло сливать два изоморфных фрагмента. Оказывается, однако, что требование изоморфизма можно ослабить до отношения, которое мы назовём бисимуляцией полипрограмм (являющееся обобщением понятия бисимуляции для размеченных систем переходов), поэтому мы будем называть это преобразование слиянием по бисимуляции. В данной работе мы введём урезанное понятие бисимуляции, не учитывающее, что в соответствующих друг другу функциях параметры могут идти в разном порядке. Отметим, что в схеме Бёрстолла-Дарлингтона аналогичное преобразование называется переопределением (redefinition).

В слиянии по бисимуляции возникает та же самая проблема, что и при выделении монопрограмм — не все фрагменты, находящиеся в отношении бисимуляции, корректно сливать, например, следующие две функции входят в два изоморфных фрагмента, состоящих из одного определения каждый:

$$f(x) \equiv f(f(x))$$
$$g(x) \equiv g(g(x))$$

Но эти фрагменты говорят только, что соответствующие функции являются идемпотентными, из чего не следует, что они должны быть равны. Эта проблема решается так же, как и при выделении монопрограмм — на фрагменты накладываются условия, являющиеся признаками единственности модели.

Слияние по бисимуляции является преобразованием, которое соответствует частному случаю применения доказательства по индукции или коиндукции для равенства двух функций. В некоторых случаях ему не хватает мощности, однако в большинстве случаев его достаточно для проведения индуктивных доказательств.

Введём теперь понятие бисимуляции (упрощённое).

Определение 13. Будем говорить, что две полипрограммы P_1 и P_2 со множествами функциональных символов F_1 и F_2 находятся в отношении бисимуляции, если существует третья полипрограмма H со множеством функциональных символов F и два отображения $\phi_i : F \to F_i, i = 1, 2$, таких, что если заменить в H все функциональные символы f на $\phi_i(f)$, то получится P_i с точностью до удаления определений-дубликатов. H будем называть полипрограммой бисимуляции.

Если в полипрограмме P существуют такие полипрограммы-фрагменты P_1 и P_2, находящиеся в отношении бисимуляции, причём H — полипрограмма этой бисимуляции, и если для каждой интерпретации μ_h таких функций h, что $\phi_1(h) = \phi_2(h)$ существует не более одной модели μ полипрограммы H такой, что $\mu(h) = \mu_h$, то в полипрограмму P можно добавить определения вида $\phi_1(f)(\overline{x_i}) \equiv \phi_2(f)(\overline{x_i})$, получив полипрограмму P', эквивалентную исходной полипрограмме P.

Отметим, что требуется не абсолютная единственность модели, а единственность модели для каждой интерпретации функций, которые отображаются в одну и ту же функцию P. Такие функции соответствуют случаю доказательства по рефлексивности и бывают полезны во многих случаях.

Рассмотрим теперь примеры применения слияния по бисимуляции. Начнём с бесконечных последовательностей:

$$f \equiv A(f)$$
$$g \equiv A(g)$$

В качестве полипрограммы бисимуляции возьмём $f \equiv A(f)$, а отображениями функций будут $\{f \mapsto f\}$ и $\{f \mapsto g\}$. Функция f в данной полипрограмме является защищённо-рекурсивной, потому что семантика правой части $(\!|A(f)|\!)_{\mu\sigma}$ имеет вид $A(\mu(f))$. Значит, полипрограмма бисимуляции имеет не более одной модели и можно добавить определение $f \equiv g$ в полипрограмму:

$$f \equiv A(f)$$
$$g \equiv A(g)$$
$$f \equiv g$$

Теперь завершим пример ассоциативности сложения (Пример 4):

$$(8)\ f(x,y,z) \equiv \textbf{case}\ x\ \textbf{of}\{Z \to \mathrm{add}(y,z); S(x') \to S(f(x',y,z))\}$$
$$(10)\ g(x,y,z) \equiv \textbf{case}\ x\ \textbf{of}\{Z \to \mathrm{add}(y,z); S(x') \to S(g(x',y,z))\}$$

В этом случае в качестве полипрограммы бисимуляции можно взять просто первое определение:

$$f(x,y,z) \equiv \textbf{case}\ x\ \textbf{of}\{Z \to \mathrm{add}(y,z); S(x') \to S(f(x',y,z))\}$$

При этом $\phi_1(f) = f$, $\phi_2(f) = g$, $\phi_i(\mathrm{add}) = \mathrm{add}$. Рассмотрим семантику правой части:

$$(\!|\textbf{case}\ x\ \textbf{of}\{Z \to \mathrm{add}(y,z); S(x') \to S(f(x',y,z))\}|\!)_{\mu\sigma} =$$
$$= \begin{cases} \mu(\mathrm{add})(\sigma(y),\sigma(z)) & \text{если } \sigma(x) = Z \\ \mu(f)(a,\sigma(y),\sigma(z)) & \text{если } \sigma(x) = S(a) \\ \bot & \text{иначе} \end{cases}$$

Т.к. $a < \sigma(x)$, если $\sigma(x) = S(a)$ и $S(a)$ конечное, то f структурно-рекурсивная. Поэтому для каждой интерпретации add существует не более одной интерпретации f, такой, что вместе они образуют модель. Поэтому можно добавить определение

$$(11)\ f(x,y,z) \equiv g(x,y,z)$$

Пример 6. Слияние по бисимуляции обычно применяется для завершения доказательства, однако иногда его имеет смысл применять и в середине процесса преобразования полипрограммы, что соответствует обнаружению и применению леммы. Рассмотрим следующую полипрограмму:

$$(1)\ f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to f(f(y))\}$$
$$(2)\ g(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to g(x)\}$$

Требуется доказать, что f и g эквивалентны. Для удобства вынесем подвыражение $f(f(y))$ в отдельную функцию при помощи правила расчленения:

$$(3)\ f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}$$
$$(4)\ h(x) \equiv f(f(x))$$

Раскроем внешний вызов f в определении (4), получится следующее определение:

$$(5)\ h(x) \equiv \textbf{case}\ f(x)\ \textbf{of}\{Z \to Z; S(y) \to h(y)\}$$

Теперь раскроем $f(x)$ в разбираемой позиции:

$$
\begin{array}{l}
(6)\, h(x) \equiv \\
\quad \textbf{case}\ (\textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\})\ \textbf{of} \\
\qquad Z \to Z \\
\qquad S(y) \to h(y)
\end{array}
$$

Теперь применим правило сопоставления с образцом результата сопоставления с образцом:

$$
\begin{array}{l}
(7)\, h(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad Z \to \textbf{case}\ Z\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\} \\
\qquad S(y) \to \textbf{case}\ h(y)\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}
\end{array}
$$

В ветви Z произведём редукцию сопоставления с образцом:

$$
\begin{array}{l}
(8)\, h(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad Z \to Z \\
\qquad S(y) \to \textbf{case}\ h(y)\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}
\end{array}
$$

Теперь возьмём определение (3) и раскроем в нём $h(x)$ по определению (5):

$$
\begin{array}{l}
(9)\, f(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad Z \to Z \\
\qquad S(y) \to \textbf{case}\ f(y)\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}
\end{array}
$$

Определения (8) и (9) очень похожи, и, действительно, к ним можно применить слияние по бисимуляции. Полипрограмма бисимуляции выглядит следующим образом:

$$
\begin{array}{l}
f(x) \equiv \\
\quad \textbf{case}\ x\ \textbf{of} \\
\qquad Z \to Z \\
\qquad S(y) \to \textbf{case}\ f(y)\ \textbf{of}\ \{Z \to Z; S(y) \to h(y)\}
\end{array}
$$

При этом $\phi_1 = \{f \mapsto f, h \mapsto h\}$ и $\phi_2 = \{f \mapsto h, h \mapsto h\}$. Это как раз пример, когда фрагменты не изоморфны. Нетрудно видеть, что рекурсия в полипрограмме бисимуляции структурная, поэтому можно добавить следующее определение к нашей полипрограмме:

$$(10)\, f(x) \equiv h(x)$$

Теперь применим конгруэнтность, заменив h на f в определении (3):

$$(11)\ f(x) \equiv \textbf{case}\ x\ \textbf{of}\ \{Z \to Z; S(y) \to f(y)\}$$

Теперь можно применить ещё одно слияние по бисимуляции для определений (11) и (2), что даст нам требуемое $f(x) \equiv g(x)$. $\triangle$

7 Заключение

В данной работе было введено понятие полипрограммы — программы, в которой каждой функции разрешено иметь несколько определений. Были рассмотрены полипрограммы для нестрогого функционального языка первого порядка.

Для полипрограмм была введена денотационная семантика, отличающаяся от случая обычных программ тем, что рассматривается не наименьшая неподвижная точка, а все неподвижные точки (модели). Это необходимо для замены эквивалентных фрагментов полипрограмм на эквивалентные.

Полипрограмма позволяет представлять множество обычных программ (монопрограмм), однако наивное выделение монопрограммы из полипрограммы путём выбора для каждой функции по одному определению может привести к монопрограмме, не эквивалентной всей полипрограмме. Поэтому при выделении монопрограммы необходимо проверять достаточные условия эквивалентности монопрограммы полипрограмме. В данной работе было рассмотрено применение признаков структурной и защищённой рекурсии, которые обычно применяются для обеспечения завершаемости и продуктивности. В случае нашего языка они обеспечивают единственность модели.

Были также рассмотрены возможные правила преобразования полипрограмм. Были введены две системы правил: одна похожая на схему Бёрстолла-Дарлингтона, лучше подходящая для ручных выкладок, а другая похожая на метод насыщения равенствами, лучше подходящая для программной реализации за счёт удаления лишних определений полипрограммы и более простой формулировки самих правил.

Наконец, было описано преобразование слияния по бисимуляции, позволяющее выводить равенство функций, если они имеют одинаковые рекурсивные определения с точностью до переименования функций. Это преобразование сталкивается с той же проблемой эквивалентности, что и выделение монопрограмм, и требует таких же проверок признаков единственности модели.

Понятие полипрограммы заслуживает дальнейшего изучения, в частности, для будущих исследований остаются следующие вопросы:

- Можно ли ввести операционную семантику полипрограмм, чтобы исполнять их напрямую, без выделения монопрограмм? При каких условиях такая операционная семантика будет согласована с денотационной?
- Можно ли иначе ввести денотационную семантику (например, на основе теории Сэндса [12]), чтобы упростить признаки единственности модели?

(В идеале хотелось бы, чтобы единственность выполнялась автоматически.)

- Имеет ли практический смысл вместо проверок признаков единственности при выделении монопрограмм проверять эти признаки при совершении опасных преобразований (свёртки) так, чтобы монопрограммы можно было извлекать наивным способом?

- Возникнут ли дополнительные сложности, если язык будет не первого порядка, а высшего порядка?

- Как обеспечить конфлюэнтность и завершаемость системы правил, включающей деструктивные правила?

Список литературы

1. *Abel A.* foetus – Termination Checker for Simple Functional Programs: Programming Lab Report / LMU München. — July 1998.
2. *Abel A.*, *Altenkrich T.* A Predicative Analysis of Structural Recursion // Journal of Functional Programming. — 2002. — Vol. 12, no. 1. — Pp. 1–41. — URL: `http://www.cs.cmu.edu/~abel/foetuswf.ps.gz`.
3. *Burstall R. M.*, *Darlington J.* A transformational system for developing recursive programs // Journal of the ACM. — 1977. — Jan. — Vol. 24, no. 1. — Pp. 44–67.
4. Equality saturation: a new approach to optimization / R. Tate, M. Stepp, Z. Tatlock, S. Lerner // SIGPLAN Not. — New York, NY, USA, 2009. — Jan. — Vol. 44, issue 1. — Pp. 264–276. — ISSN 0362-1340. — URL: `http://doi.acm.org/10.1145/1594834.1480915`.
5. *Grechanik S.* Inductive Prover Based on Equality Saturation (Extended Version) // Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. — Pereslavl-Zalessky, Russia : Pereslavl Zalessky: Publishing House "University of Pereslavl", July 2014. — Pp. 26–53. — ISBN 978-5-901795-31-6.
6. *Grechanik S.* Towards Unification of Supercompilation and Equality Saturation (Extended Abstract) // Proceedings of the Fifth International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. — Pereslavl-Zalessky, Russia : Pereslavl Zalessky: Publishing House "University of Pereslavl", June 2016. — Pp. 52–55. — ISBN 978-5-901795-33-0.

7. *Grechanik S. A.* Overgraph Representation for Multi-Result Supercompilation // Proceedings of the Third International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. — Pereslavl-Zalessky, Russia : Pereslavl-Zalessky: Ailamazyan University of Pereslavl, July 2012. — Pp. 48–65. — ISBN 978-5-901795-28-6. — URL: `http://pat.keldysh.ru/~grechanik/doc/overgraph.pdf`.

8. *Grechanik S.*, *Klyuchnikov I.*, *Romanenko S.* Staged Multi-Result Supercompilation: Filtering by Transformation // Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation / ed. by A. Klimov, S. Romanenko. — Pereslavl-Zalessky, Russia : Pereslavl Zalessky: Publishing House "University of Pereslavl", July 2014. — Pp. 54–78. — ISBN 978-5-901795-31-6.

9. *Nelson*, *Oppen* Fast Decision Procedures Based on Congruence Closure // JACM: Journal of the ACM. — 1980. — Vol. 27, no. 2. — Pp. 356–364.

10. *Nieuwenhuis R.*, *Oliveras A.* Congruence Closure with Integer Offsets // Logic for Programming, Artificial Intelligence, and Reasoning, 10th International Conference, LPAR 2003, Almaty, Kazakhstan, September 22-26, 2003, Proceedings. Vol. 2850 / ed. by M. Y. Vardi, A. Voronkov. — Springer, 2003. — Pp. 78–90. — (Lecture Notes in Computer Science). — ISBN 3-540-20101-7. — URL: `http://dx.doi.org/10.1007/978-3-540-39813-4_5`.

11. *Plump D.* Evaluation of Functional Expressions by Hypergraph Rewriting: PhD thesis / Plump D. — Univ. Bremen, 1993.

12. *Sands D.* Total correctness by local improvement in the transformation of functional programs // ACM Trans. Program. Lang. Syst. — 1996. — Vol. 18, no. 2. — Pp. 175–234.

13. *Scott D.*, *Strachey C.* Towards a Mathematical Semantics for Computer Languages: Technical Report / Oxford University Computer Laboratory. — 1971. — PRG–6.

14. *Sørensen M.*, *Glück R.*, *Jones N.* A Positive Supercompiler. // Journal of Functional Programming. — 1993. — Vol. 6, no. 6. — Pp. 811–838.

15. *Turchin V.* The concept of a supercompiler // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1986. — Vol. 8, no. 3. — Pp. 292–325.

16. *Wadler P.* Deforestation: Transforming Programs to Eliminate Trees // ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300). — Berlin: Springer-Verlag, 1988. — Pp. 344–358.