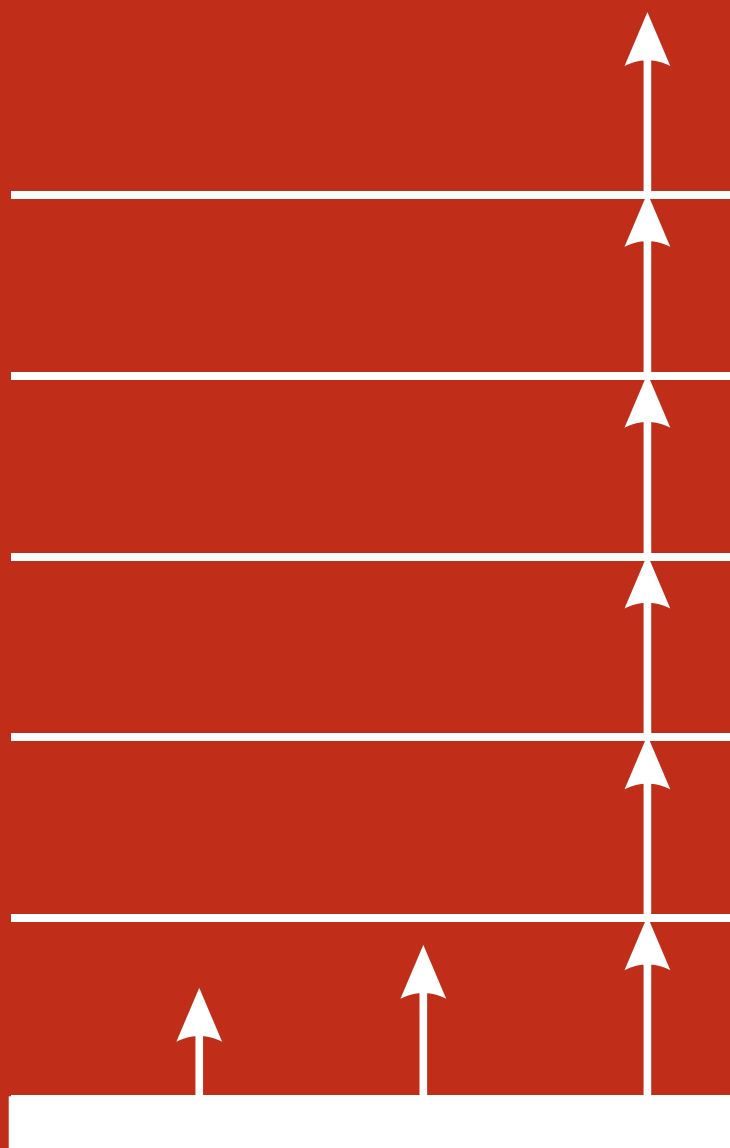


С.М.Абрамов

Метавычисления и их применение



С.М.Абрамов

Метавычисления
и их применение

ББК 32.97
С72
УДК 681.322

Абрамов С.М.

Методы метавычислений и их применение.—Издание второе, дополненное и переработанное, Переславль-Залесский, Издательство «Университет города Переславля имени А.К.Айламазяна», 2006.—128 с., ил.

Книга сотрудника Института программных систем РАН, представляющая собой описание вопросов теории метавычислений и их применения. Метавычисления—раздел теории и практики программирования, посвященный разработке методов анализа и преобразования программ за счет реализации конструктивных метасистем (метапрограмм) над программами. В книге приводятся основные понятия метавычислений, примеры построения и использования простых метапрограмм. Изложение рассчитано на специалистов в области программирования.

Для научных сотрудников и студентов, изучающих методы автоматического преобразования программ или проводящих исследования в данной области.

Оглавление

Введение	7
1 Язык реализации	13
1.1 Данные в языке TSG	13
1.2 Синтаксис языка TSG	13
1.3 Семантика языка TSG	16
2 Представление множеств	21
2.1 С-переменные, с-выражения	21
2.2 С-связи, с-среды и с-состояния	22
2.3 Рестрикции с-переменных	23
2.4 С-конструкции	26
2.5 Подстановки	27
2.6 Отождествление с-выражений	30
2.7 Представляемое множество	33
2.8 Суперпозиция подстановок	37
2.9 Сужения и разбиения множеств	38
3 Дерево процессов	45
3.1 Дерево конфигураций	45
3.2 Построение дерева процессов	46
4 Окрестностный анализ	55
4.1 Основные понятия	55
4.2 Окрестности	56
4.3 Окрестностный анализатор	58
4.4 Операции над классами и окрестностями	66
5 Окрестностное тестирование программ	73
5.1 Тестирование. Основные понятия	73
5.2 Постановка проблемы	76
5.3 Построение окрестностного тестирования	78
5.4 Окрестностный критерий выбора тестов	81
5.5 Свойства окрестностного тестирования	82
5.6 Решение традиционных проблем теории тестирования	83

6	Универсальный решающий алгоритм	91
6.1	Постановка проблемы	91
6.2	Приведение функции программы к табличной форме	92
6.3	Алгоритм инверсного вычисления программ	95
6.4	Развитие УРА	100
6.5	Различные подходы к инверсии программ	106
7	Инверсное программирование	109
7.1	Инверсная семантика языка реализации	109
7.2	Инверсное и логическое программирование	110
7.3	Перенос инверсной семантики	117
8	Нестандартные семантики	121
8.1	Модификаторы семантик	121
8.2	Специализация программ	124
8.3	Эффективная реализация нестандартных языков	127
	Литература	129

Список иллюстраций

1.1	Синтаксис данных и программ в языке TSG	14
1.2	Программа проверки вхождения подстроки в строку	15
1.3	Синтаксис среды в языке TSG	17
1.4	Вспомогательные функции интерпретатора языка TSG	18
1.5	Интерпретатор языка TSG	19
2.1	Синтаксис с-переменных и с-выражений	22
2.2	Синтаксис с-состояния, с-среды и с-связи	23
2.3	Рестрикции с-переменных	24
2.4	С-конструкции	27
2.5	Подстановка	28
2.6	Алгоритм отождествления списков с-выражений	32
2.7	Синтаксис сужения для языка TSG	38
2.8	Определение отношения $<$ на множестве с-выражений	42
3.1	Синтаксис представления дерева конфигураций	46
3.2	Алгоритм построения дерева процессов. Функции <code>ptr</code> и <code>evalT</code>	47
3.3	Алгоритм построения дерева процессов. Функция <code>ccond</code>	48
3.4	Улучшенный алгоритм построения дерева процессов	52
4.1	Вспомогательная функция <code>mkCExps</code>	58
4.2	Окрестностный анализатор для языка TSG	60
4.3	Пересечение и разность классов	67
4.4	Декомпозиция окрестности	69
4.5	Алгоритм декомпозиции окрестности	70
5.1	Процесс тестирования программы	75
5.2	Конечно-автоматное преобразование строки	77
5.3	Синтаксис конструкций языка JobSL	86
5.4	Программа <code>corr(φ, p, ψ)</code> на языке JobSL	87
5.5	Вспомогательные функции интерпретатора языка JobSL	87
5.6	Интерпретатор языка JobSL	88
6.1	Алгоритм построения табличной формы функции	94
6.2	Структура алгоритма инверсного вычисления программ	97
6.3	Алгоритм инверсного вычисления программ	97
6.4	Альтернативное представление результатов инверсного вычисления	98

6.5	Алгоритм mg вычисления наиболее общего унификатора для заданной системы клэшей	101
6.6	Алгоритм пересечения двух классов	103
6.7	Структура алгоритма sura	104
6.8	Алгоритма sura	105
7.1	Синтаксис представления графов, пример представления	113
7.2	Алгоритм инверсного вычисления в произвольном языке L	119

*С благодарностью посвящается Учителю
Валентину Федоровичу Турчину*

Введение

В 1975 году автору данной работы, тогда студенту, посчастливилось быть вовлеченным в работу неформальной Московской рефал-группы и познакомиться с руководителем группы, замечательным ученым и человеком В.Ф.Турчиным. Рефал-группа занималась разработкой теории метавычислений и практической реализацией средств метавычислений для языка Рефал. С тех пор теория и практика метавычислений стали главными научными интересами автора, чему он всегда будет обязан В.Ф.Турчину и всем участникам рефал-группы.

В данной работе рассмотрены различные результаты, полученные в теории и практике метавычислений. Это не означает, что здесь собраны *все* результаты. Даже, наоборот, самая большая (самая интересная, но и самая сложная) часть—теория суперкомпиляции—осталась за рамками рассмотрения. Здесь главное внимание было уделено *основным, общим для различных метапрограмм, понятиям*, построению простейших метапрограмм и демонстрации того, чего можно достичь применением даже этих тривиальных метапрограмм в теоретическом и практическом программировании.

Было бы ошибкой считать, что представлены только результаты, полученные самим автором. Некоторые результаты принадлежат В.Ф.Турчину, некоторые—участникам рефал-группы, некоторые—автору данной работы, а остальные были получены в дискуссиях на рефал-семинарах, то есть в условиях, когда установить конкретное авторство достаточно тяжело. Тем не менее, в дополнительных материалах [FTP]¹ к данной работе среди прочего приводится список ссылок на авторов идей и результатов, включенных в данную книгу.

В последние годы исследователи за рубежом, которые занимались в близких областях (например, смешанными вычислениями), проявляют большой интерес к теории метавычислений Турчина. Отрадно видеть, что и на родине теории (даже в нынешнее время) интерес к ней сохранился. Более того, к данной работе присоединяются новые научные силы. Особенно приятно, что в Институте программных систем РАН складывается коллектив исследователей, активно занимающихся теорией и практикой метавычислений, и что эта работа выполняется в тесном сотрудничестве с В.Ф.Турчиным. Хочу выразить свою благодарность Институту за то, что он всегда признавал важность данных работ и поддерживал их. Так и с этой книгой—без поддержки Института она бы не состоялась.

Основное содержание работы

В данной работе рассматриваются вопросы теории *метавычислений* и примеры использования методов метавычислений. Метавычисления—это раздел теории и практики программирования, посвященный разработке методов анализа и преобразования программ за счет реализации конструктивных метасистем (метапрограмм) над программами. В метавычисления в первую очередь включают теорию суперкомпиляции и близкие

¹ О дополнительных материалах [FTP]—см. в конце Введения.

методы и средства [1, 4, 10, 14, 15, 35]. Приставка “мета” указывает на то, что программа в метавычислениях рассматривается как *объект* анализа и/или преобразования.

Первые работы по метавычислениям были выполнены В.Ф.Турчиным, при участии других членов московской Рабочей группы по языку Рефал, в 1970-тых годах [1, 3, 4]. Базовыми идеями данных работ являлись:

- применение теории метасистем и метасистемных переходов [2, 4, 19, 2];
- *процесс-ориентированный подход* к построению методов анализа и преобразования программ—разработка метапрограмм, которые “наблюдают” за *процессами* вычисления исходной программы и управляют ими;
- использование языка программирования Рефал [8] и его диалектов в качестве языка реализации метапрограмм и программ, над которыми выполняются метавычисления.

Рассмотрим подробнее основные принципы разрабатываемых в данных работах методов.

Для проведения метавычислений над некоторой программой p вводится в рассмотрение метапрограмма M , для которой программа p является объектом анализа, преобразования и т.п. Метапрограмма M должна быть способна анализировать процессы выполнения программы p на конкретных данных d и на классах (множествах) C данных. По сути, метапрограмма M наблюдает за множеством процессов² выполнения программы p и управляет данными процессами. Таким образом, M является метасистемой по отношению к системе p . Именно из результатов наблюдения за процессами выполнения программы p метапрограмма M определяет наличие тех или иных свойств программы p , выводит возможность применения того или иного преобразования программы.

Для построения методов анализа процессов выполнения программ необходимо определить и зафиксировать некоторый язык программирования R —язык реализации. Все программы p , к которым можно применять метапрограмму M , должны быть написаны на языке R . Если сама метапрограмма M будет написана на языке R , то будет возможно применить к ней ее самую (самоприменение) или другие метапрограммы. Это открывает возможность автоматического выполнения нескольких метасистемных переходов. На практике это означает возможность автоматических кардинальных преобразований программ—как эквивалентных преобразований, так и построение новых программ, функции которых сложным образом определяются через функции исходных программ.

Главные усилия большинства работ по метавычислениям были направлены на разработку *суперкомпилятора* [3, 7]. Эти усилия оказались плодотворными—в последнее время появились первые реализации суперкомпилятора [5, 21, 22], демонстрирующие на практике нетривиальные примеры метавычислений, принципиальная осуществимость которых была показана еще в первых работах по метавычислениям.

Стремление к скорейшему завершению работ по созданию теории суперкомпиляции и практической реализации суперкомпилятора привело к тому, что некоторые методы метавычислений и некоторые вопросы применения метавычислений в практическом программировании были недостаточно исследованы. *Желание хотя бы частично закрыть данный пробел и привело к появлению этой работы.*

² *Процесс-ориентированность*—одна из принципиальных отличительных черт подхода В.Ф.Турчина.

Данная работа посвящена следующим вопросам:

1. Базовые понятия метавычислений.
2. Разработка и исследование *окрестностного анализа*—одной из разновидностей метавычислений.
3. Применение окрестностного анализа в тестировании программ. Построение и исследование свойств нового метода тестирования программ—*окрестностного тестирования*.
4. Изучение *инвертирования* программ, основанного на использовании одного из средств метавычислений—*универсального решающего алгоритма*, УРА. Введение понятий *инверсной семантики* и *инверсного программирования*.
5. Исследование понятия *нестандартная семантика языка программирования*. Анализ окрестностного тестирования и инверсного программирования как двух примеров нестандартных семантик. Исследование возможности автоматического получения при помощи метавычислений эффективных реализаций нестандартных семантик.

Таким образом, вопросы теории и практики суперкомпиляции (из которых и начала формироваться теория метавычисления) останутся за рамками рассмотрения данной работы. Заинтересованным данными вопросами можно порекомендовать обратиться к соответствующим работам, описывающим базовые принципы [3, 4, 7, 18], технические вопросы реализации суперкомпилятора [9, 11, 12] и примеры суперкомпиляции [5, 6, 22].

Структура изложения

Как сказано выше, при разработке методов метавычислений необходимо определить *язык реализации R*, для которого разрабатываются метавычисления.

В большинстве работ по метавычислениям в качестве языка реализации использовался язык Рефал и его диалекты. В данной работе в качестве языка реализации будет использован язык TSG—простой и компактный алгоритмически полный язык, диалект языка S-Graph, использованного в работе [18]. Простота TSG и предметной области языка позволяет значительно упростить изложение методов метавычислений для TSG, в то же время не упуская ни одного *существенного* момента.

Фиксация языка реализации позволяет давать конкретные описания алгоритмов метавычислений и доказывать их свойства, и вовсе не означает, что метавычисления могут быть определены только для данного языка. Все построения и рассуждения могут быть повторены и для любого другого языка реализации, если он удовлетворяет тем свойствам, которые существенно использовались в алгоритмах метавычислений и в доказательствах свойств этих алгоритмов³.

В главе 1 будут даны описание синтаксиса языка TSG, интерпретационное определение семантики TSG, введены такие понятия как: *состояние* вычисления программы p

³ В данной работе во всех построениях и доказательствах используются только следующие свойства языка TSG: *алгоритмическая полнота языка* и определение предметной области языка при помощи конечного набора *жестких* конструкторов над конечным алфавитом атомов. Таким образом, все результаты данной работы справедливы и для любого другого языка реализации с такими же свойствами.

над данными d , *процесс* вычисления программы p над данными d —последовательность переходов от одного состояния вычисления программы p к другому,—*трасса вычисления* программы p над данными d .

При выполнении метавычислений имеют дело не только с одиночными данными (значениями) программы, с конкретными (одиночными) состояниями вычисления и т.п., но также и с *множествами* данных, *множествами* состояний вычислений и т.д. Эти множества должны быть представлены конструктивно—в виде выражений некоторого языка. Таким образом, предстоит *разработать* конструктивные *методы представления* таких множеств. При этом надо учитывать следующее обстоятельство: при любом выборе способа конструктивного представления множеств найдутся множества данных, непредставимые данным способом⁴. Следовательно, для каждого приложения, в котором необходимо представлять множества данных в виде конструктивных объектов, необходимо *выбирать* средства представления множеств данных так, чтобы все *интересующие нас* в данном приложении множества были бы представимы данными средствами.

В главе 2 будут определены такие средства представления множеств данных в виде выражений—так называемых *классов*, что все множества данных, возникающие в процессе метавычислений, окажутся представимыми в виде классов. Здесь же предъявлены и обоснованы алгоритмы выполнения операций над классами, определены средства представлений и других множеств, необходимых для метавычислений, например, представление множеств состояний вычислений в виде так называемых *конфигураций*.

Для большинства алгоритмов метавычислений базовым понятием является понятие *дерева процессов* и базовым алгоритмом—алгоритм построения дерева процессов.

В главе 3 будет дано точное определение дерева процессов вычисления программы p на классе данных, приведен и обоснован алгоритм построения этого дерева.

Окрестностный анализ—один из методов метавычислений—является инструментом формализации интуитивного вопроса:

Какая информация о тексте d была использована, и какая информация о d не была использована в некотором процессе обработки текста d ?

Здесь рассматриваются *конструктивные* процессы обработки текста, реализованные некоторой программой p на языке R .

Окрестностный анализ впервые был разработан В.Ф.Турчиным, при участии других членов московской Рабочей группы по языку Рефал, в 1970-ых годах в контексте работ над суперкомпилятором для языка Рефал [7, 9]. Первоначально окрестностный анализ рассматривался как *специальный* метод, предназначенный для построения алгоритмических “приближенных” решений неразрешимой проблемы останова и с тех пор он использовался только для этих целей. Как результат, были разработаны только те теоретические вопросы окрестностного анализа, которые имели отношение к данному приложению.

По своей природе окрестностный анализ является методом общего применения, а не средством для решения одной частной задачи. Поэтому представляется интересным построить и изучить точный окрестностный анализ именно как общий метод, без привязки к какому-то применению.

⁴ Это вытекает из сравнения мощностей множества различных множеств данных— 2^{\aleph_0} —и мощности различных представлений—не более \aleph_0 .

В главе 4 будет дано полное описание окрестностного анализа, на базе алгоритма построения дерева процессов, будет построен и обоснован алгоритм окрестностного анализатора *nap*, определены некоторые операции над окрестностями и исследованы их свойства, даны примеры вычисления окрестностей.

Будучи по своей природе методом общего назначения, окрестностный анализ может быть применен в различных областях теории и практики программирования. В данной работе рассмотрено использование окрестностного анализа в *тестировании программ*.

В главе 5 будет сформулирован неформальный подход построения “предельно надежного” метода тестирования. Окрестностный анализ позволит выполнить формализацию данного подхода. Будет предложен новый метод тестирования программ— *окрестностное тестирование*. В этой же главе будут исследованы свойства и возможности окрестностного тестирования.

В данной работе будет рассмотрена еще одна модификация алгоритма построения дерева процессов— *универсальный решающий алгоритм (УРА)*, являющийся средством вычисления значений функций, обратных к функции заданной программы— *средством инверсного вычисления программ*.

В главе 6 будет описан универсальный решающий алгоритм *ura* для языка TSG и исследованы его свойства.

Реализуя инверсное вычисление программ, УРА позволяет рассматривать в дополнение к стандартному вычислению программ на языке R (стандартной семантике) инверсное вычисление программ (инверсную семантику языка R).

Использование УРА позволяет писать программы в стиле *инверсного программирования*. То есть создавать программу не с требуемой функцией *f*, а с такой функцией *p*, которая при инверсном вычислении совпадает с *f*.

В главе 7 будет введено понятие инверсной семантики и инверсного программирования, будет исследована связь между инверсным и логическим программированием, будет предложен способ переноса инверсной семантики с языка R на любой язык L.

В окрестностном тестировании и в инверсном программировании используются похожие схемы метавычислений с двумя уровнями интерпретации. Обе эти схемы метавычислений являются частными случаями общей схемы реализации *нестандартной семантики*, получаемой при помощи некоторого *модификатора семантики md*. При этом, модификатор *md* позволяет реализовать нестандартную семантику для любого языка L. Однако, эта изначальная схема реализации оказывается весьма неэффективной из-за двух вложенных уровней интерпретации.

В главе 8 будут введены понятия *модификатор семантики*, *нестандартная семантика*. Будет показано, что при помощи метавычислений—а именно, при помощи *специализации*,—по заданному модификатору $md \in R$ и интерпретатору $intL \in R$ произвольного языка L возможно автоматическое построение интерпретирующей (с одним уровнем интерпретации) и компилирующей реализации нестандартной семантики языка L, порожденной модификатором *md*. Универсальный решающий алгоритм *ura* и окрестностный анализатор *nap* будут рассмотрены как частные примеры модификаторов семантик.

FTP-приложение

Технические детали излагаемого материала—подробные доказательства утверждений, развернутые примеры и т.п.,—вынесены из данной книги. Они оформлены в виде

файлов, доступных⁵ заинтересованному читателю по анонимному FTP:

```
ftp://ftp.botik.ru/pub/local/Sergei.Abramov/book.appndx/
```

Везде ниже ссылка на данные материалы будет обозначаться следующим образом: [FTP].

Используемые обозначения

В данной книге используются обозначения, общепринятые в работах по математике и программированию. Для записи алгоритмов метапрограмм использован входной язык системы **Gofer**, являющийся диалектом языка Haskell [38].

Все приводимые в работе тексты метапрограмм являются фрагментами работающей Gofer-программы, а все приводимые примеры метавычислений—результатами реального выполнения этой программы. Полный текст программы и примеров можно найти в [FTP].

Систему **Gofer** и документацию по ней заинтересованный читатель может получить по анонимному FTP:

```
ftp://ftp.cs.nott.ac.uk/haskell/gofer/, или  
ftp://ftp.dcs.glasgow.ac.uk/pub/haskell/gofer/, или  
ftp://ftp.botik.ru/pub/lang/gofer/
```

Следует пояснить, что приведенные описания метапрограмм являются скорее *исполняемыми спецификациями*, а не промышленной реализацией метавычислений. При их создании автор стремился достичь ясности и компактности описания алгоритмов, часто в ущерб эффективности.

⁵ В случае затруднений с получением данных материалов просьба обращаться к автору данной работы по электронному адресу: “abram@botik.ru”.

Глава 1

Язык реализации

В данном разделе будет определен язык TSG, используемый в качестве языка реализации при описании метавычислений в последующих разделах. Язык TSG¹ является модификацией языка S-Graph [18], который, в свою очередь, тесно связан с понятием “Рефал-граф” из работы [9].

Как и S-Graph, язык TSG является алгоритмически полным, функциональным языком первого порядка, ориентированным на обработку символьной информации.

1.1 Данные в языке TSG

Определение 1

В TSG в качестве данных рассматриваются:

- атомы—из некоторого фиксированного *конечного* множества атомов **Atoms**;
- S-выражения, построенные из атомов при помощи конструктора **CONS**—нециклические списки, известные из языка Lisp.

Синтаксис данных (значений) приведен на рисунке 1.1. Будем использовать следующие термины и обозначения:

- *a-значения*—значения, имеющие вид *a-val*, то есть (**ATOM** *atom*);
- *e-значения*—значения, имеющие вид *e-val*—a-значения или произвольные S-выражения;
- **AVal**—множество всех a-значений;
- **EVal**—множество всех e-значений;

1.2 Синтаксис языка TSG

На рисунке 1.1 приведено описание синтаксиса TSG-программ.

Будем использовать следующие сокращения:

¹ “TSG” от английского “Typed S-Graph”—типизированный вариант языка S-Graph.

```

a-val ::= (ATOM atom )

e-val ::= a-val | (CONS e-val e-val )

prog  ::= [ def1, ..., defn ]                -- n ≥ 1

def   ::= (DEFINE fname
              [ parm1, ..., parmn ] term )    -- n ≥ 0

term  ::= (ALT cond term1 term2 )
           | (CALL fname [ arg1, ..., argn ] )    -- n ≥ 0
           | e-exp

cond  ::= (EQA' a-exp a-exp )
           | (CONS' e-exp e-var e-var a-var )

a-exp ::= a-val | a-var

e-exp ::= a-val | a-var | e-var | (CONS e-exp e-exp)

parm  ::= a-var | e-var

arg   ::= a-exp | e-exp

a-var ::= (PVA name )

e-var ::= (PVE name )

```

Рис. 1.1: Синтаксис данных и программ в языке TSG

- '*atom*'—сокращенная запись а-значения (ATOM *atom*);
- *a.name*, *e.name*—сокращенные записи для а- и е-переменных.

На рисунке 1.2 приведен пример простой программы проверки вхождения подстроки в строку, записанной с использованием данных сокращений.

1.2.1 Типы 'а-' и 'е-' в языке TSG

Язык TSG определен таким образом, что

- значением а-переменной или а-выражения может быть только а-значение:

(ATOM *atom*);

- значением е-переменной или е-выражения может быть произвольное е-значение: S-выражение или (ATOM *atom*).


```

prog :: Prog
prog = [
  (DEFINE "Match" [e.substr,e.string]
    (CALL "CheckPos" [e.substr,e.string,e.substr,e.string])
  ),
  (DEFINE "CheckPos" [e.subs,e.str,e.substring,e.string]
    (ALT (CONS' e.subs e.subs-head e.subs-tail a._)
      (ALT (CONS' e.subs-head e._ e._ a.subs-head)
        'FAILURE
      (ALT (CONS' e.str e.str-head e.str-tail a._)
        (ALT (CONS' e.str-head e._ e._ a.str-head)
          'FAILURE
        (ALT (EQA' a.subs-head a.str-head)
          (CALL "CheckPos" [e.subs-tail,e.str-tail,
            e.substring,e.string])
          (CALL "NextPos" [e.substring,e.string])
        )
      )
    )
    'FAILURE
  )
  )
  'SUCCESS
),
  (DEFINE "NextPos" [e.substring,e.string]
    (ALT (CONS' e.string e._ e.string-tail a._)
      (CALL "Match" [e.substring,e.string-tail])
    'FAILURE
  )
)
]

```

Рис. 1.2: Программа проверки вхождения подстроки в строку

Типы формальных и фактических параметров функций в языке TSG обязаны быть согласованными. Это означает, что в вызове функции (`CALL fname [arg1, ..., argn]`) длина списка фактических параметров должна равняться длине списка параметров в описании функции `fname`: (`DEFINE fname [parm1, ..., parmn] term`), и, если `parmi` имеет тип 'а-', то фактический параметр `argi` обязан быть а-выражением, а если `parmi` имеет тип 'е-', то `argi` может быть как а-, так и е-выражением.

В языке TSG первая функция программы считается входной. Поэтому входные данные программы должны удовлетворять требованиям, приведенным выше. А именно, для программы

$$p = [(\text{DEFINE } f \text{ } [parm_1, \dots, parm_n] \text{ term}), \dots]$$

в качестве входных данных допустим список а- и е-значений

$$[data_1, \dots, data_n] \in [EVal]$$

длина которого равняется длине списка параметров в описании первой функции f из программы p и, если parm_i имеет тип ‘a-’, то data_i является а-значением, а если parm_i имеет тип ‘e-’, то data_i может быть как а-, так и е-значением.

Таким образом, множество списков е-значений $D = [\text{EVal}]$ является множеством возможных значений входных данных для различных программ на языке TSG, то есть предметной областью языка TSG.

1.2.2 Условия ветвлений в языке TSG

В условных конструкциях ($\text{ALT } \text{cond } \text{term}_1 \text{ term}_2$) языка TSG в качестве условия ветвления— cond —могут быть использованы следующие конструкции:

- (EQA’ $x \ y$)

Проверка равенства двух а-значений (атомов)—если значения а-выражений x и y —суть один и тот же атом, то проверка считается успешной (условие выполнено), в противном случае проверка считается неуспешной (условие не выполнено);

- (CONS’ $x \ \text{evh} \ \text{evt} \ \text{av}$)

Анализ вида е-значения:

1. если значение $xval$ е-выражения x имеет вид ($\text{CONS } a \ b$), то е-переменные evh , evt связываются (принимают значения) соответственно с е-значениями a , b и проверка считается успешной;
2. если $xval$ имеет вид ($\text{ATOM } a$), то а-переменная av связывается с а-значением $xval$ и проверка считается неуспешной.

Заметим, что действие проверки ($\text{CONS’ } x \ \text{evh} \ \text{evt} \ \text{av}$) включает в себя определение значений для переменных evh и evt или переменной av , в зависимости от успеха или неуспеха проверки. То есть, в данной проверке *определяются новые* переменные, и в условной конструкции ($\text{ALT } (\text{CONS’ } x \ \text{evh} \ \text{evt} \ \text{av}) \ \text{term}_1 \ \text{term}_2$) допустимо использовать переменные evh и evt в терме term_1 , а переменную av —в терме term_2 .

1.3 Семантика языка TSG

Определение языка TSG мы завершим, предъявив алгоритм интерпретатора int для него. Предварительно введем несколько понятий.

Определение 2

Будем называть *средой* список из *связей*—упорядоченных пар вида $\text{var} := \text{value}$, связывающих программные переменные— var —с их значениями— value , при этом:

- значением а-переменной может быть только а-значение: ($\text{ATOM } \text{atom}$);
- значением е-переменной может быть произвольное е-значение: ($\text{ATOM } \text{atom}$) или S-выражение.

Упорядоченную пару $(term, env)$, где $term$ —программный терм, env —среда, связывающая программные переменные из $term$ с их значениями, будем называть *состоянием вычисления*.

Точное определение синтаксиса среды, состояния и связи дано на рисунке 1.3.

```

state      ::= (term, env)

env        ::= [ binding1, ..., bindingn ]  -- n ≥ 0

binding    ::= a-var := a-val
              | e-var := e-val

```

Рис. 1.3: Синтаксис среды в языке TSG

В определении алгоритма интерпретатора **int** мы будем использовать следующие функции и операторы (описание вспомогательных функций и операторов приведено на рисунке 1.4):

- **t /. env**

замена в конструкции **t** вхождений переменных на их значения из среды **env**. Например, в результате операции:

```
(CONS (CONS a.x e.z) a.y) /. [a.x:='A, a.y:='B, e.z:='C]
```

получаем:

```
(CONS (CONS 'A 'C) 'B)
```

- **env +. env'**

обновление среды **env** средой **env'**—результатом является среда, связывающая переменную *var* со значением *value* в том, и только в том случае, если:

- *var* связано со значением *value* в среде **env'**, или
- *var* связано со значением *value* в среде **env** и при этом *var* не связано ни с каким значением в среде **env'**.

Например, результатом операции:

```
[a.x:='A, a.y:='B, e.z:='C] +. [e.z:=(CONS 'D 'E), e.u:='F]
```

является

```
[e.z:=(CONS 'D 'E), e.u:='F, a.x:='A, a.y:='B]
```

В определении оператора **(+.)** используются: встроенный оператор **(++)** конкатенации списков; встроенная функция **nub** удаления из списка повторных (в смысле отношения **(==)**) элементов; отношение **(==)** для связей—две связи считаются “равными”, если они связывают одну и ту же программную переменную.

```

class    APPLY a b where (/.) :: a -> b -> a
instance APPLY Exp Env where
  (ATOM a)    /. env = ATOM a
  (CONS h t)  /. env = CONS (h/.env) (t/.env)
  var         /. env = head [ x | (v:=x) <-env, v==var ]

instance Eq Bind where
  (var1:=_) == (var2:=_) = (var1==var2)

class    UPDATE a where (+.) :: a -> a -> a
instance UPDATE Env where
  binds +. binds' = nub (binds'++binds)

mkEnv    :: [Var] -> [EVal] -> Env
mkEnv    = zipWith (\var val -> (var:=val))

getDef   :: Fname -> ProgR -> FDef
getDef fn p = head [ fd | fd@(DEFINE f _ _)<-p, f==fn ]

```

Рис. 1.4: Вспомогательные функции интерпретатора языка TSG

- **mkEnv vars vals**
построение среды по списку переменных **vars** и списку значений **vals**. Например, результатом вычисления:

```
mkEnv [a.x, a.y, e.z, e.u] ['A, 'B, (CONS 'D 'E), 'F ]
```

является среда

```
[ a.x:= 'A, a.y:= 'B, e.z:=(CONS 'D 'E), e.u:= 'F ]
```

- **getDef fname prog**
в тексте программы **prog** находит и возвращает в качестве своего результата определение функции с именем **fname**.

На рисунке 1.5 приведен алгоритм интерпретатора **int** языка TSG. По программе **p** и входным данным **d** в функции **int** строится начальное состояние вычисления **p d** — $s_0 = (t_0, e_0)$, а затем (см. функцию **eval**) выполняются переходы от одного состояния к другому:

$s_0 \Rightarrow s_1 \Rightarrow \dots$,
где $s_i = (t_i, e_i)$, t_i — терм, e_i — среда, $i = 0, 1, \dots$

Переходы от состояния к состоянию (рекурсивные вызовы функции **eval**) производятся до тех пор, пока не будет получено *пассивное* состояние $s_n = (t_n, e_n)$, где t_n имеет вид е-выражения. В этом случае — см. последнее предложение в определении **eval** — значение $res = t_n /. e_n$ выдается в качестве результата вычисления **p d**.

```

int :: ProgR -> [EVal] -> EVal
int  p d = eval s p
      where (DEFINE f prms _) : p' = p
            e = mkEnv prms d
            s = ((CALL f prms), e)

eval :: State -> ProgR -> EVal
eval s@((CALL f args), e) p = eval s' p
      where DEFINE _ prms t' = getDef f p
            e' = mkEnv prms (args/.e)
            s' = (t',e')

eval s@((ALT c t1 t2), e) p = case cond c e of
      TRUE  ue -> eval (t1,e+.ue) p
      FALSE ue -> eval (t2,e+.ue) p

eval s@(exp,e)          p = exp/.e

data CondRes = TRUE Env | FALSE Env
cond :: Cond -> Env -> CondRes
cond (EQA' x y)          e = let x' = x/.e; y' = y/.e in
      case (x', y') of
        (ATOM a, ATOM b) | a==b -> TRUE [ ]
        (ATOM a, ATOM b)          -> FALSE[ ]

cond (CONS' x vh vt va) e = let x' = x/.e in
      case x' of
        CONS h t          ->TRUE [vh:=h,vt:=t]
        ATOM a             ->FALSE[va:=x']

```

Рис. 1.5: Интерпретатор языка TSG

Определение 3

Процесс вычисления программы p над входным данным $d \in D$ будем обозначать следующим образом:

$$p \ d \xRightarrow{*} \text{res}$$

где $\text{res} \in D$ —результат вычисления.

Последовательность (возможно бесконечную) состояний вычисления p над данными d обозначим:

$$\text{process}(p,d) = s_0 \Rightarrow s_1 \Rightarrow \dots$$

Рассмотрим возможно бесконечный список (последовательность) результатов—TRUE или FALSE,—полученных при выполнении проверок (см. функцию `cond`) условий `EQA'`

и CONS' , производимых в процессе вычисления p над d . Данный список мы будем называть *трассой* вычисления p над d и обозначать $\text{tr}(p, d)$. Начальный отрезок списка $\text{tr}(p, d)$, соответствующий первым i шагам вычисления p над d , обозначим $\text{tr}_i(p, d)$.

Глава 2

Представление множеств

В метавычислениях имеют дело не только с одиночными данными (значениями), одиночными состояниями вычисления и т.п., но также и с *множествами* данных, *множествами* состояний вычислений и т.д. Эти множества должны быть представлены конструктивно—в виде выражений некоторого языка. Для этих целей, как и в [7, 18], будут использоваться *выражения со свободными переменными* (с так называемыми *конфигурационными* переменными или *с-переменными*).

Представление множеств при помощи выражений со свободными переменными (параметрами) принято в математике. Так, довольно часто множество нечетных чисел представляют при помощи выражения $2n + 1$ со свободной целочисленной переменной n , а выражением $(5 \cos \phi, 5 \sin \phi)$ со свободной вещественной переменной ϕ представляют множество точек на плоскости $0xy$ —окружность с центром $(0, 0)$ и радиусом 5. При этом, как правило, подразумевают тот или иной тип у используемых параметров, то есть приписывают им то или иное множество допустимых значений—значением n может быть произвольное целое число, значением ϕ может быть произвольное вещественное число и т.п. Если изобразительных средств выражений со свободными переменными не хватает для представления некоторого множества, то часто прибегают к использованию дополнительных ограничений на свободные переменные:

- $2n + 1$, где $n \geq 0$ —множество натуральных нечетных чисел;
- $(5 \cos \phi, 5 \sin \phi)$, где $0 \leq \phi \leq \frac{\pi}{2}$ —четверть окружности.

Методы, аналогичные рассмотренным выше, и будут использованы как изобразительные средства для представления множеств.

2.1 С-переменные, с-выражения

Определение 4

Для построения метавычислений над языком TSG введем два типа *конфигурационных переменных*—*с-переменных*:

- *са-переменные*—допустимыми значениями которых являются а-значения;
- *се-переменные*—допустимыми значениями которых являются е-значения.

Синтаксис с-переменных представлен на рисунке 2.1.

```

ca-var      ::= (CVA c-var-name)

ce-var      ::= (CVE c-var-name)

c-var-name ::= integer

ca-expr     ::= a-val
                | ca-var

ce-expr     ::= ca-expr
                | ce-var
                | (CONS ce-expr1 ce-expr2)

```

Рис. 2.1: Синтаксис с-переменных и с-выражений

В процессе метавычислений иногда будет необходимо строить *уникальные* с-переменные—заведомо не совпадающие с ранее использованными с-переменными. Построение таких уникальных с-переменных обеспечивается следующим:

- В качестве имен (индексов) с-переменных используются натуральные числа.
- В алгоритмах метавычислений поддерживается специальная целочисленная переменная $i :: \text{FreeIdx}$ (свободный индекс с-переменных), значение которой больше, чем индекс любой из уже построенных с-переменных. При построении очередной уникальной с-переменной в качестве ее индекса используется значение i , затем значение i увеличивается на 1.

Будем использовать следующие сокращения:

- $\mathcal{A}.index$ —сокращенная запись са-переменной (CVA *index*);
- $\mathcal{E}.index$ —сокращенная запись се-переменной (CVE *index*);

Определение 5

С-выражениями (са-выражениями, се-выражениями) будем называть выражения, построенные из атомов, с-переменных и конструкторов CONS в соответствии с правилами, приведенными на рисунке 2.1.

2.2 С-связи, с-среды и с-состояния

Процесс $p \ d \xRightarrow{*} res$ выполнения программы p над данными d описан в разделе 1.3 как последовательность переходов от одного *состояния вычисления* к другому:

$$s_0 \Rightarrow s_1 \Rightarrow \dots,$$

где $s_i = (t_i, e_i)$, t_i —вычисляемый терм, а e_i —среда, $i = 0, 1, \dots$

Состояние вычисления $s=(t,e)$ описывает, какой терм t вычисляется в данный момент и какие *конкретные значения* имеют программные переменные, входящие в терм t :

$$e = [\text{var}_1 := \text{val}_1, \dots, \text{var}_n := \text{val}_n]$$

В метавычислениях имеют дело с рассмотрением вычисления программы не над конкретными данными, а над множеством данных, представленным выражением с с-переменными. Поэтому состояние такого обобщенного вычисления надо представлять синтаксической конструкцией, во всем аналогичной обычному состоянию вычисления, но содержащей выражения с с-переменными—с-выражения—*в тех местах, где в состоянии обычного вычисления находятся конкретные данные*, обрабатываемые программой. То есть, состояние обобщенного вычисления должно быть представлено следующим образом (здесь подчеркнуты обрабатываемые программой данные):

$$\begin{aligned} \text{состояние:} & \quad (t, [\text{var}_1 := \underline{\text{val}}_1, \dots, \text{var}_n := \underline{\text{val}}_n]) \\ \text{обобщенное состояние:} & \quad (t, [\text{var}_1 := \underline{\text{cexp}}_1, \dots, \text{var}_n := \underline{\text{cexp}}_n]) \end{aligned}$$

```

c-state    ::= (term, c-env)

c-env      ::= [ c-binding1, ..., c-bindingn ] -- n ≥ 0

c-binding ::= a-var := ca-expr
           | e-var := ce-expr

```

Рис. 2.2: Синтаксис с-состояния, с-среды и с-связи

Определение 6

Будем называть *с-средой* список *с-связей*—упорядоченных пар вида $\text{var} := \text{cexp}$, связывающих программные переменные— var —с с-выражениями cexp , при этом а-переменные могут быть связаны только с са-выражениями, е-переменные могут быть связаны с любыми с-выражениями.

Будем называть *с-состоянием* упорядоченную пару $(\text{term}, \text{cenv})$, где term — программный терм, cenv — с-среда, связывающая программные переменные из term с с-выражениями.

Точное определение синтаксиса с-состояния, с-среды и с-связи дано на рисунке 2.2.

2.3 Рестрикции с-переменных

По умолчанию допустимыми значениями са-переменных и се-переменных являются, соответственно, произвольные а-значения и произвольные е-значения (атомы и S-выражения). *Рестрикции* позволяют наложить дополнительные ограничения на значения с-переменных. Способы задания ограничений—язык рестрикций—как и остальные понятия, связанные с представлением множеств, выбираются в соответствии с выбранным языком реализации.

Определение 7

Для TSG рестрикции задаются в виде системы неравенств между са-переменными и атомами. В общем случае система неравенств изображается в виде списка пар одного из следующих видов:

$$\begin{aligned} A.index_1 \neq A.index_2, \quad A.index \neq 'atom, \\ 'atom \neq A.index, \quad 'atom_1 \neq 'atom_2. \end{aligned}$$

То есть, левая и правая части неравенства—са-выражение: атом или са-переменная. Рестрикции “допускают” такие значения са-переменных, которые не противоречат данной системе неравенств—после подстановки значений в рестрикцию не возникает противоречий. Таким образом, рестрикции позволяют запретить для некоторых са-переменных принимать значения, совпадающие с некоторыми атомами или запретить двум са-переменным принимать равные значения.

Для несовместных¹ рестрикций предусмотрено особое представление:

INCONSISTENT.

Точное описание синтаксиса рестрикций приведено на рисунке 2.3. Для конструктора (\neq) будем допускать альтернативное изображение (\neq или \neq).

```

restr ::= INCONSISTENT
       | RESTR [ ineq1, ..., ineqn ] -- n ≥ 0

ineq  ::= ca-expr ≠ ca-expr

```

Рис. 2.3: Рестрикции с-переменных

Пример 1

Рестрикция $RESTR [A.1 \neq 'A, A.2 \neq 'C, A.1 \neq A.2]$ “запрещает” с-переменным $A.1$, $A.2$ и $A.3$ принимать такие значения, при которых $A.1 = 'A$ или $A.2 = 'C$ или $A.1 = A.2$.

Определение 8

Будем использовать следующие термины:

- *тавтология*—неравенство, выполняемое при любых значениях с-переменных: $'atom_1 \neq 'atom_2$, где $'atom_1$, $'atom_2$ —два различных атома;
- *противоречие*—неравенство, невыполнимое ни при каких значениях с-переменных—неравенство, в котором левая и правая части совпадают: $'atom \neq 'atom$ или $A.index \neq A.index$.

¹Разумеется, рестрикция может быть несовместной—запрещать любые значения для са-переменных. Например, такое возможно, если рестрикция содержит неравенство-противоречие “ $A.1 \neq A.1$ ”.

Ниже описаны функции `isContradiction` и `isTautology`, проверяющие, является ли неравенство противоречием или тавтологией:

```
isContradiction, isTautology :: InEq -> Bool
isContradiction (left  :=/=: right) = (left == right)

isTautology      (ATOM a :=/=: ATOM b) = (a /= b)
isTautology      (left  :=/=: right)   = False
```

Для неравенств оператор (`/=`) определим так, чтобы учитывалось, что неравенство по своему смыслу является неупорядоченной парой:

```
instance Eq InEq where
  (l1:=/:r1) == (l2:=/:r2) | (l1==l2) && (r1==r2) = True
                           | (l1==r2) && (r1==l2) = True
                           | otherwise           = False
```

Так как рестрикция—система неравенств, то следующие преобразования являются эквивалентными преобразованиями, упрощающими рестрикцию: приведение рестрикции к виду `INCONSISTENT`, если в рестрикции имеется хоть одно противоречие; удаление повторных неравенств²; удаление тавтологий. Ниже определена функция `cleanRestr`, выполняющая данные упрощения рестрикции:

```
cleanRestr :: Restr -> Restr
cleanRestr INCONSISTENT = INCONSISTENT
cleanRestr (RESTR ineqs) = INCONSISTENT, if any isContradiction ineqs
cleanRestr (RESTR ineqs) = RESTR (nub(filter (not.isTautology) ineqs))
```

Утверждение 1

Пусть `rs`—рестрикция. Тогда `cleanRestr rs`—рестрикция.

Доказательство

Следует из определения функции `cleanRestr` и из определения 7.

Ниже приводится описание оператора (`+`), объединяющего две рестрикции. При этом рестрикции рассматриваются как системы неравенств—если хотя бы одна рестрикция несовместна, то и вместе они несовместны. В противном случае системы объединяются, после чего убираются повторные неравенства при помощи функции `cleanRestr`:

```
instance UPDATE Restr where
  INCONSISTENT +. _           = INCONSISTENT
  _             +. INCONSISTENT = INCONSISTENT
  (RESTR r1)    +. (RESTR r2)  = cleanRestr (RESTR(r1++r2))
```

² Для этого будет использована стандартная функция `nub`, удаляющая повторные элементы из списка. В силу определения оператора (`==`) для неравенств, функция `nub` будет сравнивать неравенства как неупорядоченные пары.

2.4 С-конструкции

Выше определены несколько синтаксических конструкций, в которые могут входить с-переменные—*с-конструкции*. В данном разделе: определяются дополнительные термины для с-конструкций; дается обзор синтаксиса с-конструкций (см. рисунок 2.4); вводятся понятия *SR-база* и *SR-выражение*³.

Определение 9

Будем использовать общий термин “*с-конструкция*” для с-выражений, списков с-выражений, с-сред, с-состояний, неравенств, рестрикций и SR-выражения. Ниже перечислены все правила построения с-конструкций и определены термины *SR-база* и *SR-выражение*:

1. *С-выражения*: атомы, с-переменные и $(\text{CONS } se_1 \ se_2)$, где se_1, se_2 —с-выражения.
2. *SR-базы*:
 - (a) с-выражения se ;
 - (b) списки с-выражений $[se_1, \dots, se_n]$;
 - (c) с-связи $pvar:=se$;
 - (d) с-среды $[cbind_1, \dots, cbind_n]$;
 - (e) с-состояния $(term, cenv)$;

где se, se_1, \dots, se_n —с-выражения, $n \geq 0$, $pvar$ —программная переменная, $term$ —программный терм TSG, $cbind_1, \dots, cbind_n$ —с-связи, cnv —с-среда;
3. *Неравенства* $ca_1 \neq ca_2$, где ca_1, ca_2 —с-выражения;
4. *Рестрикции*: INCONSISTENT и $\text{RESTR } ineqs$, где $ineqs$ —список неравенств;
5. *SR-выражения*: (sx, rs) , где sx —SR-база, rs —рестрикция.

Для любой с-конструкции sx список с-переменных, входящих в sx , мы будем обозначать $cvars \ sx$ ⁴. Непосредственно из определения синтаксиса SR-баз следует, что если в SR-базе нет ни одной с-переменной, то она является соответствующим объектом языка TSG.

Утверждение 2

Пусть sx —SR-база и $cvars \ sx = []$. Тогда:

1. если sx —с-выражение, то sx —а-значение: *'atom*;
2. если sx —с-выражение, то sx —е-значение;
3. если sx —список с-выражений, то sx —список е-значений;
4. если sx —с-связь, то sx —связь;
5. если sx —с-среда, то sx —среда;
6. если sx —с-состояние, то sx —состояние.

³ Префикс “SR-” от английского “set representation”.

⁴ Формальное определение функции $cvars$ приведено в [FTP].

$'atom$	$c\text{-выражение}$	}	$SR\text{-база}$
$cvar$	ce		
$(CONS\ ce_1\ ce_2)$			
$[ce_1, \dots, ce_n]$	$список$ $c\text{-выражений}$ ces		
$pvar:=ce$	$c\text{-связь}$ $cbind$		
$[cbind_1, \dots, cbind_n]$	$c\text{-среда}$ $cenv$	}	cx
$(term, cenv)$	$c\text{-состояние}$ cst		
<hr/>			
$ce_1 \neq ce_2$	$неравенство$ $ineq$		
$[ineq_1, \dots, ineq_n]$	$список$ $неравенств$ $ineqs$		
$RESTR\ ineqs$	$рестрикция$		
$INCONSISTENT$	rs		
<hr/>			
(cx, rs)	$SR\text{-выражение}$		

Рис. 2.4: C-конструкции

2.5 Подстановки

В математике, для получения элементов множества, представленного в виде выражения с параметрами, вместо параметров подставляют их некоторые допустимые значения. Например, пусть множество нечетных чисел представлено при помощи выражения $2n + 1$ с целочисленным параметром n . Тогда, подставив вместо n допустимое значение, например, $n = 7$, мы получим элемент из представляемого выражением множества: $(2n + 1) /. [n \rightarrow 7] = 15$. Кроме того, часто используют подстановку вместо параметров не конкретных значений, а выражений с новыми параметрами. В этом случае получают не элементы изображаемого множества, а его подмножества. Например, две подстановки $[n \rightarrow 2k]$ и $[n \rightarrow 2k + 1]$ позволяют построить два подмножества (разбиение) исходного множества:

- $(2n + 1) \text{ /. } [n \rightarrow 2k] = 4k + 1;$
- $(2n + 1) \text{ /. } [n \rightarrow 2k + 1] = 4k + 3;$

Аналогичную роль (построение элементов множеств и построение подмножеств) будут играть подстановки и в методах представлений множеств при помощи с-конструкций.

Определение 10

Подстановкой будем называть конечный (возможно пустой) список пар вида

$$subst = [cvar_1 :-> cexpr_1, \dots, cvar_n :-> cexpr_n]$$

каждая пара в котором связывает некоторую с-переменную $cvar_i$ с ее значением $cexpr_i$ и весь список удовлетворяет следующим ограничениям:

- синтаксис $subst$ соответствует приведенному на рисунке 2.5 описанию;
- в левых частях пар перечислены *различные* с-переменные.

```
subst      ::= [ c-sbind1, ..., c-sbindn ]  -- n ≥ 0

c-sbind    ::= ca-var :-> ca-expr
              | ce-var :-> ce-expr
```

Рис. 2.5: Подстановка

Будем использовать обозначение $\text{dom } subst$ для списка всех с-переменных, которым данная подстановка ставит в соответствие некоторое значение:

```
dom :: Subst -> [CExp]
dom subst = [ cvar | (cvar :-> _ ) <- subst ]
```

Подстановки используют для замены вхождений с-переменных в с-конструкции на их значения из подстановки. Обозначим через $(/.)$ оператор применения подстановки к с-конструкции. Для случая применения подстановки к с-выражению, определим его следующим образом:

```
instance APPLY CExp Subst where
  (ATOM a) /.s = ATOM a
  (CONS h t)/.s = CONS (h/.s) (t/.s)
  cvar      /.s = cvar,    if cvar 'notElem' dom s
  cvar      /.s = head[ cexp | (cv:->cexp) <- s, cv==cvar ]
```

То есть, применение подстановки $subst$ к с-выражению ce состоит в следующем:

- в ce конструкторы `CONS`, атомы, а также с-переменные, не входящие в $\text{dom } subst$, остаются неизменными (см. первые три предложения определения оператора $(/.)$);

- все с-переменные в `ce`, входящие в `dom subst`, заменяются на их значения из `subst` (см. последнее предложение определения оператора `(/.).`).

Естественным образом расширим определение оператора `(/.).` на случаи применения подстановки к неравенству, с-связи, списку с-конструкций, паре с-конструкций и к рестрикции:

```
instance APPLY InEq Subst where
  (l:=/:r) /. subst = (l/.subst) :=/: (r/.subst)

instance APPLY CBind Subst where
  (pvar := cexpr) /. subst = pvar := (cexpr/.subst)

instance APPLY a subst => APPLY [a] subst where
  cxs /. subst = map (/.subst) cxs

instance (APPLY a subst, APPLY b subst) => APPLY (a,b) subst where
  (ax,bx) /. subst = ( ax/.subst ) , (bx/.subst) )

instance APPLY Restr Subst where
  INCONSISTENT /. subst= INCONSISTENT
  (RESTR ineqs) /. subst= cleanRestr(RESTR(ineqs/.subst))
```

Итак, во всех случаях применение подстановки `subst` к с-конструкциям сводится к поиску в с-конструкциях тех с-переменных, которые входят в `dom subst`, и замене их на их значение из подстановки. При этом конструкторы, атомы, а также с-переменные, не входящие в `dom subst`, остаются неизменными. После применения подстановки к рестрикции, результат упрощается при помощи функции `cleanRestr`.

Пример 2

Рассмотрим применения подстановок к рестрикции. Пусть:

```
ineqs = [ A.2:≠:A.1, A.2:≠:'C, A.3:≠:'A ]
rs = RESTR ineqs
subst1 = [ A.1:->'A, A.2:->A.3 ]
subst2 = [ A.1:->'B, A.3:->'B ]
subst3 = [ A.2:->A.1, A.3:->'A ]
```

Тогда, будем иметь следующие результаты применения подстановок:

```
ineqs /. subst1 = [A.3:≠:'A, A.3:≠:'C, A.3:≠:'A]
rs /. subst1 = RESTR [A.3:≠:'A, A.3:≠:'C]
ineqs /. subst2 = [A.2:≠:'B, A.2:≠:'C, 'B:≠:'A]
rs /. subst2 = RESTR [A.2:≠:'B, A.2:≠:'C]
ineqs /. subst3 = [A.1:≠:A.1, A.1:≠:'C, 'A:≠:'A]
rs /. subst3 = INCONSISTENT
```

2.5.1 Свойства подстановок

Из определения оператора $(/.)$ и из определений 5 и 10 следует справедливость следующего утверждения.

Утверждение 3

1. Пусть ce —с-выражение, $subst$ —подстановка. Тогда $ce/.subst$ —с-выражение.
2. Пусть ce —са-выражение, $subst$ —подстановка. Тогда $ce/.subst$ —са-выражение.

Из определения оператора $(/.)$, определения 10, утверждений 1 и 3 следует справедливость следующего утверждения:

Утверждение 4

Пусть s —подстановка, sx —с-конструкция одного из следующих видов: са-выражение, се-выражение, список с-выражений длины n , с-связь, с-среда, с-состояние, неравенство, противоречие, тавтология, список неравенств длины n , рестрикция, SR-выражение. Тогда, $sx/.s$ является с-конструкцией *того же вида*.

Следующее свойство подстановок следует из “жесткости” конструкторов, при помощи которых из атомов и с-переменных конструируются SR-базы. Здесь под жесткостью n -местного конструктора s понимается следующее свойство:

$$(s \ x_1 \ \dots \ x_n) == (s \ y_1 \ \dots \ y_n) \iff \bigwedge_{i=1}^n (x_i == y_i)$$

Утверждение 5

Если sx —SR-база, s_1, s_2 —подстановки, то $sx/.s_1 == sx/.s_2$ тогда и только тогда, когда для любой с-переменной cv из sx выполнено $cv/.s_1 == cv/.s_2$:

$$(sx/.s_1 == sx/.s_2) \iff \bigwedge_{cv \in cvars \ sx} (cv/.s_1 == cv/.s_2)$$

Доказательство утверждения 5 приведено в [FTP].

2.6 Отождествление с-выражений

Как показано в утверждении 5, если список с-выражений $ces2$ является результатом применения некоторой подстановки к списку с-выражений $ces1$, то существует единственный набор значений с-переменных из $ces1$ такой, что при подстановке в $ces1$ вместо с-переменных их значений мы получим $ces2$. В данном разделе будет предъявлен простой алгоритм отождествления **unify**, который по двум спискам с-выражений $ces1$ и $ces2$ строит следующий результат:

1. **(False, [])**—если не существует подстановки s такой, что $ces1/.s = ces2$.
2. **(True, s)**—если существует такая подстановка. При этом, s —подстановка, такая, что $ces1/.s = ces2$ и $dom \ s = cvars \ ces1$.

Ниже описан тип результата алгоритма `unify` и введена константа `fail`—для результата, соответствующего первому случаю.

```
type UnifRes = (Bool, Subst)
fail :: UnifRes
fail = (False, [])
```

Введем новые синтаксические конструкции: *клэш*—пара с-выражений $ce_1 := ce_2$; *система клэшей*—список клэшей. Клэш будем рассматривать как уравнение, а список клэшей—как систему уравнений на с-переменные из левых частей клэшей. Требуется подобрать, если это возможно, такие значения с-переменных, чтобы при подстановке их в левые части клэшей левые части совпали с правыми. Решением системы клэшей является подстановка, связывающая с-переменные с указанными значениями.

Алгоритм `unify` заменяет задачу поиска подстановки s , такой, что $ces1/.s = ces2$ на эквивалентную задачу решения системы клэшей и далее, эквивалентными преобразованиями упрощает систему, пока не будет найдено решение, или не будет обнаружена несовместность системы. Текст алгоритма `unify` приведен на рисунке 2.6.

В функции `unify` начальные аргументы:

```
ces1 = [ce11, ..., ce1i1]
ces2 = [ce21, ..., ce2i2]
```

проверяются на совпадение их длин—если $i1 \neq i2$, то (в силу утверждения 4) ни при какой подстановке s значений с-переменных в `ces1` результат `ces1/.s` не совпадет с `ces2`. В этом случае функция `unify` возвращает `fail` в качестве результата.

Если длины `ces1` и `ces2` совпадают—см. второе предложение функции `unify`,—то вызывается функция `unify'` со следующими аргументами:

```
unify' rchs0 chs0
rchs0 = [ ]
chs0 = [(ce11 := ce21), ..., (ce1i1 := ce2i1)]
```

Заметим, что при этом система клэшей

$$rchs_0 ++ chs_0 = [(ce1_1 := ce2_1), \dots, (ce1_{i1} := ce2_{i1})]$$

эквивалентна исходной задаче. В паре взаимно рекурсивных функций `unify'` и `moveC1` системы $rchs_i ++ chs_i$ преобразуются

$$rchs_0 \ chs_0 \Rightarrow rchs_1 \ chs_1 \Rightarrow rchs_2 \ chs_2 \Rightarrow \dots$$

таким образом, что выполнены следующие утверждения (инвариант цикла):

1. число знаков в записи системы chs_k строго убывает, что обеспечивает терминируемость алгоритма;
2. система $rchs_{k+1} ++ chs_{k+1}$ эквивалентна системе $rchs_k ++ chs_k$;
3. в системе клэшей $rchs_k$ присутствуют только *разрешенные* клэши: $cvar := cexp$, где $cvar$ —с-переменные из `ces1`, $cexp$ —с-выражение, причем, если $cvar$ —са-переменная, то $cexp$ —са-выражение;

```

unify :: CExps -> CExps -> UnifRes
unify ces1 ces2
  | (length ces1) /= (length ces2) = fail
  | otherwise                       = unify' [ ] chs
                                   where chs = zipWith(\a b->(a==:b))ces1 ces2

unify' :: Clashes -> Clashes -> UnifRes
unify' rchs [] = (True, subst)
               where subst = (map (\(a==:b)->(a->b)) rchs)
unify' rchs chs@(ch:chs') =
  case ch of
    ATOM a      :=:ATOM b | a==b -> unify' rchs chs'
    ATOM a      :=:cex      -> fail
    cvar@(CVA _) :=:caex@(ATOM _) -> moveCl rchs chs
    cvar@(CVA _) :=:caex@(CVA _) -> moveCl rchs chs
    cvar@(CVA _) :=:cex      -> fail
    cvar@(CVE _) :=:cex      -> moveCl rchs chs
    CONS a1 b1  :=:CONS a2 b2 -> unify' rchs (p++chs')
                                   where p=[a1==:a2,b1==:b2]
    CONS a1 b1  :=:cex      -> fail

moveCl :: Clashes -> Clashes -> UnifRes
moveCl rchs chs@(ch@(cvar:=:cexp):chs') =
  case [ y | (x:=:y)<-rchs, x==cvar ] of
    [ ]      -> unify' (ch:rchs) chs'
    [y] | y==cexp -> unify' rchs      chs'
    [y] | otherwise -> fail

```

Рис. 2.6: Алгоритм отождествления списков с-выражений

4. в системе клэшей \mathbf{rchs}_k все левые части различны.

Справедливость инварианта очевидна из текста алгоритма `unif`.

Алгоритм завершается, когда обнаружена несовместность системы $\mathbf{rchs}_k++\mathbf{chs}_k$ или когда система клэшей \mathbf{chs}_k становится пустой—см. первое предложение функции `unify'`. В этом случае система \mathbf{rchs}_k эквивалентна исходной задаче, и для данной системы искомая подстановка \mathbf{subs} определяется очевидным образом:

$$\begin{aligned}
 \mathbf{rchs}_k &= [\mathbf{var}_1 :=: \mathbf{cex}_1, \dots, \mathbf{var}_n :=: \mathbf{cex}_n] \\
 \mathbf{subs} &= [\mathbf{var}_1 \rightarrow \mathbf{cex}_1, \dots, \mathbf{var}_n \rightarrow \mathbf{cex}_n] \text{ , то есть} \\
 \mathbf{subs} &= \text{map } (\backslash(a :=: b) \rightarrow (a \rightarrow b)) \mathbf{rchs}_k
 \end{aligned}$$

В силу справедливости инварианта, \mathbf{subs} —правильная подстановка—левые части без повторов, са-переменные связаны с са-выражениями.

Таким образом, имеет место следующее утверждение (более подробное доказательство приведено в [FTP]):

Утверждение 6

Алгоритм `unify` по любым двум спискам *c*-выражений `ces1` и `ces2`, за конечное число шагов строит следующий результат:

1. `(False, [])`—если не существует подстановки *s* такой, что `ces1/.s=ces2`.
2. `(True, s)`—если существует такая подстановка. При этом, *s*—подстановка, такая, что `ces1/.s=ces2` и `dom s=cvars ces1`.

2.7 Представляемое множество

Определение 11

Подстановку *s* будем называть *полной подстановкой для c-конструкции cx*, если она связывает:

- все *c*-переменные из *cx*: `cvars cx ⊆ dom s`;
- с *a*-, *e*-значениями, то есть для любой *c*-переменной *cv* из *cx*, `cv/.s` не содержит *c*-переменных:

$$\forall cv \in (\text{cvars } cx) \quad (\text{val} = cv /. s) \Rightarrow \text{val} \in \text{Eval}$$

Пусть `cx=(cb,rs)`—SR-выражение. Тогда подстановку *s* будем называть *допустимой подстановкой для cx*, если применение *s* к `rs` не приводит к противоречиям: `(rs/.s) /= INCONSISTENT`.

Утверждение 7

Пусть `cx`, `cy`—SR-базы, *s*—полная подстановка для `cx`. Тогда

1. если `(cvars cy) ⊆ (cvars cx)`, то *s*—полная подстановка для `cy`;
2. если `cy`—подконструкция `cx`, то есть `cx` имеет вид `...cy...`, то *s*—полная подстановка для `cy`.

Доказательство

Непосредственно следует из определения 11 и определения функции `cvars`. ■

Утверждение 8

Пусть `cx`—SR-база, *s*—полная подстановка для `cx`. Тогда:

1. если `cx`—*a*-выражение, то `cx/.s`—*a*-значение: `'atom`;
2. если `cx`—*e*-выражение, то `cx/.s`—*e*-значение;
3. если `cx`—список *c*-выражений, то `cx/.s`—список *e*-значений, той же длины, что и список `cx`;
4. если `cx`—*c*-связь, то `cx/.s`—связь;
5. если `cx`—*c*-среда, то `cx/.s`—среда;

6. если cx —с-состояние, то $cx/.s$ —состояние.

Доказательство

Следует из того, что $cx/.s$ не содержит с-переменных (по определению 11) и из утверждений 2 и 4. ■

Пусть (cx, rs) —SR-выражение. Рассмотрим полную подстановку s для (cx, rs) . Применим s к (cx, rs) :

$$(cx, rs)/.s = ((cx/.s), (rs/.s)) = (x, rs')$$

Заметим, что $x=cx/.s$ и $rs'=rs/.s$ не содержат с-переменных. В силу утверждения 8 x является синтаксическим объектом языка TSG. В рестрикции rs' нет с-переменных, то есть не может быть неравенств с с-переменными. Неравенства, не содержащие с-переменные (см. определение 8), могут быть только противоречиями или тавтологиями. Таким образом,

- либо $rs/.s = \text{INCONSISTENT}$ и s недопустимая для (cx, rs) ;
- либо $rs/.s = \text{RESTR } []$ и s допустимая для (cx, rs) , а все неравенства из rs при применении s стали тавтологиями.

Доказано следующее утверждение.

Утверждение 9

Пусть (cx, rs) — SR-выражение, s — полная подстановка для (cx, rs) и

$$(cx, rs)/.s = (cx/.s, rs/.s) = (x, rs').$$

Тогда:

1. s —допустимая для (cx, rs) тогда и только тогда, когда $rs'=\text{RESTR } []$;
2. s —недопустимая для (cx, rs) , тогда и только тогда, когда $rs'=\text{INCONSISTENT}$;
3. $x=cx/.s$ является синтаксической конструкцией TSG, соответствующей виду SR-выражения cx (см. утверждение 8).

Определение 12

Пусть $se=(cx, rs)$ —SR-выражение. Множество полных допустимых для se подстановок обозначим через $FVS(se)=FVS(cx, rs)$.

Будем использовать se для представления множества синтаксических объектов языка TSG. Это множество будем обозначать $\langle se \rangle$ и $\langle cx, rs \rangle$. Определим это множество следующим образом:

$$\langle se \rangle = \langle cx, rs \rangle = \{ cx/.s \mid s \in FVS(cx, rs) \}$$

Смысл данного определения в следующем: (cx, rs) представляет множество

$$\langle cx, rs \rangle,$$

а именно cx —играет роль выражения с параметрами, а rs —дополнительных ограничений на значения параметров. Все элементы множества $\langle cx, rs \rangle$ (и только они) могут быть получены путем замены s -переменных (параметров) в cx на e -значения. При этом значения s -переменных должны соответствовать типам (“ a ” и “ e ”) s -переменных и ограничениям rs —при подстановке значений в ограничения все они должны быть выполнены (перейти в тавтологии). Непосредственно из утверждения 9 следует справедливость следующего утверждения, обеспечивающего корректность определения 12.

Утверждение 10

Пусть (cx, rs) —SR-выражение. Тогда $\langle cx, rs \rangle$ является:

1. множеством a -значений $\langle cx, rs \rangle \subseteq AVal$, если cx — sa -выражение;
2. множеством e -значений $\langle cx, rs \rangle \subseteq EVal$, если cx — se -выражение;
3. множеством списков (длины n) e -значений $\langle cx, rs \rangle \subseteq [EVal]$, если cx —список длины n s -выражений;
4. множеством связей, если cx — s -связь;
5. множеством сред, если cx — s -среда;
6. множеством состояний, если cx — s -состояние.

2.7.1 Представления множеств e -значений

В силу утверждения 10 множества e -значений⁵ могут быть представлены при помощи SR-выражения (se, rs) , где se — s -выражение.

Пример 3

Рассмотрим примеры представления множеств e -значений:

1. $(\mathcal{E}.1, RESTR[])$ представляет множество $EVal$ всех e -значений.
2. $(\mathcal{A}.1, RESTR[])$ представляет множество $AVal$ всех a -значений—' $atom$ '.
3. $\langle (CONS \ 'A \ \mathcal{E}.1), RESTR[] \rangle$ —множество всех e -значений, имеющих вид

$$(CONS \ 'A \ e_1),$$

где e_1 —произвольное e -значение.

4. $\langle (CONS \ \mathcal{A}.1 \ \mathcal{E}.2), RESTR [\mathcal{A}.1:\neq:\ 'A] \rangle$ —множество всех e -значений, имеющих вид $(CONS \ a_1 \ e_2)$, где a_1 —произвольный атом, отличный от ' A ', а e_2 —произвольное e -значение.

5. Пусть $e \in EVal$. Тогда, $\langle e, RESTR[] \rangle$ —одноэлементное множество:

$$\langle e, RESTR[] \rangle = \{e\}.$$

6. Пусть se —любое s -выражение. Тогда, $\langle se, INCONSISTENT \rangle = \emptyset$.

⁵Напомним, что “ e -значение” и “ S -выражение”—синонимы, и $EVal$ —множество всех S -выражений.

2.7.2 Классы и L-классы

Определение 13

Пусть $сх$ —SR-база. Будем называть $сх$ *линейной*, если в $сх$ нет се-переменных, имеющих повторные вхождения в $сх$. Линейные $с$ -выражения будем называть *L-выражениями*, линейные списки $с$ -выражений будем называть *L-списками с-выражений*.

Из определения 13 непосредственно следует утверждение 11.

Утверждение 11

Пусть $сх, су$ —SR-базы, $сх$ —линейная и $су$ —подконструкция $сх$, то есть $сх$ имеет вид: $сх = \dots су \dots$. Тогда $су$ —линейная.

Определение 14

Пусть $сес$ —список $с$ -выражений, rs —рестрикция. Тогда SR-выражение $C = (сес, rs)$ будем называть *классом*. Если $сес$ —L-список $с$ -выражений, то класс $C = (сес, rs)$ будем называть *L-классом*.

Из утверждения 10 и определения 14 следует утверждение 12.

Утверждение 12

Пусть $C = (сес, rs)$ —класс, $length\ сес = n$. Тогда каждый элемент множества $\langle C \rangle$ является списком из n e -значений и $\langle C \rangle \subseteq [EVal]$.

Вспомним, что $[EVal]$ —предметная область языка TSG. Таким образом, классы могут использоваться для изображения множеств входных данных программ из TSG.

Пример 4

Рассмотрим класс:

$$C = ([A.1, (CONS\ \mathcal{E}.2\ A.3)],\ RESTR[A.1:\neq:A.3])$$

Тогда C представляет множество всех данных, имеющих вид

$$[a_1, (CONS\ e_2\ a_3)],$$

где e_2 —произвольное e -значение, a_1 и a_3 —два несовпадающих a -значения.

Определение 15

Пусть $d \in D = [EVal]$ — список e -значений. Тогда L-класс $(d, RESTR[])$ обозначим через $sngl(d)$.

Утверждение 13

Пусть $d \in D = [EVal]$. Тогда $\langle sngl(d) \rangle = \{d\}$.

Доказательство

Так как $cvars(sngl\ d) = []$, то в силу утверждения 5 и определения оператора $(/.)$ для любой подстановки s выполнено:

$$sngl(d) /. s = (d, RESTR[]) /. s = (d, RESTR[]) /. [] = (d, RESTR[])$$

Следовательно, по определению 12, $\langle sngl(d) \rangle = \{d\}$. ■

2.7.3 Конфигурации

Определение 16

Будем называть *конфигурацией* SR-выражение conf , имеющее вид:

$$\text{conf} = ((t, \text{cenv}), \text{rs}),$$

где (t, cenv) —с-состояние, rs —рестрикция, t —программный терм, cenv —с-среда.

Формальное доказательство следующего (достаточно очевидного) утверждения можно найти в [FTP].

Утверждение 14

Пусть $\text{conf} = ((t, \text{cenv}), \text{rs})$ —конфигурация. Тогда $\langle \text{conf} \rangle = \langle (t, \text{cenv}), \text{rs} \rangle$ —множество состояний вычислений и $\langle \text{conf} \rangle = \{ (t, \text{env}) \mid \text{env} \in \langle \text{cenv}, \text{rs} \rangle \}$.

Таким образом, определены средства для представления множеств сред (с-среды) и множеств состояний вычислений (конфигурации) для языка TSG. Конечно, не всякое множество состояний вычисления можно представить в виде конфигурации. Однако, ниже будет видно, что введенных средств достаточно для представления всех множеств, которые возникают при метавычислениях.

2.8 Суперпозиция подстановок

Определение 17

Пусть sa, sb —подстановки. Обозначим через $\text{sa} \cdot \text{sb}$ список пар вида

$$c\text{-var} \rightarrow c\text{-expr},$$

получаемый следующим образом: если cvar является элементом $\text{dom } \text{sa}$ или $\text{dom } \text{sb}$, то в список $\text{sa} \cdot \text{sb}$ входит пара $\text{cvar} \rightarrow ((\text{cvar} / \text{sa}) / \text{sb})$. Точное определение оператора \cdot приведено ниже.

```
(.*) :: Subst -> Subst -> Subst
sa . sb = [ cvar -> ((cvar / sa) / sb) | cvar <- dom_sa_sb ]
      where
        dom_sa_sb = nub ((dom sa) ++ (dom sb))
        -- объединение без повторов dom sa и dom sb
```

Утверждение 15

Пусть sa, sb —подстановки. Тогда $\text{s} = \text{sa} \cdot \text{sb}$ —подстановка.

Доказательство (несложная проверка всех требований определения 10) утверждения 15 приведено в [FTP].

Утверждение 16

Пусть $\text{sa}, \text{sb}, \text{s} = \text{sa} \cdot \text{sb}$ —подстановки. Тогда для произвольной с-конструкции cx выполнено: $\text{cx} / \text{s} = (\text{cx} / \text{sa}) / \text{sb}$.

Доказательство (использующее индукцию по структуре cx) утверждения 16 приведено в [FTP].

Определение 18

Пусть sa, sb —подстановки. Тогда подстановку $\text{s} = \text{sa} \cdot \text{sb}$ будем называть *суперпозицией подстановок* sa и sb .

2.9 Сужения и разбиения множеств

Пусть cx —SR-выражение. В данном разделе будут введены следующие понятия: *сужения*, позволяющие получать представления cx' подмножеств $\langle cx \rangle$, и *разбиения*, позволяющие получать представления непересекающихся подмножеств— cx' и cx'' —множества $\langle cx \rangle$, объединение которых совпадает с исходным множеством.

В разделе 2.5 упоминалось, что подстановка вместо s -переменных s -выражений позволяет получать подмножества исходного множества. Вторым очевидным способом сужения—усиление рестрикций, добавление новых ограничений. В метавычислениях будут использованы оба этих способа.

2.9.1 Сужения

Определение 19

Пусть s —подстановка, r —рестрикция. Тогда *сужением* $contr$ будем называть следующие конструкции: $(S\ s)$ и $(R\ r)$. Описание синтаксиса сужения приведено на рисунке 2.7

$$\begin{array}{lcl} contr & ::= & S\ subst \\ & | & R\ restr \end{array}$$

Рис. 2.7: Синтаксис сужения для языка TSG

Применение сужения $contr$ к (cx, rs) заключается в:

- применении к (cx, rs) подстановки s , если $contr$ имеет вид $S\ s$;
- добавлении к рестрикциям rs рестрикции r , если $contr$ имеет вид $R\ r$.

Ниже приведено точное определение оператора $(/.)$ для вычисления результата применения сужения $contr$ к (cx, rs) .

```
instance APPLY c Subst => APPLY (c, Restr) Contr where
  (cx, rs) /. (S subst) = (cx/.subst, rs/.subst)
  (cx, rs) /. (R restr) = (cx, rs+.restr)
```

Утверждение 17

Пусть (cx, rs) —SR-выражение, c —сужение.

Тогда, $\langle cx', rs' \rangle \subseteq \langle cx, rs \rangle$, где $(cx', rs') = (cx, rs) /. c$.

Доказательство

Заметим, что если $rs' = \text{INCONSISTENT}$, то $\langle cx', rs' \rangle = \emptyset$ и утверждение выполнено: $\langle cx', rs' \rangle = \emptyset \subseteq \langle cx, rs \rangle$.

Кроме того, если $rs = \text{INCONSISTENT}$, то $rs' = \text{INCONSISTENT}$, независимо от того, является сужение подстановкой или рестрикцией⁶ и утверждение будет выполнено: $\langle cx', rs' \rangle = \emptyset \subseteq \langle cx, rs \rangle = \emptyset$.

⁶ В силу определения операторов $(/.)$ и $(+.)$ для несовместных рестрикций.

Поэтому, ниже будет рассматриваться случай, когда рестрикции rs' и rs имеют вид $RESTR\ inegs$.

Рассмотрим произвольный $x \in \langle cx', rs' \rangle$. По определению 12, существует полная допустимая для (cx', rs') подстановка s , такая, что $x = cx'/.s$ и $rs'/.s = RESTR\ []$. Для завершения доказательства утверждения необходимо доказать, что $x \in \langle cx, rs \rangle$, то есть, предъявить полную допустимую подстановку s' для (cx, rs) , такую, что $x = cx/.s'$. Рассмотрим два возможных случая для сужения c :

A. $c = S\ sc$. Рассмотрим суперпозицию подстановок $s' = sc.*.s$. Выполнено следующее:

- $cx/.s' = (cx/.sc)/.s = cx'/.s = x$,
что в частности означает, что s' — полная подстановка для cx , а значит и для (cx, rs) . Действительно, в результате $cx/.s' = x$ применения подстановки s' к cx нет s -переменных. Значит, все s -переменные из cx связаны подстановкой s' с e -значениями.
- $rs/.s' = (rs/.sc)/.s = rs'/.s = RESTR\ []$,
то есть, s' допустима для (cx, rs) .

Таким образом, подстановка s' является полной допустимой для (cx, rs) и $x \in \langle cx, rs \rangle$.

B. $c = R\ r$. В этом случае $cx' = cx$, $rs' = rs + .r$ — содержит все неравенства из rs . Подстановка s является полной для (cx, rs) , так как она полная для (cx', rs') . При применении подстановки s к любому неравенству из rs' получается тавтология. Следовательно, при применении s к любому неравенству из rs , результатом будет тавтология.

Значит, подстановка s является полной допустимой для (cx, rs) . И, так как $x = cx'/.s = cx/s$, то $x \in \langle cx, rs \rangle$.

Утверждение 17 доказано. ■

Определение 20

Пусть cx, cx' — SR-выражения. Будем использовать обозначения $cx' \preceq cx$ и $cx \succeq cx'$, если существуют такие сужения c_1, \dots, c_n , $n \geq 0$, что $cx' = cx/.c_1/.c_2 \dots /.c_n$.

Из определения 20 следует справедливость утверждения 18.

Утверждение 18

Отношение \preceq является рефлексивным и транзитивным отношением на множестве SR-выражений.

Из определения 20 и утверждения 17 справедливость утверждения 19.

Утверждение 19

Если $cx' \preceq cx$, то $\langle cx' \rangle \subseteq \langle cx \rangle$, где cx и cx' — SR-выражения.

Утверждение 20

Данное $d \in D$ принадлежит множеству $\langle C \rangle$, представленному классом \mathcal{C} тогда, и только тогда, когда $sngl(d) \preceq C$.

Доказательство

1. Если $d \in \langle C \rangle$, то (определение 12) существует подстановка s , такая, что

$$C/.s=(d, \text{RESTR}[]).$$

То есть, $C/(S \ s)=\text{sngl}(d)$ и $\text{sngl}(d) \preceq C$.

2. Если $\text{sngl}(d) \preceq C$, то (в силу утверждения 19) выполнено $\{d\}=\langle \text{sngl}(d) \rangle \preceq \langle C \rangle$ и $d \in \langle C \rangle$. ■

Определение 21

Введем специальные обозначения— idC , emptC для следующих двух сужений:

$$\begin{aligned} \text{idC}, \text{emptC} &:: \text{Contr} \\ \text{idC} &= S \ [] \\ \text{emptC} &= R \ \text{INCONSISTENT} \end{aligned}$$

Непосредственно из определения оператора $(/.)$ и из определений 19 и 21 следует справедливость следующего утверждения.

Утверждение 21

Пусть (cx, rs) —SR-выражение. Тогда, выполнено:

$$\begin{aligned} (cx, rs)/.\text{idC} &= (cx, rs), \langle (cx, rs)/.\text{idC} \rangle = \langle cx, rs \rangle; \\ (cx, rs)/.\text{emptC} &= (cx, \text{INCONSISTENT}), \langle (cx, rs)/.\text{emptC} \rangle = \emptyset. \end{aligned}$$

Утверждение 22

Если x и x' —SR-выражения, то $x' \preceq x$, тогда и только тогда, когда существуют два сужения: $c_1=(S \ s)$ и $c_2=(R \ r)$, такие, что $x'=x/.c_1/.c_2$.

Доказательство

По определению 20, $x' \preceq x$ тогда и только тогда, когда существуют такие сужения c_1, \dots, c_n , $n \geq 0$, что

$$x' = x/.c_1/.c_2 \dots /.c_n = x/(S \ [])/.c_1/.c_2 \dots /.c_n/(R \ \text{RESTR}[])$$

Суперпозицию сужений: $/(S \ [])/.c_1/.c_2 \dots /.c_n/(R \ \text{RESTR}[])$ всегда можно преобразовать к форме: $/(S \ s)/(R \ r)$ при помощи следующих правил:

$$\begin{aligned} SS &\rightarrow S: & /.(S \ s_1)/.(S \ s_2) &= /.(S \ s_1.*.s_2); \\ RR &\rightarrow R: & /.(R \ r_1)/.(R \ r_2) &= /.(R \ r_1+.r_2); \\ RS &\rightarrow SR: & /.(R \ r)/.(S \ s) &= /.(S \ s)/(R \ (r/.s)). \end{aligned}$$

2.9.2 Каноническая форма класса

Классы являются средством представления множеств данных. Это представление неоднозначно по разным причинам. Одна из причин неоднозначности связана со следующим:

1. с-переменные, входящие в класс, являются *свободными параметрами*, то есть согласованное переименование всех с-переменных по сути не изменяет класса;
2. рестрикция класса—система неравенств, а каждое неравенство—неупорядоченная пара “*левая часть*: \neq :*правая часть*”, то есть перестановка левой и правой части неравенства и переупорядочивание неравенств в рестрикции класса по сути не изменяет класса;

3. в рестрикции неравенства могут входить многократно—преобразования рестрикций, выполняемые функцией `cleanRestr`, по сути, не изменяет класса.

Последнее обстоятельство легко исправимо—достаточно вместо класса \mathcal{C} с повторными неравенствами и/или тавтологиями/противоречиями в рестрикциях рассматривать класс $\mathcal{C}/. []$ —полностью эквивалентный исходному— $\langle \mathcal{C}/. [] \rangle = \langle \mathcal{C} \rangle$,—но не содержащий повторных неравенств, тавтологий и противоречий в рестрикции. Поэтому везде ниже предполагается, что в рестрикциях классов отсутствуют повторные неравенства, тавтологии и противоречия.

Формальное доказательство следующего очевидного утверждения можно найти в [FTR].

Утверждение 23

Пусть $\mathcal{C}_1 = (\text{ces}_1, \mathbf{r}_1)$ и \mathcal{C}_2 —классы. Класс \mathcal{C}_2 получен из \mathcal{C}_1 при помощи:

1. *переупорядочивания рестрикций*: $\mathcal{C}_1^* = (\text{ces}_1, \mathbf{r}'_1)$, где \mathbf{r}'_1 получено из \mathbf{r}_1 при помощи некоторой перестановки левых и правых частей неравенств и переупорядочивания самих неравенств в рестрикции \mathbf{r}_1 ;
2. *переименования с-переменных*—то есть, применения подстановки $\mathbf{s} \leftarrow \mathcal{C}_2 = \mathcal{C}_1^* /. \mathbf{s}$ —такой, что: $(\text{dom } \mathbf{s}) = (\text{cvars } \mathcal{C}_1)$;
любой са-переменной $va \in (\text{cvars } \mathcal{C}_1)$ из класса \mathcal{C}_1 подстановка \mathbf{s} ставит в соответствие са-переменную $va /. \mathbf{s} = va$;
любой се-переменной $va \in (\text{cvars } \mathcal{C}_1)$ из класса \mathcal{C}_1 подстановка \mathbf{s} ставит в соответствие се-переменную $va /. \mathbf{s} = va$;
разным с-переменным из $\text{cvars } \mathcal{C}_1$ подстановка \mathbf{s} ставит в соответствие разные с-переменные.

Тогда $\langle \mathcal{C}_1 \rangle = \langle \mathcal{C}_2 \rangle$.

Определение 22

Определим отношение ($<$) на множестве с-выражений в соответствии с рисунком 2.8.

Неравенства будем сравнивать по лексикографическому порядку:

$$((x_1 : \neq : y_1) < (x_2 : \neq : y_2)) \stackrel{\text{def}}{\iff} (x_1 < x_2) \mid \mid ((x_1 == x_2) \&\& (y_1 < y_2)).$$

Класс \mathcal{C}^* будем называть *канонической формой* класса \mathcal{C} , если:

1. \mathcal{C}^* может быть получен из класса \mathcal{C} путем переименования с-переменных и переупорядочивания неравенств в рестрикции (в соответствии с условиями утверждения 23);
2. $\text{cvars } \mathcal{C}^*$ имеет вид: $[vt_1 \ 1, vt_2 \ 2, \dots, vt_n \ n]$,
где $vt_i \in \{\text{CVA}, \text{CVE}\}$ —конструктор с-переменной. То есть, для с-переменных в \mathcal{C}^* используются в качестве индексов все числа от 1 до n , где n —число различных с-переменных. И если первое вхождение с-переменной v встречается в \mathcal{C}^* раньше (левее) первого вхождения с-переменной v' , то должно выполняться $v < v'$.
3. Для любого неравенства $x : \neq : y$ в \mathcal{C}^* выполнено $x < y$ и все неравенства в рестрикции отсортированы по отношению ($<$).

Вид x	Вид y	Определение отношения (<)
(ATOM a)	(ATOM b)	$(x < y) \stackrel{\text{def}}{\iff} (a < b)$
(ATOM a)	(CVA j)	$y < x$
(ATOM a)	(CVE j)	$y < x$
(ATOM a)	(CONS p q)	$x < y$
(CVA i)	(ATOM b)	$x < y$
(CVA i)	(CVA j)	$(x < y) \stackrel{\text{def}}{\iff} (i < j)$
(CVA i)	(CVE j)	$(x < y) \stackrel{\text{def}}{\iff} (i < j)$
(CVA i)	(CONS p q)	$x < y$
(CVE i)	(ATOM b)	$x < y$
(CVE i)	(CVA j)	$(x < y) \stackrel{\text{def}}{\iff} (i < j)$
(CVE i)	(CVE j)	$(x < y) \stackrel{\text{def}}{\iff} (i < j)$
(CVE i)	(CONS p q)	$x < y$
(CONS r s)	(ATOM b)	$y < x$
(CONS r s)	(CVA j)	$y < x$
(CONS r s)	(CVE j)	$y < x$
(CONS r s)	(CONS p q)	$(x < y) \stackrel{\text{def}}{\iff} ((r < p) \mid\mid ((r = p) \&\& (s < q)))$

Рис. 2.8: Определение отношения < на множестве с-выражений

Утверждение 24

1. Если \mathcal{C}^* является канонической формой \mathcal{C} , то $\langle \mathcal{C}^* \rangle = \langle \mathcal{C} \rangle$.
2. Существует алгоритм `canon`, который для любого класса \mathcal{C} за конечное число шагов строит класс $(\text{canon } \mathcal{C}^*)$, являющийся канонической формой \mathcal{C} .

Доказательство

Первый пункт утверждения следует из утверждения 23 и определения 22. Описание одного из возможных способов реализации алгоритма `canon`⁷ можно найти в [FTP]. ■

2.9.3 Подклассы и надклассы**Определение 23**

Пусть $\mathcal{C}_1, \mathcal{C}_2$ —классы. Тогда,

- класс \mathcal{C}_2 будем называть *подклассом* класса \mathcal{C}_1 , а класс \mathcal{C}_1 будем называть *надклассом* класса \mathcal{C}_2 , если $\mathcal{C}_2 \preceq \mathcal{C}_1$;
- класс \mathcal{C}_2 будем называть *собственным подклассом* класса \mathcal{C}_1 , а класс \mathcal{C}_1 будем называть *собственным надклассом* класса \mathcal{C}_2 , и будем обозначать $\mathcal{C}_2 \prec \mathcal{C}_1$, если $\mathcal{C}_2 \preceq \mathcal{C}_1$ и канонические формы для классов \mathcal{C}_1 и \mathcal{C}_2 не совпадают: $\text{canon } \mathcal{C}_2 \neq \text{canon } \mathcal{C}_1$.

⁷ Для дальнейшего изложения текст алгоритма `canon` не потребуется.

Если класс $C_2=(ces_2, r_2)$ является подклассом класса $C_1=(ces_1, r_1)$, то (утверждение 22) существуют подстановка s и рестрикция r такие, что $C_2=C_1/.(S\ s)/.(R\ r)$. То есть, $ces_2 = ces_1/.s$. Из этого и из определений 22 и 10 и следует справедливость следующего утверждения.

Утверждение 25

Пусть $C_1=(ces_1, r_1)$, $C_2=(ces_2, r_2)$ —классы, $L(ces)$ —число атомов, с-переменных и конструкторов CONS в списке с-выражений ces . Тогда:

1. Если $C_2 \preceq C_1$, то $L(ces_2) \geq L(ces_1)$.
2. Если $C_2 = \text{canon } C_1$, то $L(ces'_2) = L(ces)$.

Важнейшим следствием из утверждения 25 является следующее утверждение.

Теорема 26

Множество канонических форм надклассов класса C

$$\{ \text{canon } C^* \mid C \preceq C^* \}$$

конечно для любого класса C .

Доказательство

Пусть $C=(ces_0, r_0)$, $L_0=L(ces_0)$, где L определено так же, как и в утверждении 25. Тогда, в силу утверждения 25, можем записать:

$$\begin{aligned} & \{ \text{canon } C^* \mid C \preceq C^* \} \subseteq \\ & \{ (ces', r') \mid C^*=(ces, r), L(ces) \leq L_0, (ces', r') = \text{canon } C^* \} \end{aligned}$$

Последнее множество конечно, так как для его элементов (ces', r') верно следующее:

- Длина ces' ограничена числом L_0 , ces' может содержать *только* атомы (из конечного алфавита атомов), конструкторы CONS и с-переменные (CVA i) и (CVE j), $i = 1, \dots, L_0, j = 1, \dots, L_0$ (“алфавит” из $2 \times L_0$ с-переменных). Таких ces' конечное число.
- Либо $r' = \text{INCONSISTENT}$, либо $r' = \text{RESTR } \text{ineqs}$, где ineqs —список пар $x \neq y$ без повторов, где x и y —атомы (из конечного алфавита атомов) и са-переменные (CVA i), $i = 1, \dots, L_0$, (конечный алфавит из L_0 са-переменных). Таких r' конечное число.

Из предыдущих двух пунктов следует, что различных (ces', r') —конечное число. ■

2.9.4 Разбиения

Определение 24

Пусть (cx, rs) —SR-выражение. Тогда пару сужений $sp=(c_1, c_2)$ назовем *разбиением для (cx, rs)* , а сами сужения c_1 и c_2 назовем *элементарными* сужениями для (cx, rs) , если пара (c_1, c_2) или пара (c_2, c_1) имеет один из следующих видов:

1. $(\text{idC}, \text{emptC})$;
2. $(S[\mathcal{E}.i:->(\text{CONS } \mathcal{E}.m\ \mathcal{E}.n)], S[\mathcal{E}.i:->\mathcal{A}.p])$;
3. $(S[\mathcal{A}.j:->\mathcal{A}.k], R\ \text{RESTR}[\mathcal{A}.j \neq \mathcal{A}.k])$;
4. $(S[\mathcal{A}.j:->'atom'], R\ \text{RESTR}[\mathcal{A}.j \neq 'atom'])$.

где $\mathcal{E}.i$, $\mathcal{A}.j$, $\mathcal{A}.k$ —переменные из cx , то есть они входят в $\text{cvars } cx$; $\mathcal{E}.m$, $\mathcal{E}.n$, $\mathcal{A}.p$ —новые c -переменные для cx , то есть индексы m , n , p не используются в качестве индексов c -переменных в cx ; ' $atom$ '—некоторый атом.

Непосредственно из определения оператора $(/.)$ и определений 13, 24 следует справедливость утверждения 27.

Утверждение 27

Пусть $sp=(c1, c2)$ —разбиение для SR-выражения (cx, rs) с линейной SR-базой и

$$(cx1, rs1)=(cx, rs)/.c1, (cx2, rs2)=(cx, rs)/.c2.$$

Тогда, SR-базы $cx1$ и $cx2$ —линейные.

Следствие 28

Пусть $sp=(c1, c2)$ —разбиение для L-класса \mathcal{C} . Тогда, $\mathcal{C}_1=\mathcal{C}/.c1$ и $\mathcal{C}_2=\mathcal{C}/.c2$ —L-классы.

Основное свойство разбиений состоит в том, что в результате применения разбиения к некоторому SR-выражению, представляющему некоторое множество M , получают два SR-выражения, представляющих два непересекающихся подмножества множества M , объединение которых равно исходному множеству M . То есть, имеет место следующая теорема (доказательство теоремы 29) приведено в [FTP]).

Теорема 29

Пусть $sp=(c1, c2)$ —разбиение для SR-выражения x , $x1=x/.c1$, $x2=x/.c2$.

Тогда $\langle x1 \rangle$ и $\langle x2 \rangle$ —разбиение множества $\langle x \rangle$: $\langle x1 \rangle \subseteq \langle x \rangle$, $\langle x2 \rangle \subseteq \langle x \rangle$, $\langle x1 \rangle \cup \langle x2 \rangle = \langle x \rangle$, $\langle x1 \rangle \cap \langle x2 \rangle = \emptyset$.

Глава 3

Дерево процессов

В разделе 1.3 дано определение процесса вычисления программы $p \in R$ над данными $d \in D$, как последовательности (возможно бесконечной) переходов от одного состояния вычисления к другому:

$$\begin{aligned} \text{process}(p, d) &= s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n \\ s_i &= (t_i, e_i) \end{aligned}$$

В главе 2 были определены средства представления множеств состояний вычислений—*конфигурации*. Для представления *множеств процессов* вычисления программы p используют ориентированные *графы конфигураций*.

Почему важно уметь представлять множества процессов в виде графа конфигураций? Чтобы ответить на этот вопрос, рассмотрим множество процессов вычисления программы p на данных из некоторого класса C : $\mathcal{P}(p, C) = \{ \text{process}(p, d) \mid d \in C \}$. Это множество полностью определяет вычислительное поведение (семантику) программы p на всех данных из C . Допустим, построен граф конфигураций, представляющий $\mathcal{P}(p, C)$. Этот граф является некоторой формой записи программы [4, 11, 12, 18], а именно, специализированной версии программы p , рассчитаной на случай входных данных из класса C . Разработка методов построения оптимальных графов конфигураций (а значит и оптимальных специализированных программ) и есть основное содержание теории суперкомпиляции. Вопросы теории суперкомпиляции лежат за рамками данной работы. Поэтому ограничимся рассмотрением построения алгоритма дерева конфигураций, представляющего $\mathcal{P}(p, C)$.

3.1 Дерево конфигураций

Определение 25

Деревом конфигураций **tree** будем называть ориентированное (возможно бесконечное) дерево,

- каждому узлу n которого приписана некоторая конфигурация s ;
- каждому ребру a , выходящему из некоторого узла n с конфигурацией s приписано сужение cnt ;

- причем, если из узла n с конфигурацией c выходит $k \geq 1$ ребер a_1, \dots, a_k с сужениями $\text{cnt}_1, \dots, \text{cnt}_k$, то множества $\langle c/. \text{cnt}_i \rangle$ попарно не пересекаются, а их объединение совпадает с $\langle c \rangle$, $i = 1, \dots, k$.

Дерево конфигураций будем представлять синтаксическими конструкциями в соответствии с рисунком 3.1.

```

tree    ::= (LEAF conf)
          | (NODE conf [branch1, ..., branchn]) -- n ≥ 0

branch ::= (contr, tree)

```

Рис. 3.1: Синтаксис представления дерева конфигураций

Рассмотрим некоторый (возможно бесконечный) путь w в дереве конфигураций **tree** и некоторую последовательность P (возможно бесконечную) переходов от одного состояния вычисления в языке TSG к другому:

$$\begin{aligned}
 w &= c_0 \xrightarrow{\text{cnt}_0} c_1 \xrightarrow{\text{cnt}_1} \dots c_n \dots \\
 P &= s_0 \rightarrow s_1 \rightarrow \dots s_n \dots
 \end{aligned}$$

Будем говорить, что P отождествляется с путем w дерева **tree** (или, w представляет P), если выполнены следующие условия: P и w имеют одинаковые длины и для каждого состояния s_i выполнено: $s_i \in \langle c_i \rangle$, и если s_i не последнее в P , то $s_i \in \langle c_i/. \text{cnt}_i \rangle$.

Определение 26

Пусть $C = (\text{ces}, r)$ — класс, $p \in R$ — программа, описание первой функции в p имеет вид: (DEFINE f [$\text{parm}_1, \dots, \text{parm}_n$] term). Тогда класс C будем называть *обобщенным данным (с-данным) для программы $p \in R$* , если выполнено следующее: рестрикция $r = \text{RESTR}$ ineqs не содержит противоречий, список с-выражений имеет следующий вид $\text{ces} = [c_1, \dots, c_n]$ и, если parm_i имеет тип ‘a-’, то c_i является са-выражением.

Замечание. Условия в определении 26 гарантируют (см. утверждение 10), что все данные $d \in C$ будут удовлетворять требованиям (см. раздел 1.2.1) на входные данные для программы p : $d = [d_1, \dots, d_n]$, $d_i \in \text{EVal}$ и если parm_i имеет тип ‘a-’, то $d_i \in \text{AVal}$.

Определение 27

Пусть, $p \in R$ — программа, C — обобщенное данное для p . Дерево конфигураций **tree** будем называть *деревом процессов программы $p \in R$ на классе C* , если для любого данного $d \in C$ существует путь из корневой вершины дерева, представляющий процесс $\text{process}(p, d)$ вычисления p над d .

3.2 Построение дерева процессов

В данном разделе будет предъявлен (см. рисунки 3.2 и 3.3) и обоснован алгоритм **ptr**, позволяющий по заданным программе p и классу C — обобщенному данному для p , строить дерево **tree** процессов программы p на классе C .


```

ptr :: ProgR -> Class -> Tree
ptr  p cl@(ces, r) = evalT c p i
                        where
                            (DEFINE f prms _) : p' = p
                            ce = mkEnv prms ces
                            c  = ((CALL f prms, ce), r)
                            i  = freeindx 0 cl

evalT :: Conf -> ProgR -> FreeIndx -> Tree
evalT c@(( CALL f args , ce), r) p i =
    NODE c [ (idC, evalT c' p i) ]
    where
        DEFINE _ prms t' = getDef f p
        ce' = mkEnv prms (args/.ce)
        c' = ((t',ce'),r)

evalT c@(( ALT cnd t1 t2 , ce), r) p i =
    NODE c [(cnt1,evalT c1' p i'),(cnt2,evalT c2' p i')]
    where
        ((cnt1,cnt2), uce1, uce2, i') = ccond cnd ce i
        (_,ce1),r1) = c/.cnt1
        c1' = ((t1, ce1+.uce1), r1)
        (_,ce2),r2) = c/.cnt2
        c2' = ((t2, ce2+.uce2), r2)

evalT c@((exp,ce),r) p i = LEAF c

```

Рис. 3.2: Алгоритм построения дерева процессов. Функции ptr и evalT

3.2.1 Вспомогательные функции в алгоритме ptr

Функции `mkCAVar` и `mkCEVar` используются в алгоритме `ptr` для построения *уникальной* с-переменной (са-переменной и се-переменной, соответственно), отличной от всех с-переменных, уже используемых в текущей конфигурации. Для этого поддерживается свободный индекс с-переменных: `i :: FreeIndx`.

```

mkCAVar, mkCEVar :: FreeIndx -> (CVar, FreeIndx)
mkCEVar i = ((CVE i), i+1 )
mkCAVar i = ((CVA i), i+1 )

```

Функции `splitA` и `splitE` строят разбиения:

```

splitA :: CVar -> CExp -> Split
splitA cv@(CVA _) caexp = (S[cv:->caexp], R (RESTR[cv:=/:caexp]))

splitE :: CVar -> FreeIndx -> (Split,FreeIndx)

```

```

ccond :: Cond->CEnv->FreeIndx -> (Split,CEnv,CEnv,FreeIndx)
ccond (EQA' x y)          ce i =
    let x' = x/.ce; y' = y/.ce in
    case (x', y') of
        (a,      b      )|a==b->((idC ,emptC), [],[],i)
        (ATOM a,ATOM b)    ->((emptC, idC), [],[],i)
        (CVA _, a      )   ->( splitA x' a , [],[],i)
        (a,      CVA _ )   ->( splitA y' a , [],[],i)

ccond (CONS' x vh vt va) ce i =
    let x' = x/.ce in
    case x' of
        CONS h t ->((idC,emptC), [vh:=h,vt:=t],          [], i )
        ATOM a   ->((emptC,idC), [],                      [va:=x'], i )
        CVA _    ->((emptC,idC), [],                      [va:=x'], i )
        CVE _    ->( split,      [vh:=ch,vt:=ct], [va:=ca], i')
        where
            (split,i') = splitE x' i
            (S[_:->(CONS ch ct)], S[_:->ca]) = split

```

Рис. 3.3: Алгоритм построения дерева процессов. Функция ccond

```

splitE cv@(CVE _) i = ((S[cv:->(CONS cvh cvt)],
    S[cv:->cva]), i')
    where (cvh,i1)=mkCEVar i
           (cvt,i2)=mkCEVar i1
           (cva,i')=mkCAVar i2

```

3.2.2 Обоснование алгоритма ptr

Аргументами входной функции **ptr** алгоритма являются: **p** — TSG-программа, **cl=(ces,r)**—класс *C*, **i**—начальное значение свободного индекса *s*-переменных, целое число, большее индекса любой *s*-переменной из **cl**.

Текст алгоритма **ptr** во многом совпадает с текстом интерпретатора **int** языка TSG (ср. рисунок 1.5 на странице 19 с рисунками 3.2 на странице 47 и 3.3 на странице 48), а вспомогательные функции и операторы—**mkEnv**, **/.**, **+**—просто являются общими для **ptr** и **int**. Главная идея алгоритма **ptr** состоит в следующем. В **ptr** выполняется обобщенное вычисление программы **p** над обобщенными данными *C*, почти точно так же, как это происходит при обычных вычислениях, выполняемых алгоритмом **int**. Различия относятся к следующим моментам:

1. Обобщенные данные содержат *s*-переменные и представляют множество данных.
2. Состояние обобщенного вычисления представлено конфигурацией, значения программных переменных—*s*-средой.

Выполнение каждого шага обобщенного вычисления над конфигурацией c состоит в следующем:

1. Если вычисляемый терм в c является выражением, то c —пассивная конфигурация, $\langle c \rangle$ содержит только пассивные состояния. В этом случае ptr строит терминальную (листовую) вершину дерева: (**LEAF** c).
2. Для непассивных конфигураций выполняется шаг обобщенного вычисления, тем же образом, как это делается в **int**. При этом возможны два случая:

- Либо наличие конфигурационных переменных в обобщенном состоянии (конфигурации) c не мешает однозначно построить следующее обобщенное состояние c' —то есть, для всех состояний из c очередной шаг выполняется одним и тем же способом. В этом случае выполняется построение из вершины с конфигурацией c одного ребра:

NODE c [(**idC**, subtree)]

где **subtree**—поддерево конфигураций, построенное для конфигурации c' .

- Либо наличие конфигурационных переменных в обобщенном состоянии (конфигурации) c мешает однозначно построить следующее обобщенное состояние—при разных значениях конфигурационных переменных получаются состояния, выбирающие разные способы продолжения вычислений. В этом случае *удается* построить такое разбиение $\text{sp}=(\text{cnt1}, \text{cnt2})$, что для каждой из подконфигураций $c1=c/. \text{cnt1}$ и $c2=c/. \text{cnt2}$ уже можно однозначно выполнить шаг вычислений и перейти к конфигурации $c1'$ —в результате выполнения шага для конфигурации $c1$, и к конфигурации $c2'$ —в результате выполнения шага для конфигурации $c2$. В этом случае выполняется построение развилки—двух ребер из вершины c с конфигурацией c :

NODE c [(**cnt1**, subtree1), (**cnt2**, subtree2)]

где **subtree1**—поддерево конфигураций, построенное для конфигурации $c1'$, **subtree2**—поддерево конфигураций, построенное для конфигурации $c2'$.

Рассмотрим подробнее алгоритм **ptr**. По своим аргументам $p, c1=(\text{ces}, r)$, функция **ptr** формирует начальную конфигурацию обобщенного вычисления $c = ((t, \text{ce}), r)$, где:

- $t = (\text{CALL } f \text{ } \text{prms})$ —терм, который необходимо вычислить, f —первая функция из программы p , prms —список ее параметров;
- $\text{ce} = \text{mkEnv } \text{prms } \text{ces}$ —начальная среда обобщенного вычисления;

Заметим, что t и ce —определяются по тем же правилам, по которым в обычных вычислениях (см. алгоритм **int** на рисунке 1.5) определяется начальное состояние вычисления $s = (t, e)$.

Далее вызывается функция **evalT** с p и i , которая на каждом шаге строит:

- либо вершину с одним ребром (первое предложение **evalT**):

NODE c [(**idC**, subtree)];

- либо вершину с двумя ребрами (второе предложение `evalT`):

`NODE c [(cnt1, subtree1), (cnt2, subtree2)];`

- либо терминальную вершину (третье предложение `evalT`):

`LEAF c;`

где `subtree`, `subtree1`, `subtree2`—деревья, построенные рекурсивными вызовами функции `evalT` для новых конфигураций, `(cnt1, cnt2)`—разбиение конфигурации `c`, определяемое функцией `ccond`.

В первом предложении функции `evalT` рассматривается случай конфигурации, у которой вычисляемый терм является вызовом функции. В этом случае обобщенные вычисления (переход к новой конфигурации `c'`) выполняются однозначно, по тем же правилам, как это делается в обычных вычислениях (ср. первое предложение функции `eval` из алгоритма `int` и первое предложение функции `evalT` из алгоритма `ptr`). Генерируется вершина дерева, с одной ветвью:

`NODE c [(idC, subtree)]`

где `subtree`—дерево, построенное рекурсивным вызовом функции на новой конфигурации: `evalT c' p i`.

Во втором предложении `evalT` рассмотрена конфигурация `c=((t, ce), r)`, вычисляемый терм `t` которой имеет вид условной конструкции: `t = (ALT cnd t1 t2)`.

В этом случае вызывается функция `ccond cnd ce i`, которая является обобщенной версией функции `cond` из алгоритма `int`. При помощи функции `ccond` строится разбиение `(cnt1, cnt2)` для конфигурации `c`, такое, что:

1. в множество `<c/.cnt1>` попадают те и только те состояния `s ∈ <c>`, для которых на данном шаге обычного вычисления результат проверки условия `cnd` будет положительным—`TRUE`;
2. в множество `<c/.cnt2>` попадают те и только те состояния `s ∈ <c>`, для которых на данном шаге обычного вычисления результат проверки условия `cnd` будет отрицательным—`FALSE`.

Указанное свойство разбиения `(cnt', cnt'')` следует из определения функции `ccond`, теоремы 29 и утверждения 5, в котором показано, что различные значения `c`-переменных соответствуют различным элементам множества `<c>`.

Помимо указанного разбиения функция `ccond` определяет два изменения `c`-среды—`use1` и `use2`,—которые являются изменениями `c`-среды для случаев, если результат проверки условия `cnd` равен, соответственно, `TRUE` и `FALSE`.

По сути, функция `ccond` является таблицей, в которой разобраны различные случаи вычисления условия `c` на `c`-среде—см. рисунок 3.3.

По результату вычисления функции `ccond`:

`((cnt1, cnt2), use1, use2, i') = ccond cnd ce i`

выполняют следующее:

1. Строят две подконфигурации конфигурации `c`, для которых можно однозначно выполнить текущий шаг обобщенного вычисления:

$$\begin{aligned} c1 &= ((t, ce1), r1) = c/.cnt1 \\ c2 &= ((t, ce2), r2) = c/.cnt2 \end{aligned}$$

2. Выполняют шаг вычислений для конфигурации $c1$, которая по построению содержит все состояния, для которых проверка условия cnd приводит к результату **TRUE**—определяют новое обобщенное состояние в полном соответствии с правилами выполнения обычных вычислений для результата **TRUE** проверки условия cnd :

$$c1' = ((t1, ce1+.uce1), r1)$$

3. Выполняют шаг вычислений для конфигурации $c2$, которая по построению содержит все состояния, для которых проверка условия cnd приводит к результату **FALSE**—определяют новое обобщенное состояние в полном соответствии с правилами выполнения обычных вычислений для результата **FALSE** проверки условия cnd :

$$c2' = ((t2, ce2+.uce2), r2)$$

4. Формируют вершину дерева с развилкой:

$$\text{NODE } c \text{ [(cnt1, subtree1), (cnt2, subtree2)]}$$

где поддеревья $subtree1$ и $subtree2$ строятся рекурсивными вызовами функции $evalT$ для конфигураций $c1'$ и $c2'$, соответственно:

$$\begin{aligned} subtree1 &= evalT \ c1' \ p \ i' \\ subtree2 &= evalT \ c2' \ p \ i' \end{aligned}$$

В третьем предложении функции $evalT$ рассматривается пассивная конфигурация c , у которой вычисляемый терм является выражением. В этом случае обобщенное вычисление завершается—генерируется терминальная вершина дерева: **LEAF** c .

Более подробный разбор алгоритма ptr и доказательство теоремы 30 можно найти в [FTP].

Теорема 30

Пусть $p \in R$ —программа, $c1 = (ces, r)$ —обобщенное данное для p , i —целое число, большее индекса любой s -переменной из $c1$. Тогда определяемое алгоритмом ptr дерево $tree = ptr \ p \ c1 \ i$ является деревом процессов программы p на классе $c1$.

3.2.3 Улучшенный алгоритм построения дерева процессов

Рассмотрим дерево $tree$, получаемое в результате вычисления $ptr \ p \ c1 \ i$. В дереве $tree$ могут быть “лишние” ветви, а именно ветви b (будем называть их *сухие ветви*), ведущие от некоторой вершины с конфигурацией $c1$ к вершине с пустой конфигурацией c :

$$\begin{aligned} c1 &\xrightarrow{cnt} c \\ b &= (cnt, \text{NODE } c \ bs), \ c = ((t, ce), \text{INCONSISTENT}) \end{aligned}$$

Как следует из текста функции `evalT`, если от вершины с некоторой конфигурацией `c1` имеется ребро с рестрикцией `cnt`, ведущее к вершине `c`:

$$c1 \xrightarrow{cnt} c$$

то две конфигурации `c1/.cnt` и `c` имеют текстуально одну и ту же рестрицию. Таким образом для сухой ветви `b` выполнено:

$$\begin{aligned} c1 &\xrightarrow{cnt} c \\ b &= (cnt, \text{NODE } c \text{ } bs), \quad c = ((t, ce), \text{INCONSISTENT}), \\ c1/.cnt &= ((t', ce'), \text{INCONSISTENT}), \\ \langle c \rangle &= \emptyset, \quad \langle c1/.cnt \rangle = \emptyset \end{aligned}$$

В силу этих соотношений и по определениям 25 и 27, имеет место следующее утверждение:

```
xptr :: ProgR -> Class -> Tree
xptr  p cl = NODE c (cutT brs)
      where
          tree      = ptr p cl
          NODE c brs = tree

cutT :: [Branch] -> [Branch]
cutT [ ]          = [ ]
cutT (b@(cnt, tree) : bs) =
    case tree of
        LEAF (_, INCONSISTENT) -> cutT bs
        NODE (_, INCONSISTENT) _ -> cutT bs
        LEAF c                    -> (b : cutT bs)
        NODE c                    bs' -> (b' : cutT bs)
        where tree' = NODE c (cutT bs')
              b'    = (cnt, tree')
```

Рис. 3.4: Улучшенный алгоритм построения дерева процессов

Утверждение 31

Пусть $p \in R$ —программа, cl —обобщенное данное для p , i —целое число, большее индекса любой s -переменной из cl , $tree'$ —дерево, полученное из дерева $tree$ удалением сухих ветвей (вместе с вершиной и поддеревом, к которому ведет сухая ветвь), где $tree = ptr \ p \ cl \ i$. Тогда $tree'$ —дерево процессов программы p на данных класса cl .

Возможность появления сухих ветвей в дереве $tree$ особенно очевидна для тех случаев, когда `scond` возвращает в качестве разбиения `(idC, emptC)` или `(emptC, idC)`. В работе [18] обсуждаются менее тривиальные возможности появления сухих ветвей в дереве процессов. В силу утверждения 31 можно улучшить алгоритм `ptr`, сохранив при этом все свойства результирующего дерева: либо внести такие исправления в функцию `evalT`, чтобы сухие ветви не порождались, либо вспомогательной функцией отсечь

сухие ветви в построенном дереве. Остановимся на втором варианте¹. На рисунке 3.4 приведен улучшенный алгоритм `xptr` построения дерева процессов. Данный алгоритм вычисляет дерево `tree = ptr p cl i` и удаляет при помощи функции `cutT` в этом дереве сухие ветви. По определению алгоритма `xptr`, в силу теоремы 30 и утверждения 31, имеет место следующая теорема.

Теорема 32

Пусть $p \in R$ —программа, $cl = (ces, r)$ —обобщенное данное для p , i —целое число, большее индекса любой s -переменной из cl . Тогда `tree = xptr p cl i`—является деревом процессов программы p на данных класса cl .

¹ Второй вариант по эффективности не отличается от первого: в силу ленивой семантики языка *Gofers*—сухая ветвь будет отсекается раньше, чем будет построено поддерево, к которому ведет данная ветвь.

Глава 4

Окрестностный анализ

В данной главе для языка TSG будет построена и исследована алгоритмическая реализация точного окрестностного анализа. При этом окрестностный анализ будет рассматриваться как общий метод, без привязки к какому-нибудь применению.

4.1 Основные понятия

Окрестностный анализ является инструментом формализации интуитивного вопроса:

Какая информация о тексте d была использована, и какая информация о d не была использована в некотором процессе p обработки текста d ?

Здесь рассматриваются только *конструктивные* процессы обработки текста, реализованные программами p на языке R .

Пусть $p \in R$ —программа, реализующая некоторые процессы обработки текстов. Для заданного текста $d \in D$ будем иметь конкретный процесс $p \ d \xRightarrow{*} res$ его обработки, где res —результат вычисления p на d .

Какой смысл имеет интуитивная фраза “в процессе $p \ d \xRightarrow{*} res$ использовалась не вся информация о тексте d ”? Очевидно, это означает, что текст d может быть *изменен*—то есть заменен на некоторый другой текст d' —так, что процесс обработки измененного текста $p \ d' \xRightarrow{*} res$ будет точно таким же, как и процесс $p \ d \xRightarrow{*} res$. При этом, если нет вариантов изменения текста d , сохраняющих процесс $p \ d \xRightarrow{*} res$ неизменным, то *вся информация* о тексте d была использована в процессе $p \ d \xRightarrow{*} res$; если есть варианты изменения текста d , сохраняющих процесс $p \ d \xRightarrow{*} res$ неизменным, то *часть информации* о тексте d была использована в процессе $p \ d \xRightarrow{*} res$, а часть—нет. Множество всех возможных модификаций текста d —замен текста d на другой текст d' —сохраняющих процесс $p \ d \xRightarrow{*} res$ неизменным:

$$o(p, d) = \left\{ d' \mid \begin{array}{l} \text{процесс } p \ d' \xRightarrow{*} res \text{ точно такой же, как} \\ \text{процесс } p \ d \xRightarrow{*} res \end{array} \right\} \quad (4.1)$$

определяет, какая информация о тексте d была использована, и какая информация о тексте d не была использована в процессе $p \ d \xRightarrow{*} res$.

Определение (4.1) все еще неформальное, так как неясно, что означает условие “процесс $p \ d' \xRightarrow{*} res$ точно такой же, как процесс $p \ d \xRightarrow{*} res$ ”. Чтобы его точно определить, используем понятие *трассы* $tr(p, d)$ (см. раздел 1.3). При различных входных

данных d процесс $p \ d \xRightarrow{*} res$ может иметь различные пути (истории) своего развития. В главе 3 введено понятие дерева процесса. Различные пути этого дерева (в общем случае—бесконечного) представляют различные истории развития процесса $p \ d \xRightarrow{*} res$ при различных данных $d \in D$. Развилки этого дерева соответствуют вариантам ветвлений в программе p . Каждое конкретное данное d выбирает в данном дереве один путь, соответствующий конкретной истории развития процесса $p \ d \xRightarrow{*} res$. Трасса процесса в разделе 1.3 определена для TSG как список результатов—**True** или **False**—всех проверок, выполненных в процессе вычисления $p \ d$. Таким образом, трасса содержит информацию, которая из всех возможных путей развития вычисления p позволяла бы выбрать тот самый путь, что был выбран данными d .

Таким образом, в неформальном определении (4.1) множества $o(p, d)$ условие “*процесс $p \ d' \xRightarrow{*} res$ точно такой же, как процесс $p \ d \xRightarrow{*} res$* ” можно формализовать, используя понятие трассы:

$$tr(p, d) = tr(p, d') \ \& \ p \ d = p \ d'$$

После чего множество $o(p, d)$, описывающее, какая информация о тексте d была использована, а какая—нет, в процессе $p \ d \xRightarrow{*} res$ обработки текста d , получает формальное определение:

$$o(p, d) = \{ d' \mid tr(p, d) = tr(p, d'), p \ d = p \ d' \} \quad (4.2)$$

Чтобы предъявлять множество $o(p, d)$ для дальнейшего использования, оно должно быть представлено в виде конструктивного объекта. В главе 2 выбраны средства представления множеств данных в виде классов. При этом было известно, что найдутся множества данных, непредставимые данным способом—см. сноску 4 на странице 10. То есть, предстоит убедиться, что для любой программы p и данных d множество $o(p, d)$ *представимо* в виде класса. Это будет сделано в разделе 4.3, где предъявлен алгоритм **nan**—окрестностный анализатор,—вычисляющий L-класс \mathcal{O} , представляющий множество $o(p, d)$.

Помимо окрестностного анализатора в инструментарий окрестностного анализа входит набор операций над окрестностями—пересечение окрестностей, декомпозиция окрестности и т.п. Поэтому в разделе 4.4 будут конструктивно (в виде алгоритмов) определены и обсуждены различные операции над классами и окрестностями.

4.2 Окрестности

Определение 28

Пусть $\mathcal{O} = (ces, rs)$ —L-класс, $d \in \langle \mathcal{O} \rangle \subseteq [EVal]$. Тогда будем называть \mathcal{O} окрестностью d .

Таким образом, L-класс \mathcal{O} является окрестностью для любого элемента из $\langle \mathcal{O} \rangle$. Все свойства L-классов справедливы и для окрестностей. Единственное отличие между L-классами и окрестностями: наличие выделенного элемента d —*центра* окрестности. Когда это будет удобным, будем записывать центр окрестности d в позиции верхнего индекса у обозначения окрестности: \mathcal{O}^d . По определению, окрестность не может быть пустой: $d \in \langle \mathcal{O}^d \rangle$.

Окрестности \mathcal{O}^d данных d будут использоваться для представления множеств $o(p, d) \leftarrow \langle \mathcal{O}^d \rangle = o(p, d)$, — которые в свою очередь определяют, какая информация о данных d была использована в процессе обработки данных d программой p . Множество $o(p, d)$, по определению не пусто, так как $d \in o(p, d)$. Если d — единственный элемент¹ $\langle \mathcal{O}^d \rangle$, то вся информация о данных d была использована в процессе p d — невозможно заменить d на другие данные d' , сохранив при этом неизменными трассу и результат процесса обработки.

Если $\langle \mathcal{O}^d \rangle = o(p, d)$ помимо d содержит и другие элементы, то не вся информация о данных d была использована в процессе p d — можно данные d заменить на другие данные d' , сохранив при этом неизменными трассу и результат процесса обработки. Чем больше вариантов таких замен — чем шире множество $\langle \mathcal{O}^d \rangle = o(p, d)$ — тем меньше информации о данных d было использовано в процессе p d ².

Пусть имеются две окрестности $\mathcal{O}_1^d, \mathcal{O}_2^d$ данных d , $\langle \mathcal{O}_1^d \rangle = o(p_1, d)$, $\langle \mathcal{O}_2^d \rangle = o(p_2, d)$ и выполнено $\langle \mathcal{O}_1^d \rangle \subset \langle \mathcal{O}_2^d \rangle$, тогда можно сказать, что в процессе p_2 d обработки данных d использовано *больше* информации о d , чем в процессе p_1 d .

Определение 29

Пусть $\mathcal{O}_1^d, \mathcal{O}_2^d$ — окрестности данных d . Тогда

- \mathcal{O}_2^d будем называть *подокрестностью* окрестности \mathcal{O}_1^d , а \mathcal{O}_1^d будем называть *надокрестностью* \mathcal{O}_2^d , если $\mathcal{O}_2^d \preceq \mathcal{O}_1^d$.
- \mathcal{O}_2^d будем называть *собственной подокрестностью* окрестности \mathcal{O}_1^d , а \mathcal{O}_1^d будем называть *собственной надокрестностью* \mathcal{O}_2^d , если $\mathcal{O}_2^d \prec \mathcal{O}_1^d$.

Учитывая утверждение 19, отношения $\mathcal{O}_2^d \preceq \mathcal{O}_1^d$ и $\mathcal{O}_2^d \prec \mathcal{O}_1^d$ можно трактовать следующим образом: в окрестности \mathcal{O}_2^d определено *больше информации о d* (меньше неопределенностей, меньше возможностей для вариаций), чем в \mathcal{O}_1^d .

В силу определения 28 и утверждения 20 выполнено следующее: *L-класс \mathcal{O} является окрестностью данных d тогда и только тогда, когда \mathcal{O} является надклассом L-класса $\text{sngl}(d)$* . С учетом данного обстоятельства, следующая теорема 33 является прямым следствием теоремы 26.

Теорема 33

Для любого данного $d \in D$ множество канонических форм окрестностей d конечно и не существует бесконечной последовательности $\mathcal{O}_1^d \succ \mathcal{O}_2^d \succ \mathcal{O}_3^d \succ \dots$ окрестностей данных d .

Теорема 33 согласуется с информационной интерпретацией окрестностей данных d . Если $\mathcal{O}_k^d \succ \mathcal{O}_{k+1}^d$, то \mathcal{O}_k^d определяет некоторую информацию о d , а \mathcal{O}_{k+1}^d определяет более полную (уточненную) информацию о d . Невозможно бесконечно уточнять информацию о данных d : *каждая конечная структура данных содержит “в себе” конечное “количество” информации*. За конечное число уточнений будет сказано все о данных d — очередная окрестность в последовательности уточнений сожмется в точку: $\mathcal{O}_n^d = \text{sngl}(d)$, $\langle \mathcal{O}_n^d \rangle = \{d\}$.

¹ В этом случае (см. утверждение 13), $o(p, d)$ представимо L-классом $\text{sngl}(d)$, который является одноэлементной — самой узкой — окрестностью d : $\langle \text{sngl}(d) \rangle = \{d\}$.

² Это полностью согласуется с классическим определением информации, как меры неопределенности, меры допустимых вариаций.

```

mkCExps :: Params -> FreeIndx -> (CExps, FreeIndx)
mkCExps [ ]          i = ([], i)
mkCExps (parm:params) i = ( (cvar:ces), i'')
                        where
                        (cvar,i' ) = case parm of
                                PVE _ -> mkCEVar i
                                PVA _ -> mkCAVar i
                        (ces, i'') = mkCExps params i'

```

Рис. 4.1: Вспомогательная функция mkCExps

4.3 Окрестностный анализатор

В данном разделе будет показано, что для любой программы $p \in R$ и данных $d \in D$ множество $o(p, d)$ представимо в виде окрестности \mathcal{O}^d данных d : $\langle \mathcal{O}^d \rangle = o(p, d)$. На рисунке 4.2 приведен *окрестностный анализатор nan*—алгоритм построения результата вычисления $p \ d \xRightarrow{*} res$ и указанной окрестности \mathcal{O}^d : $nan \ d \ p \rightarrow (res, \mathcal{O}^d)$.

4.3.1 Вспомогательные функции алгоритма nan

Как определено в разделе 1.2.1, если список параметров первой функции в программе p имеет вид: $[parm_1, \dots, parm_n]$, то входные данные для программы p должны быть списком а- и е-значений $[d_1, \dots, d_n] \in [EVal]$, причем $d_i \in AVal$, если $parm_i$ —а-переменная. Таким образом, множество $Dom(p)$ *всех возможных входных данных* программы p имеет следующее представление:

$$\mathcal{O}_0 = (ces_0, RESTR[]), \quad ces_0 = (cvar_1, \dots, cvar_n),$$

где $cvar_1, \dots, cvar_n$ —различные с-переменные и $cvar_i$ —са-переменная, если $parm_i$ —а-переменная, $cvar_i$ —се-переменная, если $parm_i$ —е-переменная.

Для построения списка с-переменных ces_0 по списку параметров первой функции программы в окрестностном анализаторе *nan* используется функция *mkCExps* (см. рисунок 4.1).

Например, если список параметров первой функции в программе p имеет вид $[a.x, a.y, e.z]$, то с помощью функции *mkCExps* будет построено следующее представление для множества $Dom(p)$:

$$ces_0 = [A.1, A.2, \mathcal{E}.3], \\ Dom(p) = \langle \mathcal{O}_0 \rangle = \langle ces_0, RESTR[] \rangle.$$

Кроме функции *mkCExps* в алгоритме *nan* используются все вспомогательные функции и операторы из алгоритмов *int* и *nan*: оператор $(/.)$ применения сужений, подстановок и сред, оператор $(+.)$ пополнения сред, функции *mkCAVar* и *mkCEVar* построения *уникальной* с-переменной (са-переменной и се-переменной, соответственно), функция *mkEnv* построения среды и с-среды, функция *cond* проверки условия ALT-конструкции и построения пополнения среды в обычном (эталонном) вычислении и ее обобщенная версия—функция *scond*, функция *unify* отождествления с-выражений—см. рисунок 2.6 на странице 32.

4.3.2 Обоснование окрестностного анализатора

Идея алгоритма `nan` состоит в следующем. В окрестностном анализаторе выполняется обычное вычисление $p \ d \xrightarrow{*} \text{res}$, которое будем называть эталонным вычислением. Эталонное вычисление выполняется *в точности так, как это делается в интерпретаторе `int`*—см. раздел 1.3.

Одновременно и синхронно с эталонным вычислением выполняется *обобщенное вычисление* программы p над s -данными—над окрестностью \mathcal{O}^d данных d . В начале обобщенных вычислений в качестве окрестности данных d выбирается максимально возможная окрестность, представляющая множество $\text{Dom}(p)$ всех возможных данных для программы p . Обобщенное вычисление выполняется *в точности так, как это делается в алгоритме построения дерева процессов `ptr`*—см. раздел 3.2.

В тех случаях, когда данные из множества, представленного \mathcal{O}^d , не все однозначно выбирают тот вариант продолжения вычисления, что и в эталонном вычислении, `nan` проводит сужение \mathcal{O}^d , отбрасывая *ровно те данные* из представляемого окрестностью множества, которые выбрали иное продолжение истории вычисления. То есть, в обобщенном вычислении, выполняемом в `nan`, отслеживается та самая ветвь в дереве процессов, которая соответствует эталонному вычислению. Другие ветви отсекаются. Текст тех фрагментов алгоритма `nan`, которые относятся к обобщенному вычислению, абсолютно совпадает с соответствующими фрагментами текста `ptr`³.

Таким образом обеспечивается построение представления множества тех данных, которые имеют ту же трассу вычисления, что и данные d ⁴.

Рассмотрим подробнее алгоритм `nan`.

Функция `nan` по заданной программе p и входным данным ds формирует аргументы функции `eval`⁵:

- $t_0 = (\text{CALL } f \text{ } prms)$ —терм, который необходимо вычислить, где f —первая функция из программы p , $prms$ —список ее параметров.
- $\mathcal{O}_0^d = (ces_0, r_0)$ начальную окрестность d , представляющую множество *всех возможных входных данных* программы p , где ces_0 определен равенством

$$(ces_0, i_0) = \text{mkCExps } prms \ 1,$$

а начальное значение рестрикции $r_0 = \text{RESTR}[]$;

- $e_0 = \text{mkEnv } prms \ ds$ —начальную среду эталонного вычисления;
- $ce_0 = \text{mkEnv } prms \ ces_0$ —начальную среду обобщенного вычисления;
- i_0 —начальное значение свободного индекса s -переменных.

³ Можно было определить `nan` следующим эквивалентным образом: строится дерево процессов $\text{tree} = \text{ptr } p \ \mathcal{O}^d$, выполняется эталонное вычисление и осуществляется выбор из дерева `tree` той ветви, которая отвечает эталонному вычислению. Однако, выбранное здесь определение `nan` представляется более удобным для обоснования алгоритма.

⁴ Заметим, что в начале развития окрестностного анализа для окрестностей использовался термин “класс конвенциальности”, что подчеркивало их смысл—множество данных, выбирающих один и тот же путь в дереве процессов.

```

nan :: ProgR -> [EVal] -> (EVal, Class)
nan p d = eval' (CALL f prms) e ce (RESTR[]) ces p i
  where (DEFINE f prms _) : p' = p
        (ces, i) = mkCExps prms 1
        e        = mkEnv prms d
        ce       = mkEnv prms ces

eval' :: Term->Env->CEnv->Restr->CExps->ProgR->FreeIndx ->
                                             (EVal, Class)

eval' (CALL f args) e ce r ces p i =
  eval' t e' ce' r ces p i
  where
    DEFINE _ prms t=getDef f p
    e'      = mkEnv prms (args/.e)
    ce'     = mkEnv prms (args/.ce)

eval' (ALT c t1 t2) e ce r ces p i =
  case cond c e of
    TRUE ue  -> eval' t1 (e+.ue) ce1' r1 ces1 p i'
    FALSE ue -> eval' t2 (e+.ue) ce2' r2 ces2 p i'
  where
    ((cnt1,cnt2), uce1, uce2, i') = ccond c ce i
    ((ce1,ces1),r1) =((ce,ces),r)/.cnt1
    ce1' =ce1+.uce1
    ((ce2,ces2),r2) =((ce,ces),r)/.cnt2
    ce2' =ce2+.uce2

eval' exp e ce r ces p i =
  (res, (ces, r)/.subst)
  where
    res =exp/.e
    (True,subst)=unify [exp/.ce] [res]

```

Рис. 4.2: Окрестностный анализатор для языка TSG

Заметим, что все, относящееся к эталонному вычислению— t_0 и e_0 —определяются в *точности* так же, как они определялись в интерпретаторе `int p d`, а с-среда ce_0 и ее рестрикция определена в *точности* так же, как они определялись бы в вычислении `ptr p \mathcal{O}_0^d i_0` .

Далее рекурсивно выполняется функция `eval'`:

```

eval' t0 e0 ce0 r0 ces0 p i0 ⇒
eval' t1 e1 ce1 r1 ces1 p i1 ⇒
...
eval' tn en cen rn cesn p in

```

пока вычисляемый терм t_n не будет пассивным.

Тем самым выполняются шаги эталонного и обобщенного вычисления—переходы от одного состояния эталонного и обобщенного вычисления $(s_n = (t_n, e_n), c_n = ((t_n, ce_n), r_n))$ к другому—и переходы от одной окрестности $\mathcal{O}_n^d = (ces_n, r_n)$ данных d к другой. Простым сравнением текстов программ **int** и **nan** легко заметить, что все преобразования состояния s_k в обеих программах определены абсолютно одинаково. Эталонное вычисление в **nan** полностью совпадает с обычным вычислением **int** $p\ d$, k -тый рекурсивный вызов функции **eval** соответствует k -тому рекурсивному вызову функции **eval**, то есть соответствует состоянию “выполнено k шагов вычисления $p\ d$ ”, где $k = 0, 1, 2, \dots$. Аналогичное совпадение можно заметить и сравнив текст **nan** с **ptr**: все преобразования обобщенного состояния s_k в обеих программах определены абсолютно одинаково. Это обеспечивает следующее свойство: $\langle \mathcal{O}_i^d \rangle = \{ d' \mid tr_i(p, d') = tr_i(p, d) \}$.

Выполнение окрестностного анализатора завершается, когда очередной вычисляемый терм t_n пассивный—имеет вид выражения—см. третье предложение функции **eval**, $t_n = \text{exp}$, где n —число шагов в эталонном вычислении $p\ d \xRightarrow{*} \text{res}$. При этом выполнено:

$$\langle \mathcal{O}_n^d \rangle = \langle ces_n, r_n \rangle = \{ d' \mid tr(p, d') = tr(p, d) \}, d \in \langle \mathcal{O}_n^d \rangle, e_n \in \langle ce_n, r_n \rangle.$$

Для завершения окрестностного анализа выполняются следующие действия:

1. Вычисляется результат **res** эталонного вычисления, точно так же, как это делается в интерпретаторе **int**: $\text{res} = \text{exp} / .e_n$.
2. Вычисляется с-выражение **cres**—значение **exp** на с-среде ce_n ,—результат обобщенного вычисления: $\text{cres} = \text{exp} / .ce_n$.
3. Вычисляется подстановка **su**, которая назначает с-переменным из **cres** такие значения, что $\text{res} = \text{cres} / .su$ (см. раздел 2.6):
 $(\text{True}, su) = \text{unify} [\text{cres}] [\text{res}]$.
4. Выполняется сужение \mathcal{O}_n^d при помощи подстановки **su**:
 $\mathcal{O}^d = \mathcal{O}_n^d / .su = (ces_n, r_n) / .su$.
5. В качестве результата окрестностного анализа выдается пара: $(\text{res}, \mathcal{O}^d)$.

По построению, $\text{res} = p\ d$. Кроме того,

$$\mathcal{O}^d = \mathcal{O}_n^d / .su \subseteq \mathcal{O}_n^d, \langle \mathcal{O}^d \rangle \subseteq \langle \mathcal{O}_n^d \rangle = \{ d' \mid tr(p, d') = tr(p, d) \}$$

Для завершения обоснования алгоритма **nan** осталось показать, что:

$$\langle \mathcal{O}^d \rangle = o(p, d) = \{ d' \mid tr(p, d) = tr(p, d'), p\ d = p\ d' \}$$

Заметим, что:

$$\begin{aligned} o(p, d) &= \\ &= \{ d' \mid tr(p, d) = tr(p, d'), p\ d = p\ d' \} \\ &= \{ d' \mid d' \in \langle \mathcal{O}_n^d \rangle, \text{res} = p\ d' \} \\ &= \{ d' \mid s = \text{FVS}(\mathcal{O}_n^d), d' = ces_n / .s, t_n / .(ce_n / .s) = \text{res} \} \\ &= \{ ces_n / .s \mid s = \text{FVS}(\mathcal{O}_n^d), (t_n / .ce_n) / .s = \text{res} \} \\ &= \{ ces_n / .s \mid s = \text{FVS}(\mathcal{O}_n^d), \text{cres} / .s = \text{res} \} \end{aligned} \tag{**}$$

А. Рассмотрим произвольные данные $d' \in o(p, d)$.

Тогда (см. (**)) существует подстановка s , такая, что

$$\begin{aligned} s &= FVS(\mathcal{O}_n^d), \\ d' &= ces_n/.s, \\ r_n/.s &= RESTR[], \quad cres/.s = res. \end{aligned}$$

Рассмотрим подстановку $s' = su.*.s$. Заметим, что выполнено:

$$cres/.s' = cres/.(su.*.s) = (cres/.su)/.s = res/.s = res = cres/.s$$

Предпоследнее равенство выполнено в силу того, что res не содержит s -переменных. В силу утверждения 5 из равенства $cres/.s' = cres/.s$ следует, что $v/.s' = v/.s$ для любой s -переменной $v \in (cvars \ cres)$. Рассмотрим любую s -переменную $v \notin (cvars \ cres)$. В силу утверждения $refl:unify, (cvars \ cres) = (dom \ su)$. Поэтому

$$v/.s' = v/.(su.*.s) = (v/.su)/.s = v/.s$$

Итак, для любой s -переменной v выполнено: $v/.s' = v/.s$. Следовательно

$$\begin{aligned} (d', RESTR[]) &= (ces_n, r_n)/.s = \mathcal{O}_n^d/.s = \mathcal{O}_n^d/.s' = \mathcal{O}_n^d/.(su.*.s) = \\ &= (\mathcal{O}_n^d/.su)/.s = \mathcal{O}^d/.s \end{aligned}$$

Так как $\mathcal{O}^d/.s = (d', RESTR[])$, то $s \in FVS(\mathcal{O}^d)$ и данные d' являются элементом $\langle \mathcal{O}^d \rangle$: $d' \in \langle \mathcal{O}^d \rangle$.

В. Рассмотрим произвольные данные $d' \in \langle \mathcal{O}^d \rangle = \langle \mathcal{O}_n^d/.su \rangle$.

Существует $s \in FVS(\mathcal{O}^d)$ такая, что $\mathcal{O}^d/.s = (d', RESTR[])$. Рассмотрим $s' = su.*.s$.

Тогда:

$$\begin{aligned} (ces_n, r_n)/.s' &= \mathcal{O}_n^d/.s' = \mathcal{O}_n^d/.(su.*.s) = (\mathcal{O}_n^d/.su)/.s = \mathcal{O}^d/.s = \\ &= (d', RESTR[]) \\ ces_n/.s' &= d' \\ r_n/.s' &= RESTR[] \end{aligned}$$

То есть, $s' \in FVS(\mathcal{O}_n^d)$. Кроме того,

$$cres/.s' = cres/.(su.*.s) = (cres/.su)/.s = res/.s = res$$

Последнее равенство выполнено в силу того, что res не содержит s -переменных. Таким образом,

$$d' = ces_n/.s', \quad s' \in FVS(\mathcal{O}_n^d) \quad \text{и} \quad cres/.s' = res.$$

Следовательно (см. соотношения (**)), данные d' являются элементом $o(p, d)$: $d' \in o(p, d)$.

Выше, в пунктах **А** и **В** показано, что $\langle \mathcal{O}^d \rangle = o(p, d)$. Тем самым завершено доказательство корректности алгоритма `nan`:

Теорема 34

Пусть p — произвольная программа на TSG, d — произвольные входные данные для p , процесс вычисления p на d завершается за конечное число n шагов. Тогда алгоритм `nan` на данных $p \ d$ завершается (за n выполнений функции `eval'`) со следующим результатом:

$$\text{nan } p \ d = (\text{res}, \mathcal{O}^d)$$

где res —результат вычисления p на d : $p \ d \xrightarrow{*} \text{res}$, \mathcal{O}^d —окрестность d такая, что:

$$\langle \mathcal{O}^d \rangle = o(p, d) = \{ d' \mid \text{tr}(p, d) = \text{tr}(p, d'), p \ d = p \ d' \}$$

Рассмотрим d_1 и d_2 —данные для программы p такие, что:

$$\text{tr}(p, d_1) = \text{tr}(p, d_2) \text{ и } p \ d_1 = p \ d_2 = \text{res}.$$

Рассмотрим вычисления:

$$\text{nan } p \ d_1 = (\text{res}, \mathcal{O}^{d_1}), \text{nan } p \ d_2 = (\text{res}, \mathcal{O}^{d_2}).$$

В силу теоремы 34 имеем два представления одного и того же множества:

$$\begin{aligned} \langle \mathcal{O}^{d_1} \rangle &= \{ d' \mid \text{tr}(p, d_1) = \text{tr}(p, d'), p \ d_1 = p \ d' \} \\ &= \{ d' \mid \text{tr}(p, d_2) = \text{tr}(p, d'), p \ d_2 = p \ d' \} \\ &= \langle \mathcal{O}^{d_2} \rangle \end{aligned}$$

На самом деле имеет место более сильное утверждение, а именно, утверждение о *текстуальном совпадении* окрестностей: $\mathcal{O}^{d_1} = \mathcal{O}^{d_2}$. Это устанавливается простым анализом текста алгоритма **nan**, из которого видно, что: *окрестность, строимая алгоритмом nan, полностью определяется программой, трассой и результатом эталонного вычисления*. Таким образом, справедливо следующее утверждение.

Утверждение 35

Пусть $p \in R$, d —произвольные входные данные для p , программа p завершается на данных d , $\text{nan } p \ d = (\text{res}, \mathcal{O}^d)$ и $d' \in \langle \mathcal{O}^d \rangle$.

Тогда программа p завершается на данных d' с теми же результатом и трассой, что и на данных d , $\text{nan } p \ d = (\text{res}, \mathcal{O}^{d'})$ и окрестности \mathcal{O}^d и $\mathcal{O}^{d'}$ текстуально совпадают: $\mathcal{O}^d = \mathcal{O}^{d'}$.

4.3.3 Интуитивный смысл окрестностей. Примеры

В разделе 4.1 множество $o(p, d)$ введено как формальный ответ на вопрос: “какая информация о данных d была использована, и какая информация о d не была использована в процессе $p \ d \xrightarrow{*} \text{res}$ обработки d ?”. При помощи **nan** множество $o(p, d)$ всегда может быть представлено в виде окрестности \mathcal{O}^d . Таким образом, эта окрестность и является представлением того, какая информация о данных d была использована в процессе $p \ d \xrightarrow{*} \text{res}$. Насколько данное представление удобно для восприятия человеком? На данный вопрос попытаемся ответить на реальных примерах.

Рассмотрим программу **prog** проверки вхождения подстроки в строку, приведенную на рисунке 1.2 (см. страницу 15).

Замечание. Программа **prog** рассчитана на следующее представление строк: строка из 4 атомов ('A 'B 'C eol)) представляется е-значением:

$$(\text{CONS 'A } (\text{CONS 'B } (\text{CONS 'C eol})))$$

где *eol*—произвольный атом. Обычно в качестве *eol* используют 'NIL, однако в программе **prog** не занимается анализом последнего (терминирующего) атома в представлениях строк и в качестве *eol* может быть использован произвольный атом. Таким образом, существует несколько (эквивалентных с точки зрения логики программы) представлений для строки ('A 'B 'C).

Рассмотрим различные данные для программы **prog**:

```
d1 = [CONS 'A 'NIL, CONS 'A 'NIL]
d2 = [CONS 'A 'NIL, CONS 'B 'NIL]
d3 = [CONS 'X 'NIL, CONS 'Y 'NIL]
d4 = [s4, str]
d5 = [s5, str]
s4 = (CONS 'A (CONS 'B 'NIL))
s5 = (CONS 'Z (CONS 'A 'NIL))
str = (CONS 'X (CONS 'Y (CONS 'Z (CONS 'A (CONS 'B 'NIL)))))
```

Выполнив окрестностный анализ программы **prog** над данными d1, d2, d3, d4, d5, получим следующие результаты⁵:

```
nan prog d1 = ('SUCCESS, Od1)
Od1 = ([CONS A.14 A.17, CONS A.14 E.10], RESTR[])

nan prog d2 = ('FAILURE, Od2)
Od2 = ([CONS A.8 E.4, CONS A.14 A.17], RESTR[A.8:≠:A.14])

nan prog d3 = ('FAILURE, Od3)
Od3 = ([CONS A.8 E.4, CONS A.14 A.17], RESTR[A.8:≠:A.14])

nan prog d4 = ('SUCCESS, Od4)
Od4 = ([CONS A.32 (CONS A.44 A.47),
        CONS A.14 (CONS A.20 (CONS A.26
        (CONS A.32 (CONS A.44 E.40))))),
        RESTR[A.32:≠:A.14, A.32:≠:A.20, A.32:≠:A.26])

nan prog d5 = ('SUCCESS, Od5)
Od5 = ([CONS A.26 (CONS A.38 A.41),
        CONS A.14 (CONS A.20 (CONS A.26 (CONS A.38 E.34)))],
        RESTR[A.26:≠:A.14, A.26:≠:A.20])
```

Замечание. Окрестности O^{d3} и O^{d4} текстуально совпадают—в полном соответствии с утверждением 35,—так как $d4 \in O^{d3}$.

Рассмотрим эти результаты подробнее. Для данных d1 построена окрестность:

⁵ Приводятся результаты реального счета в системы **Gofer**, записанные с использованием ранее определенных сокращений.

$$\begin{aligned} d1 &= [\text{CONS } 'A \text{ 'NIL}, \text{CONS } 'A \text{ 'NIL}] \\ \mathcal{O}^{d1} &= ([\text{CONS } \mathcal{A}.14 \mathcal{A}.17, \text{CONS } \mathcal{A}.14 \mathcal{E}.10], \text{RESTR}[]) \end{aligned}$$

\mathcal{O}^{d1} *наглядно* описывает все возможные вариации данных $d1$, сохраняющие процесс вычисления $\text{prog } d1 \xRightarrow{*} \text{'SUCCESS}$ неизменным: можно оба вхождения атома $'A$ заменить на произвольный атом a_{14} , первый атом $'NIL$ заменить на произвольный атом a_{17} , а последний атом NIL заменить на произвольное е-значение (атом или S-выражение) e_{10} . Вычисление p над так измененными данными пройдет точно так же, как и для $d1$ и даст тот же результат. Какая информация о $d1$ была использована в процессе вычисления prog над $d1$? Были использованы следующие свойства данных $d1$: *$d1$ является списком из двух е-значений, первое из которых имеет вид: $\text{CONS } a_{14} \ a_{17}$, а второе — $\text{CONS } a_{14} \ e_{10}$, где a_{14}, a_{17} — атомы, e_{10} — е-значение.*

Можно значительно повысить наглядность и компактность представления окрестности, подобрав удобную форму обозначения с-переменных.

Определение 30

Позволим изображать са-переменную в виде произвольного *надчеркнутого* атома, над которым надписан индекс этой переменной. Позволим изображать се-переменную в виде произвольного *дважды надчеркнутого* е-значения, над которым надписан индекс этой переменной.

В списке с-выражений ces окрестности $\mathcal{O}^d = (\text{ces}, r)$ все с-переменные запишем в форме, указанной в предыдущем абзаце, используя в качестве *надчеркиваемого* значения при изображении с-переменной v ее значение $v/.s$, получаемое при отождествлении ces с d : $(\text{True}, s) = \text{unify } \text{ces } d$. Так определенную форму записи окрестности \mathcal{O}^d данных d будем называть *эксплицированной формой* записи окрестности. Про надчеркнутые фрагменты данных d в эксплицированной записи \mathcal{O}^d будем говорить, что они *покрыты* с-переменными в окрестности \mathcal{O}^d .

Запишем \mathcal{O}^{d1} в эксплицированной форме:

$$\begin{aligned} d1 &= [\text{CONS } 'A \text{ 'NIL}, \text{CONS } 'A \text{ 'NIL}] \\ \mathcal{O}^{d1} &= ([\text{CONS } \frac{14}{'A} \frac{17}{'NIL}, \text{CONS } \frac{14}{'A} \frac{10}{'NIL}], \text{RESTR}[]) \end{aligned}$$

При использовании эксплицированной формы записи окрестности нет необходимости указывать данные, над которыми окрестность построена (центр окрестности). По определению эксплицированной формы записи, достаточно убрать надчеркивания и индексы, чтобы восстановить центр окрестности. Кроме того, такая форма в явном виде указывает фрагменты в данных, которые можно варьировать — информация о которых не полностью использована в процессе обработки данных. Ниже приведены эксплицированные формы окрестностей \mathcal{O}^{d2} , \mathcal{O}^{d3} , \mathcal{O}^{d4} и \mathcal{O}^{d5} :

$$\begin{aligned} \mathcal{O}^{d2} &= ([\text{CONS } \frac{8}{'A} \frac{4}{'NIL}, \text{CONS } \frac{14}{'B} \frac{17}{'NIL}], \text{RESTR}[\mathcal{A}.8:\neq:\mathcal{A}.14]) \\ \mathcal{O}^{d3} &= ([\text{CONS } \frac{8}{'X} \frac{4}{'NIL}, \text{CONS } \frac{14}{'Y} \frac{17}{'NIL}], \text{RESTR}[\mathcal{A}.8:\neq:\mathcal{A}.14]) \\ \mathcal{O}^{d4} &= ([\text{CONS } \frac{32}{'A} (\text{CONS } \frac{44}{'B} \frac{47}{'NIL}), \end{aligned}$$

$$\begin{aligned} & \text{CONS } \overset{14}{\text{'X}} (\text{CONS } \overset{20}{\text{'Y}} (\text{CONS } \overset{26}{\text{'Z}} (\text{CONS } \overset{32}{\text{'A}} (\text{CONS } \overset{44}{\text{'B}} \overset{40}{\text{'NIL'}})))) \\ &], \text{RESTR}[\mathcal{A}.32:\neq:\mathcal{A}.14, \mathcal{A}.32:\neq:\mathcal{A}.20, \mathcal{A}.32:\neq:\mathcal{A}.26]) \\ \mathcal{O}^{d5} = & ([\text{CONS } \overset{26}{\text{'Z}} (\text{CONS } \overset{38}{\text{'A}} \overset{41}{\text{'NIL'}}), \\ & \text{CONS } \overset{14}{\text{'X}} (\text{CONS } \overset{20}{\text{'Y}} (\text{CONS } \overset{26}{\text{'Z}} (\text{CONS } \overset{38}{\text{'A}} \overset{34}{\text{'(CONS 'B 'NIL)'}})))] , \text{RESTR}[\mathcal{A}.26:\neq:\mathcal{A}.14, \mathcal{A}.26:\neq:\mathcal{A}.20]) \end{aligned}$$

Из рассмотренного можно сделать следующие выводы:

- Окрестность \mathcal{O}^d , полученная в результате окрестностного анализа

$$\text{nan } p \ d \xRightarrow{*} (\text{res}, \mathcal{O}^d),$$

в наглядной форме определяет, какая информация о данных d была использована в процессе $p \ d \xRightarrow{*} \text{res}$, а какая—нет.

- С-переменные в \mathcal{O}^d соответствуют фрагментам данных d , которые можно варьировать, сохраняя неизменным процесс $p \ d \xRightarrow{*} \text{res}$ и его результат. Типы с-переменных и рестрикции в окрестности \mathcal{O}^d накладывают дополнительные ограничения на допустимые вариации данных фрагментов.

4.4 Операции над классами и окрестностями

В этом разделе будут обсуждены операции над классами и окрестностями, которые вместе с окрестностным анализатором **nan** составят законченный набор методов, позволяющий рассматривать окрестностный анализ как средство общего назначения.

4.4.1 Конечное объединение классов

Классы используются для представления множеств D' данных языка TSG, $D' \subseteq D$. В дальнейшем необходимо будет работать с множествами $D' \subseteq D$, которые *непредставимы* в виде класса, но *представимы в виде объединения классов* $\mathcal{C}_1, \dots, \mathcal{C}_n$:

$$D' = \langle \mathcal{C}_1 \rangle \cup \dots \cup \langle \mathcal{C}_n \rangle$$

В этом случае удобно использовать следующие обозначения:

Определение 31

Пусть $\mathcal{C}_1, \dots, \mathcal{C}_n$ —классы. Будем использовать следующие обозначения:

- $[\mathcal{C}_1 + \dots + \mathcal{C}_n]$ —для списка классов $[\mathcal{C}_1, \dots, \mathcal{C}_n]$;
- $\langle \mathcal{C}_1 + \dots + \mathcal{C}_n \rangle$ —для множества $\langle \mathcal{C}_1 \rangle \cup \dots \cup \langle \mathcal{C}_n \rangle$.

4.4.2 Пересечение и разность классов

В [FTP] приведен алгоритм **unifC**, позволяющий по заданным двум классам— C^a — L-класс, C^b —произвольный класс,—построить (см. рисунок 4.3): представление в виде класса для множества $\langle C^b \rangle \cap \langle C^a \rangle$ и представление в виде конечного объединения классов для множества $\langle C^b \rangle \setminus \langle C^a \rangle$. Там же доказаны основные свойства алгоритма (теорема 36 и следствие 37).

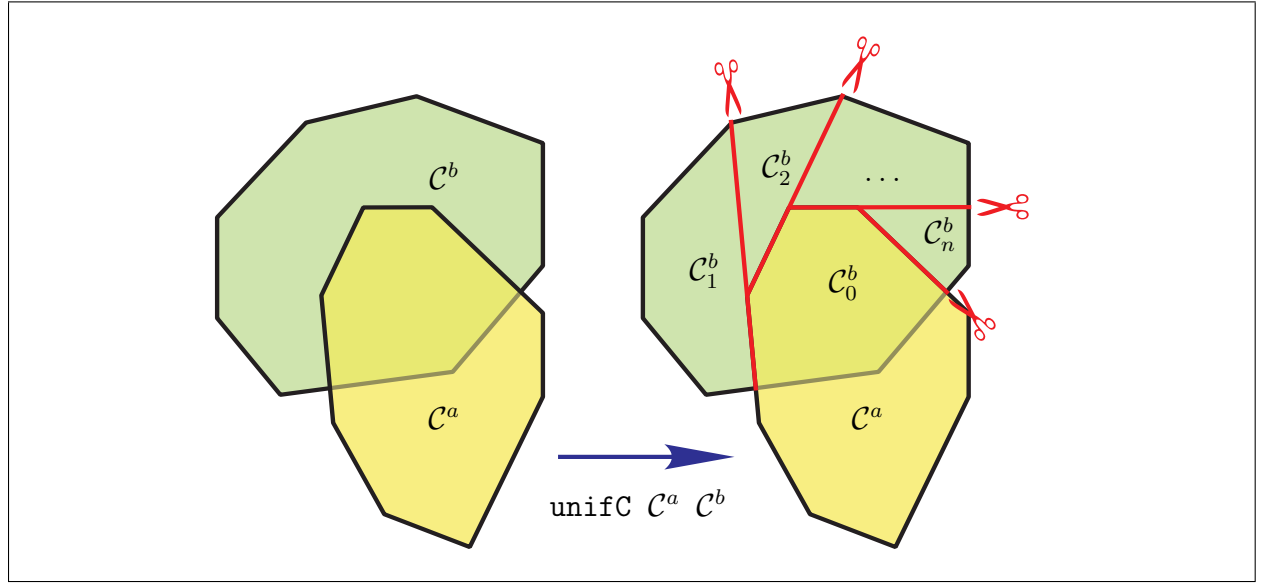


Рис. 4.3: Пересечение и разность классов

Теорема 36

Пусть C^a —L-класс, C^b —произвольный класс. Без ограничения общности будем предполагать, что с-переменные, используемые в L-классе C^a , не используются в классе C^b —если это не так, то всегда можно выполнить изменение индексов (переименование) с-переменных в одном из этих классов, чтобы получить данное свойство. Пусть i —свободный индекс: целое число, большее индекса любой с-переменной из C^a и C^b .

Тогда, за конечное число шагов алгоритм **unifC**:

$$\text{unifC } C^a \ C^b \ i \xRightarrow{*} (\text{su}, [C_0^b, C_1^b, \dots, C_n^b], i')$$

позволяет построить (см. рисунок 4.3): **su**—подстановку для C^a , C_j^b —подклассы C^b , $C_j^b \leq C^b$, $j = 0, \dots, n$, $n \geq 1$, i' —число, большее индекса любой с-переменной из C^a , C^b , $C_0^b, C_1^b, \dots, C_n^b$, такие, что:

1. $\langle C^b \rangle = \langle C_0^b + C_1^b + \dots + C_n^b \rangle$;
2. $\langle C_j^b \rangle \cap \langle C_k^b \rangle = \emptyset$, где $0 \leq j < k \leq n$;
3. $\langle C^b \rangle \cap \langle C^a \rangle = \langle C_0^b \rangle$;
4. $\langle C^b \rangle \setminus \langle C^a \rangle = \langle C_1^b + \dots + C_n^b \rangle$;
5. (а) если $\langle C^b \rangle \cap \langle C^a \rangle = \emptyset$, то $C_0^b = C^b / .\text{emptC}$, $C_1^b = C^b / .\text{idC} = C^b$, C_0^b имеет вид $(\text{ces}_0, \text{INCONSISTENT})$, $\langle C^b \rangle \cap \langle C^a \rangle = \emptyset = \langle C_0^b \rangle$, $\langle C^b \rangle \setminus \langle C^a \rangle = \langle C_1^b \rangle = \langle C^b \rangle$, и $\text{su} = []$;

(б) в противном случае выполнено $\mathcal{C}_0^b \preceq \mathcal{C}_0^a$, причем $\text{bces}' = \text{aces} / .\text{su}$, а каждое неравенство из $\text{ar} / .\text{su}$ входит в br' , где:

- bces' и aces —списки се-выражений;
- br' и ar —рестрикции;

из классов \mathcal{C}_0^b и \mathcal{C}_0^a , соответственно.

Таким образом, в случае непустого пересечения $\langle \mathcal{C}^b \rangle$ и $\langle \mathcal{C}^a \rangle$, множество $\langle \mathcal{C}^b \rangle \cap \langle \mathcal{C}^a \rangle$ представлено классом \mathcal{C}_0^b , который является подклассом как \mathcal{C}^b , так и \mathcal{C}^a .

Следствие 37

Пусть $\mathcal{C}^a, \mathcal{C}^b$ —L-классы; s -переменные, используемые в классе \mathcal{C}^a , не используются в классе \mathcal{C}^b , i —число, большее индекса любой s -переменной из \mathcal{C}^a и \mathcal{C}^b . Тогда, $\mathcal{C}_0^b, \mathcal{C}_1^b, \dots, \mathcal{C}_n^b$ —L-классы, где

$$\text{unifC } \mathcal{C}^a \mathcal{C}^b i \xRightarrow{*} (\text{su}, [\mathcal{C}_0^b, \mathcal{C}_1^b, \dots, \mathcal{C}_n^b], i')$$

Определение 32

Пусть \mathcal{C}^a —L-класс, \mathcal{C}^b —класс и

$$\text{unifC } \mathcal{C}^a \mathcal{C}^b i = (\text{su}, [\mathcal{C}_0^b, \dots, \mathcal{C}_n^b]).$$

Тогда будем использовать обозначения: $\mathcal{C}^b * \mathcal{C}^a$ —для \mathcal{C}_0^b , $\mathcal{C}^b \setminus \mathcal{C}^a$ —для $[\mathcal{C}_1^b + \dots + \mathcal{C}_n^b]$.

Прямым следствием утверждения 37 является следующее:

Утверждение 38

Объединение, пересечение и разность множеств, представленных конечным объединением L-классов, могут быть представлены при помощи конечных объединений L-классов. Конечное пересечение множеств, представленных L-классами, может быть представлено L-классом.

4.4.3 Пересечение окрестностей

Утверждение 39

Пусть \mathcal{O}_1^d и \mathcal{O}_2^d —две окрестности d . Тогда: $\mathcal{O}_1^d * \mathcal{O}_2^d$ —окрестность d .

Доказательство

Пересечение $\langle \mathcal{O}_1^d * \mathcal{O}_2^d \rangle$ —L-класс (в силу следствия 37). В силу определения 28, $d \in \langle \mathcal{O}_i^d \rangle$, $i = 1, 2$. Следовательно, $d \in \langle \mathcal{O}_1^d * \mathcal{O}_2^d \rangle$ и $\mathcal{O}^d = \mathcal{O}_1^d * \mathcal{O}_2^d$ является окрестностью d . ■

Рассмотрим две программы p_1 и p_2 на языке TSG и данное d для этих программ, $\text{nan } p_i d = (\text{res}_i, \mathcal{O}_i^d)$, $i = 1, 2$. Каждая из окрестностей \mathcal{O}_i^d является представлением множества $o(p_i, d)$, $i = 1, 2$ и является описанием того, какая информация о данных d была использована в процессе $p_i d \xRightarrow{*} \text{res}_i$.

Рассмотрим окрестность $\mathcal{O}^d = \mathcal{O}_1^d * \mathcal{O}_2^d$. Тогда:

$$\begin{aligned} \langle \mathcal{O}^d \rangle &= \langle \mathcal{O}_1^d \rangle \cap \langle \mathcal{O}_2^d \rangle = o(p_1, d) \cap o(p_2, d) = \\ &= \{ d' \mid \text{tr}(p_i, d') = \text{tr}(p_i, d), p_i d' = p_i d, i = 1, 2 \} \end{aligned}$$

Следовательно, $\mathcal{O}^d = \mathcal{O}_1^d * \mathcal{O}_2^d$ представляет множество тех и только тех вариаций d' данных d , которые сохраняют неизменными оба процесса вычислений— $p_1 d \xRightarrow{*} \text{res}_1$ и $p_2 d \xRightarrow{*} \text{res}_2$ —и их результаты.

Таким образом окрестность $\mathcal{O}^d = \mathcal{O}_1^d * \mathcal{O}_2^d$ определяет, какая информация о данных d была использована в совокупности в двух процессах— $p_1 d \xRightarrow{*} \text{res}_1$ и $p_2 d \xRightarrow{*} \text{res}_2$ —обработки данных d .

4.4.4 Проверка принадлежности данных классу

Утверждение 40

Для любого данного $d \in D = [EVal]$ и класса $C = (ces, r)$ приведенный ниже алгоритм `isElem` за конечное число шагов определяет, является ли d элементом множества $\langle C \rangle$ или нет.

```
isElem :: [EVal] -> Class -> Bool
isElem d (ces, r) = case (b, r') of
    (True, RESTR[]) -> True
    (_, _) -> False
  where (b,s) = unify ces d
        r'    = r/.s
```

Доказательство

Справедливость утверждения 40 следует из утверждений 5, 6 и определения 12. ■

4.4.5 Декомпозиция окрестности

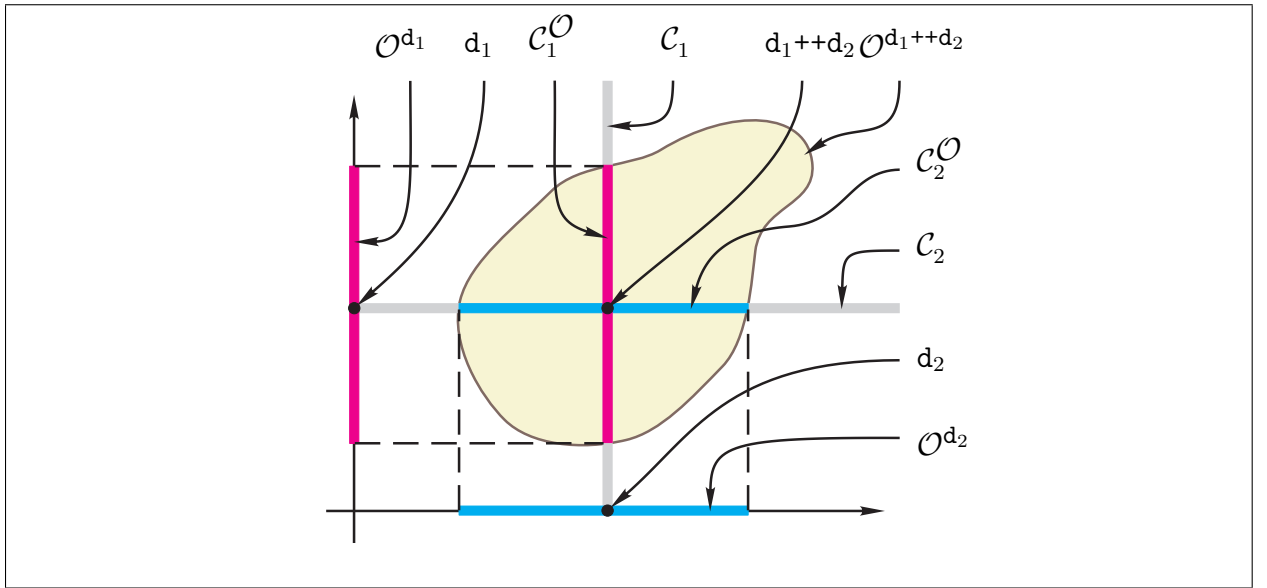


Рис. 4.4: Декомпозиция окрестности

Пусть данные $d \in [EVal]$ имеют вид конкатенации двух фрагментов: $d = d_1 ++ d_2$, $L_1 = \text{length } d_1$, $L_2 = \text{length } d_2$, и пусть $O^{d_1++d_2}$ — окрестность d_1++d_2 , см. рисунок 4.4.

Рассмотрим два списка из различных уникальных (не встречаются в $O^{d_1++d_2}$) се-переменных ces_i , $i = 1, 2$ таких, что $\text{length } ces_i = L_i$. L-классы $C_1 = (ces_1 ++ d_2, RESTR[])$ и $C_2 = (d_1 ++ ces_2, RESTR[])$ представляют множества:

$$\begin{aligned} \langle C_1 \rangle &= \{ d' ++ d_2 \mid d' \in [EVal], \text{length } d' = L_1 \}, \\ \langle C_2 \rangle &= \{ d_1 ++ d' \mid d' \in [EVal], \text{length } d' = L_2 \}. \end{aligned}$$

Следовательно, для L-классов:

```

decompose::Class->[EVal]->[EVal] -> (Class,Class)
decompose o12 d1 d2 = (o1, o2)
  where
    l1 = length d1; l2 = length d2
    i = freeindx 0 o12
    (ces1,i1) = mkCEVs [ ] l1 i
    (ces2,i2) = mkCEVs [ ] l2 i1
    c1 = (ces1++d2, RESTR[])
    c2 = (d1++ces2, RESTR[])
    (_,((ces'1,r1):_)) = unifC c1 o12      -- c1*o12
    (_,((ces'2,r2):_)) = unifC c2 o12      -- c2*o12
    o1 = ( (take l1 ces'1), r1 )             -- взяли первые L1
    o2 = ( (drop l1 ces'2), r2 )             -- откинули первые L1

    -- генератор списка уникальных се-переменных заданной длины
    mkCEVs::CVars->Int->FreeIndx -> (CVars,FreeIndx)
    mkCEVs cvs 0      i = (cvs,i)
    mkCEVs cvs (n+1) i = mkCEVs (cv:cvs) n i'
                          where (cv,i') = mkCEVar i

```

Рис. 4.5: Алгоритм декомпозиции окрестности

$$\mathcal{C}_1^{\mathcal{O}} = \mathcal{C}_1 * \mathcal{O}^{d_1++d_2}, \quad \mathcal{C}_2^{\mathcal{O}} = \mathcal{C}_2 * \mathcal{O}^{d_1++d_2}.$$

выполнено

$$\begin{aligned}
\langle \mathcal{C}_1^{\mathcal{O}} \rangle &= \{ d'++d_2 \mid d'++d_2 \in \langle \mathcal{O}^{d_1++d_2} \rangle \}, \\
\langle \mathcal{C}_2^{\mathcal{O}} \rangle &= \{ d_1++d' \mid d_1++d' \in \langle \mathcal{O}^{d_1++d_2} \rangle \}, \\
d_1++d_2 &\in \langle \mathcal{C}_1^{\mathcal{O}} \rangle \text{ и } d_1++d_2 \in \langle \mathcal{C}_2^{\mathcal{O}} \rangle.
\end{aligned}$$

Так как L-классы $\mathcal{C}_1^{\mathcal{O}}$ и $\mathcal{C}_2^{\mathcal{O}}$ являются подклассами L-классов \mathcal{C}_1 и \mathcal{C}_2 , соответственно, то они имеют вид (см. утверждения 22 и 37):

$$\mathcal{C}_1^{\mathcal{O}} = (\text{ces}'_1++d_2, r_1), \quad \mathcal{C}_2^{\mathcal{O}} = (d_1++\text{ces}'_2, r_2).$$

Рассмотрим L-классы:

$$\mathcal{O}^{d_1} = (\text{ces}'_1, r_1), \quad \mathcal{O}^{d_2} = (\text{ces}'_2, r_2).$$

По построению для них выполнено:

$$\begin{aligned}
\langle \mathcal{O}^{d_1} \rangle &= \{ d' \mid d'++d_2 \in \langle \mathcal{C}_1^{\mathcal{O}} \rangle \} = \{ d' \mid d'++d_2 \in \langle \mathcal{O}^{d_1++d_2} \rangle \} \\
\langle \mathcal{O}^{d_2} \rangle &= \{ d' \mid d_1++d' \in \langle \mathcal{C}_2^{\mathcal{O}} \rangle \} = \{ d' \mid d_1++d' \in \langle \mathcal{O}^{d_1++d_2} \rangle \} \\
d_1 &\in \langle \mathcal{O}^{d_1} \rangle \text{ и } d_2 \in \langle \mathcal{O}^{d_2} \rangle.
\end{aligned}$$

Таким образом, \mathcal{O}^{d_1} —окрестность d_1 и \mathcal{O}^{d_2} —окрестность d_2 .

Определение 33

Определенные выше по окрестности $\mathcal{O}^{d_1++d_2}$ и данным d_1 и d_2 две окрестности \mathcal{O}^{d_1} и \mathcal{O}^{d_2} будем называть *декомпозицией окрестности $\mathcal{O}^{d_1++d_2}$ по d_1 и d_2* и будем обозначать:

$$\text{decompose } \mathcal{O}^{d_1++d_2} \ d_1 \ d_2 = (\mathcal{O}^{d_1}, \mathcal{O}^{d_2})$$

На рисунке 4.5 приведен текст алгоритма декомпозиции окрестности, в точности выполняющий последовательность действий, описанных выше.

Какой смысл имеет операция декомпозиции по данным d_1 и d_2 окрестности $\mathcal{O}^{d_1++d_2}$? Пусть p —программа на языке TSG, первая функция которой имеет список параметров длины L_1+L_2 , а окрестность $\mathcal{O}^{d_1++d_2}$ получена в результате окрестностного анализа:

$$\begin{aligned} \text{nan } p \ (d_1++d_2) &= (\text{res}, \mathcal{O}^{d_1++d_2}) \\ \langle \mathcal{O}^{d_1++d_2} \rangle &= o(p, [d_1, d_2]) = \\ &\{ d' \mid \text{tr}(p, d') = \text{tr}(p, d_1++d_2), \ p \ d' = p \ (d_1++d_2) \} \end{aligned}$$

Для множества $\langle \mathcal{O}^{d_1} \rangle$ выполнено:

$$\begin{aligned} \langle \mathcal{O}^{d_1} \rangle &= \{ d' \mid d'++d_2 \in \langle \mathcal{O}^{d_1++d_2} \rangle \} = \\ &\{ d' \mid \text{tr}(p, d'++d_2) = \text{tr}(p, d_1++d_2), \ p(d'++d_2) = p(d_1++d_2) \} \end{aligned}$$

Таким образом, окрестность \mathcal{O}^{d_1} является представлением множества всех возможных вариаций d' фрагмента d_1 во входных данных d_1++d_2 , сохраняющих неизменным процесс и результат вычисления $p(d_1++d_2) \xRightarrow{*} \text{res}$. Следовательно, окрестность \mathcal{O}^{d_1} определяет, какая информация о фрагменте d_1 данных d_1++d_2 была использована в процессе обработки данных d_1++d_2 .

Аналогичным образом, окрестность \mathcal{O}^{d_2} определяет, какая информация о фрагменте d_2 данных d_1++d_2 была использована в процессе $p(d_1++d_2) \xRightarrow{*} \text{res}$ обработки данных d_1++d_2 .

4.4.6 Примеры пересечения и декомпозиции окрестностей

В разделе 4.3.3 приведены примеры окрестностного анализа программы `prog` над данными:

```
d4 = [s4, str]
d5 = [s5, str]
```

При помощи окрестностного анализа были получены следующие окрестности \mathcal{O}^{d_4} и \mathcal{O}^{d_5} данных d_4 и d_5 (см. страницу 63). После выполнения расслоения данных окрестностей:

$$\begin{aligned} \text{decompose } \mathcal{O}^{d_4} \ [s4] \ [str] &= (\mathcal{O}^{s4}, \mathcal{O}_1^{str}) \\ \text{decompose } \mathcal{O}^{d_5} \ [s5] \ [str] &= (\mathcal{O}^{s5}, \mathcal{O}_2^{str}) \end{aligned}$$

получим следующее⁶:

⁶ Приводятся эксплицированные канонические формы результатов реального счета в системе **Gofer**.

$$\mathcal{O}^{s4} = ([\text{CONS } 'A (\text{CONS } 'B \overset{1}{\text{'NIL}})], \text{RESTR } [])$$

$$\mathcal{O}^{s5} = ([\text{CONS } 'Z (\text{CONS } 'A \overset{1}{\text{'NIL}})], \text{RESTR } [])$$

$$\mathcal{O}_1^{\text{str}} = ([\text{CONS } \overset{1}{\text{'X}}(\text{CONS } \overset{2}{\text{'Y}}(\text{CONS } \overset{3}{\text{'Z}}(\text{CONS } 'A (\text{CONS } 'B \overset{4}{\text{'NIL}}))))], \\ \text{RESTR } [\mathcal{A}.1:\neq:'A, \mathcal{A}.2:\neq:'A, \mathcal{A}.3:\neq:'A])$$

$$\mathcal{O}_2^{\text{str}} = ([\text{CONS } \overset{1}{\text{'X}}(\text{CONS } \overset{2}{\text{'Y}}(\text{CONS } 'Z(\text{CONS } 'A \overset{3}{\text{'(CONS } 'B \text{'NIL})}}))))], \\ \text{RESTR } [\mathcal{A}.1:\neq:'Z, \mathcal{A}.2:\neq:'Z])$$

Окрестность $\mathcal{O}_1^{\text{str}}$ определяет, какая информация о данных **str** была использована в процессе p $[s4, \text{str}] \xRightarrow{*} \text{'SUCCESS}$.

Окрестность $\mathcal{O}_2^{\text{str}}$ определяет, какая информация о данных **str** была использована в процессе p $[s5, \text{str}] \xRightarrow{*} \text{'SUCCESS}$.

Вычислив пересечение этих окрестностей **str**, получим:

$$\mathcal{O}_{1,2}^{\text{str}} = \mathcal{O}_1^{\text{str}} * \mathcal{O}_2^{\text{str}} = \\ = ([\text{CONS } \overset{1}{\text{'X}}(\text{CONS } \overset{2}{\text{'Y}}(\text{CONS } 'Z (\text{CONS } 'A(\text{CONS } 'B \overset{42}{\text{'NIL}}))))], \\ \text{RESTR } [\mathcal{A}.1:\neq:'A, \mathcal{A}.1:\neq:'Z, \mathcal{A}.2:\neq:'A, \mathcal{A}.2:\neq:'Z])$$

Окрестность $\mathcal{O}_{1,2}^{\text{str}}$ определяет, какая информация о данных **str** была использована в совокупности в двух процессах обработки данных **str**:

$$\text{prog } [(\text{CONS } 'A (\text{CONS } 'B \text{'NIL})), \text{str}] \xRightarrow{*} \text{'SUCCESS} \text{ и} \\ \text{prog } [(\text{CONS } 'Z (\text{CONS } 'A \text{'NIL})), \text{str}] \xRightarrow{*} \text{'SUCCESS}.$$

Глава 5

Окрестностное тестирование программ

5.1 Тестирование. Основные понятия

5.1.1 Процесс тестирования программы

Пусть L —язык программирования, $p \in L$ —программа. Программа p определяет некоторое частичное рекурсивное отображение—функцию программы: $p :: D \rightarrow D$, где D —предметная область языка L , то есть множество входных и выходных данных. При разработке программы p программист имеет более или менее формализованное задание на разработку—спецификацию. В общем случае спецификация программы p описывает требования на допустимые входные данные для программы и желаемые соотношения между входными данными и результатом.

Обычно спецификацию ps для программы p рассматривают как пару предикатов $ps = (\varphi, \psi) : \varphi :: D \rightarrow \text{Bool}, \psi :: D \rightarrow D \rightarrow \text{Bool}$.

Определение 34

Программа p *частично корректна* относительно предусловия φ и постусловия ψ , если для любого данного d , такого, что φd и вычисление $p d \xRightarrow{*} \text{res}$ завершается за конечное число шагов,—выполнено постусловие $\psi d \text{ res}$.

Программа p *корректна* относительно предусловия φ и постусловия ψ , если для любого данного d , такого, что φd , справедливо следующее: вычисление $p d \xRightarrow{*} \text{res}$ завершается за конечное число шагов и выполнено постусловие $\psi d \text{ res}$.

Таким образом, полная корректность включает в себя два свойства программы p —частичную корректность и свойство терминируемости программы p (тотальность функции p) на множестве истинности предиката φ . Так как проблема останова алгоритмически неразрешима, то в теории тестирования имеют дело (как правило) с частичной корректностью программ, а в теории верификации программ рассматривают оба понятия корректности программ.

После того как программа разработана, всегда встает вопрос о проверке ее правильности—корректности. То есть, необходимо убедиться, что для всех входных данных выполнено:

$$\forall d \in D ((\varphi d, p d \xRightarrow{*} \text{res}) \Rightarrow (\psi d \text{ res})) \quad (5.1)$$

В тех случаях, когда спецификации заданы формально, имеется *принципиальная* возможность доказать 5.1, как формальную теорему. Однако, осуществить на практике

доказательство корректности программ бывает трудно. В этом случае прибегают к тестированию программы.

Определение 35

Тестирование программы p —это серия выполнений программы (серия *тестовых прогонов программы*) над входными данными (*тестами*) $d_i \in D$, удовлетворяющими предусловию. После каждого выполнения *тестового прогона* программы p $d_i \xRightarrow{*} res_i$ проверяют правильность результата, то есть проверяют истинность постусловия.

Если для тестового прогона программы p с тестом d_i постусловие выполнено, то говорят, что *тест d_i пройден программой p успешно* или, что d_i —*успешный тест для программы p* . В противном случае говорят, что *тест d_i пройден программой p неуспешно* или, что d_i —*неуспешный тест для программы p* .

Для набора тестов $T_n = [d_1, \dots, d_n]$ используют ту же терминологию: если все тесты $d_i \in T_n$ успешно пройдены программой p , то говорят, что *набор тестов T_n пройден программой p успешно*; в противном случае—*набор тестов T_n пройден программой p неуспешно*.

Если пред- и постусловие заданы в виде программ на некотором языке, то их вычисление может выполняться автоматически. Однако на практике достаточно часто используют “неформальное вычисление пред- и постусловия”: группе специалистов предъявляется данное d и результат res вычисления программы на этом данном, и они определяют, соответствует ли пара (d, res) их представлениям о допустимом данном и о желаемом соотношении между входными данным программы p и ее результатом.

Одним из важных вопросов в теории и практике тестирования программ является вопрос построения *критерия выбора тестов*. Этот критерий должен определять, когда некоторый конечный набор (список) тестов $T_n = [d_1, \dots, d_n]$ можно считать *достаточным* для тестирования данной программы p .

Один из классических критериев выбора тестов—*тестирование команд*: набор тестов T_n объявляется достаточным для тестирования программы, если в тестовых прогонах p $d_i \xRightarrow{*} res_i$, $i = 1, \dots, n$ обеспечивается выполнение каждой команды программы p . Проверка данного критерия осуществляется путем анализа только набора тестов T_n и программы p . Таким образом, данный критерий выбора тестов можно рассматривать как предикат от p и T_n . В данном разделе пока ограничимся рассмотрением критериев Cr именно такого класса: $Cr :: P \rightarrow TS \rightarrow Bool$, где $TS = [D]$ —множество наборов (списков) тестов.

Общая схема процесса тестирования программы изображена на рисунке 5.1.

5.1.2 Структурные критерии выбора тестов

Среди критериев выбора тестов, использующих только информацию о наборе тестов и тестируемой программе, наиболее известными являются так называемые *структурные критерии* [28], определяемые ниже.

Тестирование команд. Набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе в совокупности было обеспечено прохождение каждой достижимой команды данной программы.

Тестирование ветвей. Набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе в совокупности было

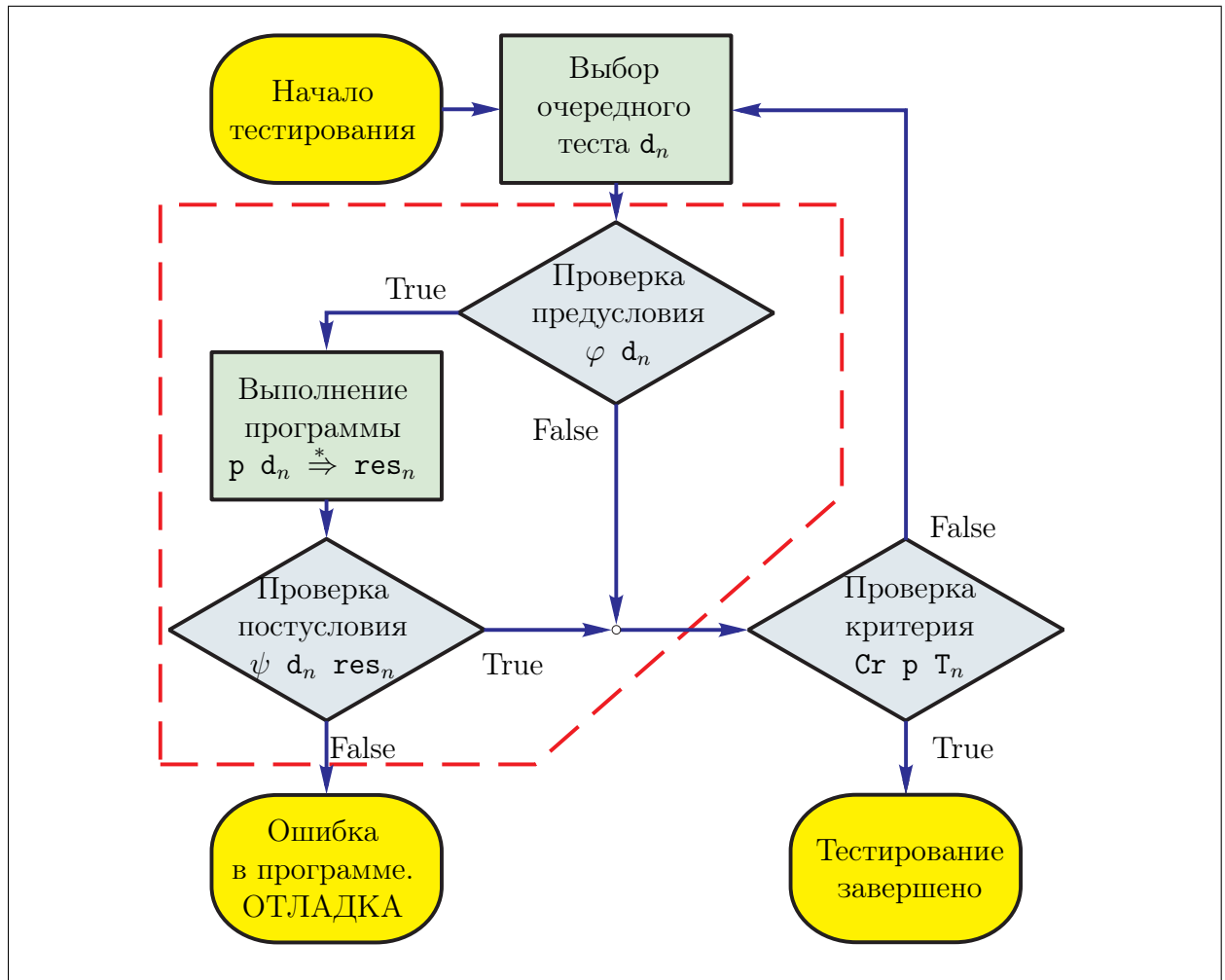


Рис. 5.1: Процесс тестирования программы

обеспечено прохождение каждого реализуемого варианта ветвления для каждой команды ветвления в данной программе.

Тестирование путей. Набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе в совокупности было обеспечено прохождение каждого реализуемого пути в *графе управляющей структуры—блок-схеме*—данной программы, с числом итераций циклов 0, 1 и 2.

Данные критерии являются, пожалуй, самыми распространенными в практике тестирования программ [28, 29, 30, 31], а критерий тестирования ветвей принят во многих организациях (например, в ВВС США [28]) в качестве стандарта при сдаче программ заказчику. В силу неразрешимости проблемы распознавания достижимости команды в программе структурные критерии невычислимы.

5.1.3 Свойства критериев выбора тестов

В теории тестирования изучают следующие свойства критериев выбора тестов.

Полнота. Критерий называют полным, если в случае наличия ошибки в программе существует набор тестов, удовлетворяющий данному критерию и раскрывающий ошибку.

Непротиворечивость (надежность). Любые два набора тестов, удовлетворяющие критерию, должны быть одновременно успешными или неуспешными для данной программы.

Простота проверки—критерий должен “легко” проверяться, например, он должен быть *вычислимым*.

Однако, данные свойства недостижимы на практике. Так, структурные критерии не являются ни надежными, ни вычислимыми. Более того, известно [28], что *не существует* полного непротиворечивого критерия, зависящего от набора тестов и программы, и *не существует* вычислимого полного непротиворечивого критерия, зависящего от набора тестов, программы и спецификаций.

Таким образом, на практике приходится заменять данные свойства их “разрешимыми приближениями”. При этом используются следующие подходы.

Исследование полноты и надежности критериев на ограниченном классе программ и/или ограниченном классе ошибок. Как правило, такому исследованию поддаются очень узкие классы программ и/или ошибок, которые трудно признать имеющими отношение к реальной практике программирования [28].

Статистические исследования “реальной” полноты и надежности критериев—в промышленных системах или на выбранных из практики реальных программах с реальными ошибками [30, 31].

Введение числовой эвристической оценки полноты тестирования вместо проверки невычислимого критерия [32, 33, 34]. Например, в системах тестирования, основанных на структурных критериях, оператору может выдаваться числовой коэффициент—“процент проверенной части” программы.

5.2 Постановка проблемы

5.2.1 Анализ структурных критериев

Идея, лежащая в основе структурных критериев, напоминает изучение лабиринта—необходимо побывать во всех комнатах, пройти все варианты ветвлений и все пути, чтобы *получить полную информацию* о структуре лабиринта. Пусть мы провели несколько тестовых прогонов для тестируемой программы, и во всех них не проходилась некоторый вариант ветвления некоторой команды ветвления. Это означает, что в данных тестовых прогонах *информация* о данном ветвлении в программе *не была использована*. Таким образом, можно предположить, что структурные критерии являются различными формализациями следующего *неформального* критерия выбора тестов:

Неформальный структурный критерий выбора тестов: набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе *в совокупности была использована вся информация о графе управляющей структуры—блок-схеме—данной программы.* (5.2)

Различные подходы к формализации фразы—“использована вся информация о графе. . .”—приводят к различным структурным критериям—тестированию команд, ветвей, путей.

Заметим, что критерий 5.2 явно ориентирован на классические императивные языки. Он неприменим, когда для используемого языка программирования трудно дать

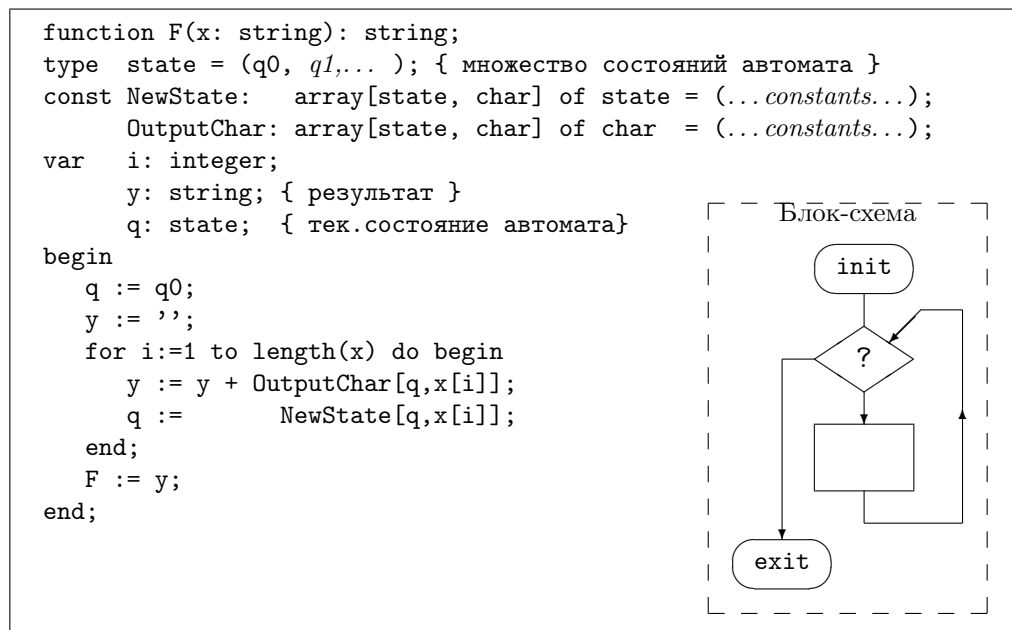


Рис. 5.2: Конечно-автоматное преобразование строки

естественное определение *управляющей структуры программ*. Признание необходимости определения *специальных аналогов* структурных критериев для языков непроедурного типа легко найти в литературе по методам тестирования программ (см., например, [28, стр. 16]).

Теперь обратим внимание на следующее: цель тестирования—проверка *функции программы* на совпадение с желаемой функцией, а полнота проверки в структурных критериях определяется по полноте использования информации об *управляющей структуре* программы. Но разве функция программы полностью определяется только управляющей структурой программы? Конечно, это не так: если взять некоторую программу и произвольно поменять в ней значения различных констант, имена переменных и т.д., не изменяя при этом управляющую структуру программы, то функция программы скорее всего изменится и очень сильно.

Можно ожидать, что структурные критерии будут особенно ненадежны для программ, функция которых “в малой степени” определяется управляющей структурой программы. Убедимся в этом на примере. На рисунке 5.2 приведены управляющая структура—блок-схема—и текст на языке Паскаль функции *F*, реализующей *любое, сколь угодно сложное* конечно-автоматное преобразование строк (конкретный вид преобразования определяется содержимым массивов констант *NewState* и *OutputChar*). Для данной программы структурные критерии оказываются предельно ненадежными: на тестовом наборе из трех тестов $T = [',', 'A', 'AA']$ все команды, все варианты ветвления и все пути, с числом итераций цикла 0, 1 и 2, оказываются пройденными. Однако, совершенно очевидно, что в случае сложного автоматного преобразования и ошибки в программе—в значениях массивов констант *NewState* и *OutputChar*—расхождение между желаемым преобразованием и функцией программы скорее всего не будет обнаружено на таком скромном наборе тестов.

Таким образом, (1) *анализ только управляющей структуры программы для оценки полноты тестирования—одна из возможных причин ненадежности структурных*

критериев, (2) апелляция к понятию “управляющая структура” ограничивает область применения критериев (только классические императивные языки).

5.2.2 Неформальный предельный критерий

Уже в неформальной формулировке 5.2 (страница 76) структурных критериев определены оба, упомянутых выше, недостатка структурных критериев. Как надо изменить неформальный критерий, чтобы можно было ожидать избавления от этих недостатков? Ответ очевиден—для оценки полноты тестирования надо анализировать *то, что* полностью и однозначно определяет функцию программы—то есть, *текст программы целиком!* Таким образом, приходим к следующей формулировке:

Неформальный предельный критерий выбора тестов—набор тестов считается достаточным для тестирования программы, если в процессе тестовых прогонов на данном наборе *в совокупности* было обеспечено использование всей информации о полном тексте данной программы. (5.3)

Обсудим данный критерий, оставив пока в стороне вопрос о способе его формализации.

Предельным критерий 5.3 назван по той причине, что в его формулировке отображен только тот факт, что для оценки полноты тестирования анализируется текст программы и набор тестов. Таким образом, все критерии, основанные на анализе только текста программы и набора тестов, могут рассматриваться как *частные случаи (различные формализации)* данного критерия, и можно ожидать получение критерия с “предельной” надежностью, в случае *правильной* формализации данного критерия.

Заметим, что все структурные критерии являются *ослаблениями (неточными формализациями)* критерия 5.3, так как они основаны на анализе только той *части* текста, которая относится к понятию “управляющая структура”.

В своей неформальной формулировке критерий 5.3 свободен от предположений о способе записи программы—это может быть любой язык программирования или любая другая нотация для записи программ. Важно одно: программа должна быть *представлена в виде текста*.

5.3 Построение окрестностного тестирования

Для построения *формализации* критерия 5.3 будет использован *окрестностный анализ*. В результате будет получен *окрестностный критерий выбора тестов* и основанный на нем новый метод тестирования программ—*окрестностное тестирование*.

5.3.1 Общие предположения

Пусть L —язык программирования, $p \in L$ —программа, которую надо будет тестировать, R —язык реализации. В данном разделе будут определены некоторые предположения о языках L , R и о программе p . Последующее изложение данной главы построено исходя из того, что R , L и p удовлетворяют этим предположениям.

Язык реализации R —некоторый язык программирования, для которого разработан окрестностный анализ, включающий: методы представления множеств данных языка

R —классы и окрестности; алгоритм окрестностного анализа **nan**; операции над классами и окрестностями. Везде далее не будет предполагаться, что R обязательно является языком TSG. Однако, потребуем от реализации окрестностного анализа в R тех же свойств (наличие всех понятий, операций и выполнение утверждений), что были описаны ранее для языка TSG.

L —произвольный алгоритмический язык. В отличие от работ, посвященных тестированию для алгоритмически неполных языков или рассматривающих возможность программных ошибок только некоторого вида, будем допускать алгоритмически полные языки программирования и допускать наличие произвольной ошибки в программе. Не делается никаких предположений о типе языка L —императивный ли он или функциональный и т.п. Единственное предположение—программы в L являются текстами. А точнее, *данные и программы языка L представлены¹ данными языка R* . Это естественное требование, так как будет необходимо рассматривать процесс интерпретации программ $p \in L$: $\text{int}L(p:d) \xrightarrow{*R} \text{res}$ где $\text{int}L \in R$.

Предметная область языка L рекурсивно перечислима. То есть, существует алгоритм перечисления всех данных языка L —алгоритм генерации (возможно бесконечного) списка $[d_1, d_2, \dots]$, со следующими свойствами: элементы списка являются элементами предметной области L , каждый элемент предметной области языка L представлен в этом списке, префикс любой длины n данного списка строится за конечное (зависящее от n) число шагов алгоритма.

Следствием данного предположения является следующее, известное из теории рекурсивных функций [39], утверждение 41 (доказательство утверждения 41 также приведено в [FTP]).

Утверждение 41

Для любой программы p множество $\text{dom}(p)$ данных, на которых программа p определена², рекурсивно перечислимо.

Алгоритмическая перечислимость множества $\text{dom}(p)$, где p —произвольная программа на языке L будет использована при изучении свойств окрестностного тестирования.

Множество допустимых входных данных для программы p представлено некоторым классом D_p . Здесь не предполагается, что D_p представляет множество $\text{dom}(p)$: $\langle D_p \rangle$ может быть шире, чем $\text{dom}(p)$. Задавая класс D_p , тем самым определяют множество данных, которые в принципе могут быть поданы в качестве входных данных программы p . Например, можно рассматривать максимально широкий класс: $\langle D_p \rangle = D$.

5.3.2 Метасистемные уровни в окрестностном тестировании

Как описано в разделе 5.1, процесс тестирования программы $p \in L$ —это серия выполнений *тестовых прогонов* $p \ d_i \xrightarrow{*L} \text{res}_i$ программы над входными данными—*тестами*

¹ При необходимости, такое представление всегда может быть достигнуто при помощи того или иного кодирования.

² То есть, $\text{dom}(p)$ —множество таких d , для которых процесс вычисления программы p завершается за конечное число шагов. Не следует путать множество $\text{dom}(p)$ и множество $\text{Dom}(p)$ *всех возможных входных данных* TSG-программы p , введенное в разделе 4.3.1.

$d_i \in \langle D_p \rangle$ (см. рисунок 5.1 на странице 75). Цель данного раздела—таким образом организовать тестирование программы p , чтобы была получена *формализация* неформального предельного критерия выбора тестов 5.3 (страница 78).

Первый момент, подлежащий формализации в данном критерии, заключен в вопросе: “Что” использует информацию о тексте программы p в процессе тестовых прогонов? Как организовать тестовый прогон, чтобы это “что-то” было явно наблюдаемым? Необходимо процесс $p \ d_i \xrightarrow[L]{*} res_i$, в котором программа p играет активную роль (субъект) а данные d_i —пассивную роль (объект) преобразовать следующим образом: ввести в явное рассмотрение систему \mathcal{X} , выполняющую вычисление $p \ d_i \xRightarrow{*} res_i$, которая бы имела в качестве объектов текст программы p и данные d_i .

Таким образом, необходимо перейти от изначальной формы организации тестового прогона $p \ d_i \xrightarrow[L]{*} res_i$ к новой форме его организации: $\mathcal{X}(p:d_i) \xRightarrow{*} res_i$.

Это хорошо известное в теории программирования преобразование—переход от непосредственного исполнения к интерпретации,—и \mathcal{X} —ничто иное, как интерпретатор языка L : $intL$. После данного преобразования имеем более точный (более формальный) критерий: набор тестов d_1, \dots, d_n считается достаточным для тестирования программы p , если в процессе тестовых прогонов $intL(p:d_i) \xRightarrow{*} res_i$ на данном наборе тестов в совокупности было обеспечено использование интерпретатором $intL$ всей информации о p . Следующий шаг формализации—точное определение понятия “использована информация”. Для него требуется введение метасистемы, наблюдающей, как интерпретатор $intL$ выполняет вычисление $p \ d_i$ и как он при этом потребляет информацию о p . Метасистема такого сорта описана в главе 4—это окрестностный анализатор. Таким образом, приходим к следующей схеме организации процесса тестового прогона $nan \ intL \ (p:d_i)$, где $intL \in R$ —интерпретатора языка L : для любой программы $p \in L$ и данных d выполнено

$$intL \ (p:d) \xrightarrow[R]{*} res \iff p \ d \xrightarrow[L]{*} res.$$

В такой схеме— $nan \ intL \ (p:d_i)$ —организации процесса тестового прогона имеем следующее распределение метасистемных уровней:

1. $(p:d)$ —объекты, данные для $intL$;
2. $intL$ —система, выполняет вычисление $p \ d_i \xRightarrow{*} res_i$;
3. nan —метасистема, наблюдает, как работает система $intL$ на данных $(p:d)$, определяет, какая информация о данных $(p:d)$ при этом используется системой $intL$.

Помимо того, что такое распределение метауровней является необходимым для формализации критерия 5.3, оно еще является и естественным: тестирование программы p —деятельность над программой p , в которой изучаются процессы выполнения программы p на тестовых данных. Поэтому естественно p отнести к объектам в данной деятельности. А поскольку необходимо изучать процессы исполнения программы p , то естественно выделить систему, реализующую данные процессы, и метасистему—в которой собственно и происходит изучение процессов исполнения программы p . Приведенная выше схема организации процесса тестирования соответствует тому, что тестирование—это метадеятельность над программой.

5.3.3 Шаг окрестностного тестирования

Определение 36

Процесс окрестностного тестирования программы p на тестах d_1, d_2, \dots является последовательностью ($i = 1, 2, \dots$) вычислений:

$$\begin{aligned} \text{nan intL } (p:d_i) &\xRightarrow{*} (\text{res}_i, \mathcal{O}_i^{(p:d_i)}) \\ \text{decompose } \mathcal{O}_i^{(p:d_i)} [p] d_i &\xRightarrow{*} (\mathcal{O}_i^p, \mathcal{O}_i^{d_i}) \\ \overline{\mathcal{O}}_i^p &= \mathcal{O}_1^p * \mathcal{O}_2^p * \dots * \mathcal{O}_i^p \end{aligned}$$

со следующими результатами на каждом шаге i :

- res_i —результат вычисления $\text{intL } (p:d_i)$,
- $\mathcal{O}_i^{(p:d_i)}$ —окрестность $(p:d_i)$,
- \mathcal{O}_i^p и $\overline{\mathcal{O}}_i^p$ —окрестности p ,
- $\mathcal{O}_i^{d_i}$ —окрестность d_i .

В силу свойств окрестностного анализатора **nan** (теорема 34) и по определению intL , res_i является результатом вычисления программы p на данных d_i : $p d_i \xRightarrow{*}_L \text{res}_i$.

В силу свойств **nan**, свойств операций декомпозиции и пересечения окрестностей, выполнены следующие утверждения о полученных окрестностях (везде будем предполагать, что все классы и окрестности представлены в каноническом виде).

Окрестность $\mathcal{O}_i^{(p:d_i)}$ определяет, какая информация о $(p:d_i)$ была использована в процессе i -того тестового прогона программы p на данных d_i : $\text{intL } (p:d_i) \xRightarrow{*}_R \text{res}_i$. Окрестность \mathcal{O}_i^p определяет, какая информация о программе p была использована в процессе i -того тестового прогона программы p на данных d_i . Окрестность $\mathcal{O}_i^{d_i}$ определяет³, какая информация о тесте d_i была использована в процессе i -того тестового прогона программы p на данных d_i . Окрестность $\overline{\mathcal{O}}_i^p$ определяет, какая информация о программе p была использована *в совокупности* в процессах тестовых прогонов программы p на данных d_1, \dots, d_i .

Заметим, что по определению $\overline{\mathcal{O}}_k^p$ выполнено:

$$\overline{\mathcal{O}}_1^p = \mathcal{O}_1^p, \quad \overline{\mathcal{O}}_{k+1}^p = \overline{\mathcal{O}}_k^p * \mathcal{O}_{k+1}^p, \quad k = 1, 2, \dots$$

В силу теоремы 36, выполнено: $\overline{\mathcal{O}}_1^p \succeq \overline{\mathcal{O}}_2^p \succeq \dots \succeq \overline{\mathcal{O}}_k^p \succeq \overline{\mathcal{O}}_{k+1}^p \succeq \dots$

В силу теоремы 33 в данной последовательности окрестностей не может быть бесконечного числа *собственных подокрестностей*—бесконечного числа отношений (\succ).

5.4 Окрестностный критерий выбора тестов

Определение 37

Назовем окрестность $\overline{\mathcal{O}}_k^p$ *неподвижной окрестностью* программы p , если независимо от выбора теста d_{k+1} выполнено: $\overline{\mathcal{O}}_{k+1}^p = \overline{\mathcal{O}}_k^p$.

³ Использование $\mathcal{O}_i^{d_i}$ будет обсуждено в разделе 5.6.1.

Пусть $\bar{\mathcal{O}}_k^p$ — неподвижная окрестность программы p . Что это значит? Как сказано выше, $\bar{\mathcal{O}}_k^p$ определяет, какая информация о программе p была использована в совокупности в процессах тестовых прогонов программы p на данных d_1, \dots, d_k . Выберем любой тест d_{k+1} . Тогда $\bar{\mathcal{O}}_{k+1}^p$ определяет, какая информация о программе p была использована в совокупности в процессах тестовых прогонов программы p на данных d_1, \dots, d_k, d_{k+1} . Равенство $\bar{\mathcal{O}}_k^p = \bar{\mathcal{O}}_{k+1}^p$ означает, что эти две окрестности определяют одни и те же свойства программы p . То есть, для любого d_{k+1} никакая новая информация о p не может быть использована в процессе вычислений p на тесте d_{k+1} . Другими словами, вся информация о p , которая необходима для проведения вычисления программы p на любых данных, была использована в тестовых прогонах программы p на тестах d_1, \dots, d_k .

Следовательно, неподвижность $\bar{\mathcal{O}}_k^p$ означает, что в тестовых прогонах программы p на тестах d_1, \dots, d_k, d_{k+1} была использована вся доступная⁴ информация о p . Тем самым завершена формализация неформального предельного критерия 5.3 выбора тестов.

Определение 38

Окрестностный критерий тестирования. Набор (конечный список) тестов

$$T_k = [d_1, \dots, d_k]$$

будем считать достаточным для тестирования программы p , если $\bar{\mathcal{O}}_k^p$ является неподвижной окрестностью p .

5.5 Свойства окрестностного тестирования

Теорема 42

Для любой программы p существует конечный набор тестов T , удовлетворяющий окрестностному критерию тестирования.

Доказательство

Рассмотрим любой алгоритм перечисления (см. утверждение 41) множества $\text{dom}(p)$: $\text{dom}(p) = [d_1, d_2, \dots]$. Если выполнять окрестностное тестирование на данных d_i , $i = 1, 2, \dots$, то получим последовательность окрестностей: $\bar{\mathcal{O}}_1^p \succeq \bar{\mathcal{O}}_2^p \succeq \dots$. В этой последовательности не может быть бесконечного числа отношений (\succ) — теорема 33. То есть, существует номер n , такой, что: $\bar{\mathcal{O}}_n^p = \bar{\mathcal{O}}_j^p$ для всех $j \geq n$. Таким образом, $\bar{\mathcal{O}}_n^p$ — неподвижная окрестность, а конечный набор тестов $T = [d_1, \dots, d_n]$ удовлетворяет окрестностному критерию тестирования. ■

Теорема 43

Окрестностный критерий тестирования программ — полный: в случае наличия ошибки в программе p существует конечный набор тестов T' , раскрывающий ошибку и удовлетворяющий окрестностному критерию.

Доказательство

Пусть существует ошибка в программе p . Это значит, что вычисление программы p на некоторых данных d приводит к неверному результату. Рассмотрим любой конечный набор тестов T , удовлетворяющий окрестностному критерию тестирования (см.

⁴ То есть, вся информация о p , которая в принципе может использоваться интерпретатором `intL` при вычислении программы p на любых данных.

теорему 42). Добавим к этому набору тест d : $T' = [d] ++ T$. Конечный набор тестов T' по построению удовлетворяет окрестностному критерию тестирования и раскрывает ошибку в программе. ■

Как известно (см. раздел 5.1.3), не существует полного надежного (непротиворечивого) критерия, зависящего от набора тестов и программы. Поэтому рассмотрение вопроса о надежности окрестностного критерия будет отложено до раздела 5.6.3, где будет показано, что критерий окрестностного тестирования программы p с заданной спецификацией ps не только полный (теорема 43), но и непротиворечивый. Следовательно (см. раздел 5.1.3), *критерий окрестностного тестирования алгоритмически неразрешим*. То есть, невозможно в общем случае реализовать при помощи алгоритма проверку неподвижности окрестности \bar{O}_n^p .

Итак, в общем случае окрестностный критерий тестирования должен проверяться экспертно. В некоторых случаях неподвижность \bar{O}_n^p может быть очевидна. Например, если окрестность \bar{O}_n^p “сжалась в точку”: $\bar{O}_n^p = \text{sngl}(p)$, $\langle \bar{O}_n^p \rangle = \{p\}$, то \bar{O}_n^p — заведомо неподвижная окрестность. В разделе 5.6.1 приведено еще одно легко проверяемое условие, наступление которого гарантирует, что \bar{O}_n^p неподвижная.

В остальных случаях остается полагаться на то, что предлагаемые экспертам окрестности \bar{O}_n^p имеют достаточно наглядную форму, что позволит им проверять, является ли данная окрестность неподвижной или нет. В [FTP] рассматривается возможность введения эвристической числовой оценки близости окрестности \bar{O}_n^p к неподвижной окрестности.

Упомянем еще одно свойство окрестностного тестирования — для него несущественны *характеристики языка* L . Этот язык может быть императивным, объектно-ориентированным или функциональным с аппликативной или ленивой семантикой и т.п. В любом случае, если $\text{int}L$ задан, nan будет следить за тем, как $\text{int}L$ потребляет информацию о текстах p и d .

5.6 Решение традиционных проблем теории тестирования

В данном разделе будет исследована возможность решения в рамках окрестностного тестирования следующих традиционных проблем теории тестирования.

Выбор очередного теста. Предположим, что уже успешно выполнено тестирование программы p для набора тестов $T_n = [d_1, \dots, d_n]$, и критерий тестирования не выполнен на наборе T_n . Как выбрать следующий тест d_{n+1} ? Естественно, выбрать его так, чтобы было достигнуто продвижение в сторону выполнения рассматриваемого критерия тестирования. Так, для структурных критериев система тестирования должна отслеживать множество непройденных команд (ветвей, путей) и как-то пытаться строить (проблема в общем случае алгоритмически неразрешима) тесты, реализующие прохождение этих команд (ветвей, путей).

Тестирование и отладка. Предположим, что успешно выполнено тестирование программы p на наборе тестов $T_n = [d_1, \dots, d_n]$, и неуспешно — на тесте d_{n+1} . В программе имеется ошибка, тестирование прекращается, начинается отладка. После внесения исправления в программу — переход от программы p к программе p' — приступают к тестированию новой программы p' . В описанной ситуации возникают следующие вопросы

[28]:

1. Позволяет ли факт успешного прохождения тестов T_n и неуспешного прохождения теста d_{n+1} локализовать ошибку (уменьшить зону поиска в тексте программы места внесения исправления)?
2. После внесения исправления в программу нужно ли тестирование программы p' начинать “с нуля” или можно гарантировать, что на части тестов из T_n программа p' будет успешно работать? Как отобрать из T_n эти тесты?

Использование спецификаций при тестировании программ. Если задана формальная спецификация программы (описание решаемой задачи), то возможно основывать выбор тестов не только на анализе текста программы, но и на анализе текста спецификаций. В случае окрестностного анализа хотелось бы достичь формализации следующего критерия: *на наборе тестов T_n при проверки предусловий, тестовых прогонах и проверке постусловий должна быть использована вся информация о тексте программы и тексте спецификаций.*

5.6.1 Выбор очередного теста

Для выбора очередного теста d_{n+1} будут использованы окрестности $\mathcal{O}_i^{d_i}$ над тестами d_i . Заметим, что если взять d_{n+1} из множества $\langle \mathcal{O}_i^{d_i} \rangle$, $i \in [1..n]$, то получим (по определению операции декомпозиции окрестности—раздел 4.4.5—и в силу утверждения 35):

$$\begin{aligned} (p:d_{n+1}) \in \mathcal{O}_i^{(p:d_i)} &\Rightarrow \\ \text{tr}(\text{int}, (p:d_{n+1})) &= \text{tr}(\text{int}, (p:d_i)), \quad p \cdot d_{n+1} = p \cdot d_i = \text{res}_i, \\ \mathcal{O}_{n+1}^{(p:d_{n+1})} &= \mathcal{O}_i^{(p:d_i)}, \quad \mathcal{O}_{n+1}^p = \mathcal{O}_i^p, \quad \overline{\mathcal{O}}_{n+1}^p = \overline{\mathcal{O}}_n^p. \end{aligned}$$

Таким образом, вычисление p на d_{n+1} пройдет точно так же, как и на d_i , не будет использовано никакой новой информации о p .

Вывод. Не следует выбирать очередной тест d_{n+1} из множеств $\langle \mathcal{O}_i^{d_i} \rangle$, $i = 1, \dots, n$. Тест d_{n+1} необходимо выбирать из следующего множества, представленного конечным объединением классов (см. утверждение 38):

$$\begin{aligned} \langle D_p \rangle \setminus (\langle \mathcal{O}_1^{d_1} \rangle \cup \dots \cup \langle \mathcal{O}_n^{d_n} \rangle) &= \langle Q_n \rangle, \text{ где} \\ Q_n &= D_p \setminus [\mathcal{O}_1^{d_1} + \dots + \mathcal{O}_n^{d_n}] = [C_1^Q + \dots + C_{k_n}^Q]. \end{aligned}$$

Поэтому, в качестве новых тестов, после выполнения тестовых прогонов на наборе тестов $T_n = [d_1, \dots, d_n]$ разумно взять k_n новых тестов, по одному из каждого класса C_i^Q , $i = 1, \dots, k_n$.

Утверждение 44

Если $\langle Q_n \rangle = \emptyset$, то $\overline{\mathcal{O}}_n^p$ неподвижная окрестность.

Доказательство

Рассмотрим произвольное данное $d_{n+1} \in \langle D_p \rangle$ для программы p . Так как $\langle Q_n \rangle = \emptyset$, то $\langle D_p \rangle = \langle [\mathcal{O}_1^{d_1} + \dots + \mathcal{O}_n^{d_n}] \rangle$, существует $i \in [1..n]$ такое, что $d_{n+1} \in \langle \mathcal{O}_i^{d_i} \rangle$, что (как показано выше) влечет $\overline{\mathcal{O}}_{n+1}^p = \overline{\mathcal{O}}_n^p$. Следовательно, окрестность $\overline{\mathcal{O}}_n^p$ неподвижная. ■

5.6.2 Окрестностное тестирование и отладка программ

Локализация места ошибки. Предположим, что успешно пройден набор тестов T_n и неуспешно—тест d_{n+1} . Программа p содержит ошибку. Где прежде всего следует ее искать?

Предположим, что трассы $\text{tr}(\text{int}, (p:d_i))$, так же как и результаты вычислений $p \ d_i = \text{res}_i$, $i \in [1..n]$ являются *правильными*, а именно, мы хотим так поправить программу, чтобы не изменились не только результаты, но и трассы успешных прогонов на данных d_1, \dots, d_n . В таком предположении исправленная программа p' должна удовлетворять требованию $p' \in \bar{\mathcal{O}}_n^p$. Это означает, что исправления должны вноситься во фрагменты p , представленные в окрестности $\bar{\mathcal{O}}_n^p$ с-переменными—надчеркнутые фрагменты в эксплицированной записи $\bar{\mathcal{O}}_n^p$. С другой стороны, результат $p' \ d_{n+1}$ должен отличаться от неверного результата $p \ d_{n+1}$. Таким образом, $p' \notin \langle \mathcal{O}_{n+1}^p \rangle$, т.е. $p' \in \langle \bar{\mathcal{O}}_n^p \rangle \setminus \langle \mathcal{O}_{n+1}^p \rangle$. Последнее еще больше локализует места возможных исправлений. Итак, в случае приведенного выше предположения (о желании сохранить трассы), окрестностный анализ явно указывает нам, *какие фрагменты p должны оставаться неизменными, а какие фрагменты p обязаны быть изменены* при исправлении программы:

исправления должны вноситься в надчеркнутые фрагменты p , в эксплицированной записи $\bar{\mathcal{O}}_n^p \setminus \mathcal{O}_{n+1}^p$.

В случае, если нет уверенности в необходимости сохранения трасс успешных прогонов на наборе T_n , надо данную рекомендацию о местах внесения исправлений рассматривать как *эвристику*, с приведенным выше обоснованием, и начинать поиск ошибки именно с этих мест.

Продолжение окрестностного тестирования после исправления ошибки. Предположим, что ошибка найдена, программа p исправлена, p' —новый текст программы. Нужно ли повторять все вычисления p' на *всех* тестах d_1, \dots, d_n ?

Если выполнено условие $p' \in \langle \bar{\mathcal{O}}_i^p \rangle$, где $i \in [1..n]$, то нет необходимости выполнять вычисление p' на тесте d_i . Действительно:

$$\begin{aligned} p' \in \langle \bar{\mathcal{O}}_i^p \rangle &\Rightarrow (p':d_i) \in \langle \mathcal{O}_i^{(p:d_i)} \rangle \Rightarrow \\ \text{tr}(\text{intL}, (p':d_i)) &= \text{tr}(\text{intL}, (p:d_i)), \ p' d_i = p \ d_i = \text{res}_i, \ \mathcal{O}_i^{(p':d_i)} = \mathcal{O}_i^{(p:d_i)}, \\ \mathcal{O}_i^{p'} &= \mathcal{O}_i^p. \end{aligned}$$

Таким образом, необходимо только пересчитать $\mathcal{O}_i^{d_i}$:

$$\text{decompose } \mathcal{O}_i^{(p':d_i)} \ [p'] \ d_i = (\mathcal{O}_i^{p'}, \mathcal{O}_i^{d_i})$$

В тех случаях, когда программа p достаточно сложная, указанная выше проверка $p' \in \langle \bar{\mathcal{O}}_i^p \rangle$ и пересчет $\mathcal{O}_i^{d_i}$ займет значительно меньшее время, чем выполнение вычисления p' на тесте d_i .

5.6.3 Использование в тестировании спецификации программы

Пусть S —язык высокого уровня для написания спецификаций программ, φ —предусловие, и ψ —постусловие для тестируемой программы $p \in L$, $\varphi, \psi \in S$ —одно и двухместный

```

job    ::= (JOB [ jvar1, ..., jvarn ] jterm )           -- n ≥ 0

jterm  ::= (ExecS progS [ jexp1, ..., jexpn ] jvar      -- n ≥ 0
           jterm)                                       -- progS ∈ S
           | (ExecL progL [ jexp1, ..., jexpn ] jvar    -- n ≥ 0
           jterm)                                       -- progL ∈ L
           | (IF jexp jterm1 jterm2)
           | (EXIT e-exp)

jvar   ::= (Var name)

jexp   ::= jvar
           | (Const d)                                -- d ∈ D

jbind  ::= (jvar := d)                                -- d ∈ D

jenv   ::= [ jbind1, ..., jbindn ]                    -- n ≥ 0

```

Рис. 5.3: Синтаксис конструкций языка JobSL

предикаты на D:

$$(\varphi [d]) \in \{\text{"True"}, \text{"False"}\}, (\psi [d, \text{res}]) \in \{\text{"True"}, \text{"False"}\}$$

где "True", "False"—две различные константы (значения) из D.

В данном разделе будет описана организация окрестностного тестирования программы p с использованием заданной спецификации $ps = (\varphi, \psi)$. Предположим, что S —язык программирования и имеется $\text{int}S \in R$ —интерпретатор языка S , написанный на языке R .

Тогда большинство пунктов шага тестирования, а именно все пункты рисунка 5.1 (страница 75), обведенные в пунктирную рамку, могут вычисляться. Таким образом, процесс тестирования программы p с заданной спецификацией $ps = (\varphi, \psi)$ можно рассматривать как процесс тестирования составной программы $c(\varphi, p, \psi)$, соответствующей указанному фрагменту рисунка, с ожидаемой функцией: "True" на всей области D.

Будет удобно рассмотреть данную составную программу $c(\varphi, p, \psi)$ явно, как программу, написанную на некотором языке JobSL, являющемся *надстройкой, языком заданий* над языками S , L . На рисунке 5.3 приведен синтаксис простого языка JobSL, пригодного для написания $c(\varphi, p, \psi)$. Семантика языка JobSL очевидна; формально она определена интерпретатором intJobSL , приведенным на рисунках 5.6 и 5.5 (подробный разбор процесса вычисления интерпретатором intJobSL программы job на данных d можно найти в [FTP]). На рисунке 5.4 приведена программа $c(\varphi, p, \psi)$ на языке JobSL, соответствующая действиям⁵ по обработке теста d_i при тестировании программы p с заданной спецификацией $ps = (\varphi, \psi)$.

⁵ См. обведенный в пунктирную рамку фрагмент рисунка 5.1.


```

corr:: (S,L,S) -> JobSL

corr(phi,p,psi)=
    (JOB [ Var"d" ]

        (ExecS phi [Var"d"] {-->-} (Var"phi d")
        (IF (Var "phi d")
        -- THEN
            (ExecL p [Var"d"] {-->-} (Var"res")
            (ExecS psi [Var"d",Var"res"] {-->-} (Var"psi d res")
            (IF (Var "psi d res")
            -- THEN
                (EXIT (Const "True"))
            -- ELSE
                (EXIT (Const "False")) -- Error!
            )))
        -- ELSE
            (EXIT (Const "True"))
    ) ) -- End of JOB

```

Рис. 5.4: Программа $\text{corr}(\varphi, p, \psi)$ на языке JobSL

```

instance Eq JBind where
    ((Var n1):=_ ) == ((Var n2):=_ ) = (n1==n2)

mkJEnv :: [JVar] -> [D] -> JEnv
mkJEnv jvs ds = zipWith (\ jv d -> (jv:=d)) jvs ds

(+.) :: JEnv -> JEnv -> JEnv
je1 +. je2 = nub (je2++je1)

calcJExp :: JExp -> JEnv -> D
calcJExp (Const d) je=d
calcJExp (Var name) je=head[ d | ((Var n):=d)<-je, n==name]

calcJExps :: [JExp] -> JEnv -> [D]
calcJExps jexs je = map (\ jex -> calcJExp jex je) jexs

```

Рис. 5.5: Вспомогательные функции интерпретатора языка JobSL

Замечание. Конечно, в соответствии с разделом 5.3.1, синтаксические конструкции языка JobSL должны быть представлены данными из D , интерпретатор intJobSL должен быть написан (по заданным интерпретаторам $\text{IntS} \in R$ и $\text{IntL} \in R$) на языке R . Так как в данной главе не фиксирован язык реализации, то для конкретности изло-

```

intJobSL :: JobSL->[D]->D
intJobSL job@(JOB jvs jt0) d = evalJ jt0 je0
                                where je0 = mkJEnv jvs d

evalJ :: JTerm -> JEnv -> D
evalJ (ExecS p jexs jv jt') je = evalJ jt' je'
                                where args = calcJExps jexs je
                                      res  = intS p args
                                      je'  = je +. [jv := res]

evalJ (ExecL p jexs jv jt') je = evalJ jt' je'
                                where args = calcJExps jexs je
                                      res  = intL p args
                                      je'  = je +. [jv := res]

evalJ (IF jex jt1 jt2) je =
    if ((calcJExp jex je) == "True")
    then evalJ jt1 je
    else evalJ jt2 je

evalJ (EXIT jex) je = calcJExp jex je

intL :: L->[D]->D
-- ... описание интерпретатора языка L
intS :: S->[D]->D
-- ... описание интерпретатора языка S

```

Рис. 5.6: Интерпретатор языка JobSL

жения синтаксис языка JobSL выше отображен в предметную область языка Gofer, и текст интерпретатора intJobSL записан на языке Gofer. Для каждого частного языка реализации эти описания необходимо *адекватно отобразить* с языка Gofer в язык R.■

Рассмотрим окрестностное тестирование $c(\varphi, p, \psi)$:

$$\begin{aligned}
 & \text{nan intJobSL } (c(\varphi, p, \psi):d_i) \stackrel{*}{\Rightarrow} (\text{res}_i, \mathcal{O}_i^{(c(\varphi, p, \psi):d_i)}) \\
 & \text{decompose } \mathcal{O}_i^{(c(\varphi, p, \psi):d_i)} [c(\varphi, p, \psi)] d_i \stackrel{*}{\Rightarrow} (\mathcal{O}_i^{c(\varphi, p, \psi)}, \mathcal{O}_i^{d_i}) \\
 & \overline{\mathcal{O}}_i^{c(\varphi, p, \psi)} = \mathcal{O}_1^{c(\varphi, p, \psi)} * \mathcal{O}_2^{c(\varphi, p, \psi)} * \dots * \mathcal{O}_i^{c(\varphi, p, \psi)} \\
 & i = 1, 2, \dots
 \end{aligned}$$

Единственным допустимым результатом тестовых прогонов в данном случае является значение $\text{res}_i = \text{"True"}$, то есть проверка правильности результатов res_i может выполняться автоматически. К тестированию программы $c(\varphi, p, \psi)$ применимы все результаты предыдущих разделов данной главы. Заметим, что текст $c(\varphi, p, \psi)$ включает в себя полный текст p и $\text{ps} = (\varphi, \psi)$, вычисление $c(\varphi, p, \psi) d_i$ в общем случае включает в себя проверку предусловия φd_i , вычисление программы $p d_i \stackrel{*}{\Rightarrow} \text{res}_i$ и проверку постусловия $\psi d_i \text{res}_i$, интерпретатор IntJobSL для проведения вычислений в языках

S и L обращается к интерпретаторам $\text{Int}S$ и $\text{Int}L$. Поэтому окрестностный критерий (для завершения тестирования потребуется использование в совокупности в процессах вычислений $c(\varphi, p, \psi)$ на тестах d_i всей информации о тексте программы $c(\varphi, p, \psi)$) в данном случае равносильно следующему:

Набор тестов $T_n = [d_1, \dots, d_n]$ будет считаться достаточным для тестирования p с заданной спецификацией $ps = (\varphi, \psi)$, если на данных тестах в совокупности в процессах:

- проверок предусловий φ d_i ;
- вычисления программы p d_i ;
- проверок постусловий ψ d_i res_i ;

была использована вся доступная информация о тексте программы и о тексте спецификации.

Окрестностное тестирование программы $c(\varphi, p, \psi)$ является естественным способом использования при тестировании программы p информации и о самой программе и о формально заданной спецификации $ps = (\varphi, \psi)$. Таким образом, в рамках окрестностного тестирования естественно решается данная традиционная проблема теории тестирования (см. раздел 5.6). Это обеспечено тем, что в окрестностном тестировании не предполагается никаких свойств о языке, на котором написана тестируемая программа (в данном случае—программа $c(\varphi, p, \psi)$ на “составном” языке JobSL), кроме следующих: тестируемая программа является текстом (который в свою очередь представлен данными языка R), а семантика языка описана интерпретатором, реализованным на языке R .

Теорема 45

Окрестностный критерий тестирования программы p с заданной спецификацией $ps = (\varphi, \psi)$ является надежным. Пусть выполнено окрестностное тестирование программы $c(\varphi, p, \psi)$ на конечном наборе тестов T_n , результаты всех тестовых прогонов $\text{res}_i = \text{"True"}$, $i = 1, \dots, n$ и выполнен критерий окрестностного тестирования для набора тестов T_n . Тогда, программа p частично корректна по отношению к предусловию φ и постусловию ψ .

Доказательство

Необходимо доказать, что ни для каких данных d не может быть выполнено $c(\varphi, p, \psi) d = \text{"False"}$. По условиям теоремы результаты всех тестовых прогонов $\text{res}_i = \text{"True"}$, то есть фрагмент текста $c(\varphi, p, \psi)$, соответствующий j -терму $(\text{EXIT} (\text{Const} \text{"False"}))$ не использовался в вычислениях $c(\varphi, p, \psi)$ на d_1, \dots, d_n . В силу определения IntJobSL следует, что в неподвижной окрестности $\overline{O}_n^{c(\varphi, p, \psi)}$ указанный фрагмент текста целиком покрыт s -переменной.

По определению 37, не зависимо от выбора теста d_{n+1} выполнено

$$\overline{O}_{n+1}^{c(\varphi, p, \psi)} = \overline{O}_n^{c(\varphi, p, \psi)}.$$

То есть, независимо от выбора данных d_{n+1} в вычислении $c(\varphi, p, \psi) d_{n+1} \xrightarrow{*} \text{res}_{n+1}$ никогда не потребуется вычислять значение j -терма $(\text{EXIT} (\text{Const} \text{"False"}))$. ■

Итак, окрестностный критерий тестирования программы p при заданной спецификации $ps = (\varphi, \psi)$ не только полный (теорема 43), но и надежный. В силу того, что не существует вычислимого полного непротиворечивого критерия, зависящего от набора тестов, программы и спецификаций [28], выполнено следующее:

Следствие 46

Неразрешимость окрестностного критерия тестирования. Не существует алгоритма, который определяет, является ли окрестность $\bar{\mathcal{O}}_n^p$ неподвижной.

Глава 6

Универсальный решающий алгоритм

6.1 Постановка проблемы

Пусть $p \in R$ —программа на языке TSG, $X \subseteq D$ —множество данных для программы p , $y \in \text{Eval}$ —е-значение. Рассмотрим множество

$$X_{p^{-1}y} = \{ x \mid x \in X, p \ x \stackrel{*}{\Rightarrow} y \} = X \cap (p^{-1} y),$$

то есть, множество всех решений $x \in X$ уравнения $p \ x = y$. Будем называть проблему построения представления для множества $X_{p^{-1}y}$ проблемой *инверсного вычисления программы*. Рассмотрим следующие частные случаи данной проблемы.

Выполнение инверсного вычисления программы.

Если $X=D$, то $X_{p^{-1}y}=p^{-1} y$ —результат вычисления функции обратной, к функции программы p .

Построение множества истинности программы-предиката.

Пусть $X=D$, $y=\text{true}$, где true —константа, принятая в качестве значения “истина” при программировании предиката p . В этом случае, $X_{p^{-1}y}=\{ x \mid p \ x = \text{true} \}$ —множество истинности предиката p .

Обработка запроса к базе знаний.

Пусть p —двухместный предикат на множестве $\{ [x,y] \mid x,y \in \text{Eval} \}$:

$$\forall x,y \in \text{Eval}: p \ [x,y] \stackrel{*}{\Rightarrow} \{ \text{true}, \text{false} \}.$$

Рассмотрим некоторое значение x_0 . Пусть $X=\{ [x_0,y] \mid y \in D \}$. В этом случае, нахождение $X_{p^{-1}\text{true}}$ равносильно нахождению множества Y_0 ответов на запрос “*при каких y выполнено $p \ [x_0,y]$* ”: $Y_0 = \{ y \mid p \ [x_0,y] = \text{true} \}$, $X_{p^{-1}\text{true}}=[x_0,Y_0]$. Рассмотренный случай легко распространяется на более сложные запросы (например, “*при каких y выполнено $p \ [y,y]$* ”) и на большую аргументность предиката p . Общий случай:

$$X = \{ \text{req}[x_0,y] \mid y \in D \}, X_{p^{-1}\text{true}} = \text{req}[x_0,Y_0], \\ Y_0 = \{ y \mid p \ \text{req}[x_0,y] = \text{true} \},$$

где req —запрос, некоторое выражение над x_0 и y , x_0 —известные характеристики объекта, y —искомые характеристики.

Решение уравнений.

Рассмотрим уравнение $f \ [a,x] = b$, где a, b —параметры, x —неизвестные, f —функция, заданная программой¹. Фиксируем некоторые значения параметров a и b . Пусть

¹ Например, a —матрица размерности $m \times n$, b —вектор размерности n , x —вектор неизвестной размерности m , f —программа перемножения матрицы на вектор.

$X = \{ [a, x] \mid x \in D \}$. В этом случае построение множества $X_{f^{-1}b}$ равносильно построению множества $X_{a,b}$ решений уравнения при заданных значениях параметров a и b : $X_{a,b} = \{ x \mid f[a, x] = b \}$, $X_{f^{-1}b} = [a, X_{a,b}]$. Таким образом, если будут разработаны *эффективные* способы решения общей проблемы инверсного вычисления программ, а именно, если эти способы окажутся эффективными для построения представления множества $X_{f^{-1}b}$, то не будет причин заниматься разработкой специальных алгоритмов решения уравнений данного вида.

Общая постановка и приведенные выше частные случаи проблемы показывают актуальность задачи для теории и практики программирования. Дополнительная мотивация интереса к проблеме инверсии программ приведена в работе [20].

В разделе 6.3 будет предложен алгоритм построения представления множества $X_{p^{-1}y}$. А перед этим будет рассмотрен вспомогательный алгоритм приведения функции программы к табличной форме.

6.2 Приведение функции программы к табличной форме

В данном разделе будет предложен и обоснован алгоритм **tab**, со следующими свойствами (доказательство теоремы 47 приведено ниже).

Теорема 47

Пусть p —произвольная программа на языке TSG, x —класс, s —данное для p , i —целое число, большее индекса любой s -переменной из класса x , X —множество, представленное классом x : $\langle x \rangle = X$. Тогда алгоритм **tab** по заданным p , x , i строит перечисление—то есть список $L = [(x_1, s_{ex_1}), (x_2, s_{ex_2}), \dots]$, возможно бесконечной длины:

$$\text{tab } p \ x \ i \xRightarrow{*} L$$

где x_i —класс, s_{ex_i} — s -выражение над переменными из класса x_i . При этом выполнено следующее:

1. Классы x_i —попарно непересекающиеся подклассы класса x .
2. Для любого i и любого $d \in \langle x_i \rangle$ —программа p определена на d .
3. Для любого данного $d \in \langle x \rangle$ на котором программа p определена— $p \ d \xRightarrow{*} \text{res}$ —существует i такой, что $d \in \langle x_i \rangle$, и если подстановка s такая, что $x_i / .s = (d, \text{RESTR}[])$, то выполнено $\text{res} = c_{ex_i} / .s$. Кроме того, в данном случае, префикс длины i списка L будет построен алгоритмом **tab** за конечное число шагов.

Таким образом, алгоритм **tab** строит:

1. Представление множества $X_{\text{dom}(p)}$:

$$X_{\text{dom}(p)} \stackrel{\text{def}}{=} \{ d \mid d \in X, p \text{ определена на } d \}, X_{\text{dom}(p)} = \langle x_1 + x_2 + \dots \rangle, \text{ где } x_i \preceq x, \text{ и } \langle x_i \rangle \cap \langle x_j \rangle = \emptyset, \text{ если } i \neq j.$$

2. Табличную форму p на X —для любого $d \in X$ выполнено:

$$p \ d = \begin{cases} cex_i/.s & \text{если } d \in \langle x_i \rangle, s \text{—подстановка такая,} \\ & \text{что } x_i/.s = (d, \text{RESTR}[]) \\ \text{неопределено} & \text{если не существует } i \text{ такого, что } d \in \langle x_i \rangle \end{cases}$$

Замечание. В некоторых случаях алгоритм **tab** позволяет построить конечное табличное представление для функции **p** на множестве **X**: список **L** может быть пустым или иметь конечную длину, причем во втором случае он будет построен за конечное время².■

Рассмотрим **tree = xptr p x i**—дерево процессов программы **p** на данных класса **x**. Для каждого **d** ∈ **<x>** процесс **process(p, d)** представлен некоторым путем **w** из корневой вершины (см. главу 3). Так же, как и в алгоритме **nan**, припишем каждому узлу дерева некоторый подкласс класса **x**, а именно: корневой вершине припишем класс **x**; если некоторой вершине **n** приписан подкласс **x'** класса **x** и из вершины **n** выходит ребро с сужением **cnt**, ведущее к вершине **n'**, то вершине **n'** припишем подкласс **x'/.cnt** класса **x**. Как уже было показано в обосновании алгоритмов **ptr** и **nan** (см. разделы 3.2.2, 4.3.2), имеет место следующее утверждение 48.

Утверждение 48

Пусть **n**—некоторая вершина на *i*-том уровне дерева **tree**, **x'**—класс, приписанный вершине **n**, **c=((t, ce), r)**—конфигурация, приписанная вершине **n**, **w**—путь от корневой вершины до вершины **n**, **cnt₁, ..., cnt_i**—сужения, приписанные ребрам в пути **w**.

Тогда:

1. **x' = x/.cnt₁/.../.cnt_i = (ces, r)**;
2. **c** содержит **c**-переменные только из **cvar x'**;
3. **x'** представляет множество таких (и только таких) данных **d** ∈ **<x>**, что процесс **process_i(p, d)** соответствует пути **w** в дереве **tree**;
4. если **d** ∈ **<x'>**, **s** подстановка, такая, что **x'/.s = (d, RESTR[])**, то, для *i*-того состояния **s_i** вычисления **p d** выполнено: **s_i = (t, e)**, **e = ce/.s**.

Рассмотрим данные **d**, для которых программа определена. Процесс **process(p, d)** имеет конечную длину и в последнем состоянии процесса вычисляемый терм является пассивным. Такие процессы представлены путями дерева **tree**, ведущими к вершинам с пассивными конфигурациями, то есть к вершинам-листьям. Для вершин-листьев выполнено утверждение 49, которое является простым следствием утверждения 48 (подробное доказательство утверждения 49 приведено в [FTP]).

Утверждение 49

Пусть **{l₁, l₂, ...}** всех вершин-листьев дерева **tree**. Каждой вершине **l_i** приписаны: **c_i = ((exp_i, ce_i), r_i)**—пассивная конфигурация и **x_i = (xces_i, r_i)**—класс. Обозначим через **cex_i** **c**-выражение **exp_i/.ce_i**. Тогда, выполнено следующее:

1. **x_i ≤ x** и **x_i ∩ x_j = ∅**, если **i ≠ j**.

² Однако распознавание возможности построения такого конечного табличного представления для функции **p** на множестве **X** в общем случае является алгоритмически неразрешимой проблемой.

```

type TLevel = [(Class,Tree)]
type Tab    = [(Class, CExp)]

tab :: ProgR -> Class -> Tab
tab p x = tab' [ (x,tree) ] [ ]
        where tree = xptr p x

tab' :: TLevel -> TLevel -> Tab
tab' ((xi, LEAF c):lv1) lv2 = (xi,cex):(tab' lv1 lv2)
        where ((exp,ce),_) = c
              cex = exp/.ce

tab' ((xi, NODE _ bs):lv1) lv2 = tab' lv1 lv2'
        where lv2' = tabB xi bs lv2

tab' [ ] [ ] = [ ]

tab' [ ] lv2 = tab' lv2 [ ]

tabB :: Class -> [Branch] -> TLevel -> TLevel
tabB xi ((cnt,tree):bs) lv = tabB xi bs ((xi/.cnt,tree):lv)
tabB xi [ ] lv = lv

```

Рис. 6.1: Алгоритм построения табличной формы функции

2. Для любого x_i и любого $d \in \langle x_i \rangle$, программа p определена на d .
3. Для любого данного $d \in \langle x \rangle$, на котором программа p определена— $p \ d \xRightarrow{*} \text{res}$ —существует x_i такой, что $d \in \langle x_i \rangle$, и если подстановка s такая, что $x_i/.s = (d, \text{RESTR}[])$, то выполнено $\text{res} = \text{cex}_i/.s$.

Утверждение 49 позволяет построить алгоритм **tab** как композицию двух процессов.

1. Процесс приписывания вершинам дерева *tree* соответствующих подклассов класса x : корневой вершине должен быть приписан класс x , если вершине n приписан класс x' и из вершины n к вершине n' ведет ребро с сужением cnt , то вершине n' должен быть приписан класс $x'/.cnt$.

2. Процесс просмотра всех терминальных вершин дерева и перечисления всех пар $(x_i, \text{exp}/.ce)$, где x_i —класс, приписанный терминальной вершине, $((\text{exp}, ce), _)$ —конфигурация, приписанная этой же вершине.

На рисунке 6.1 приведен текст алгоритма **tab**, в котором реализованы оба описанных выше процесса в рамках одного просмотра *в ширину* дерева *tree*. Обход дерева в ширину обеспечивает следующее свойство: любая терминальная вершина будет просмотрена за конечное время. Из данного свойства и из утверждения 49 непосредственно следует справедливость теоремы 47.

6.3 Алгоритм инверсного вычисления программ

В данном разделе будет предложен и обоснован алгоритм **ura**, со следующими свойствами (доказательство теоремы 50 приведено ниже).

Теорема 50

Пусть p —произвольная программа на языке TSG, x —класс, s —данное для p , i —целое число, большее индекса любой s -переменной из класса x , X —множество, представленное классом x : $\langle x \rangle = X$, y — e -значение, $y \in \text{Eval}$.

Тогда алгоритм **ura'** по заданным p , x , i строит перечисление—то есть список $L' = [x'_1, x'_2, \dots]$, возможно бесконечной длины:

$$\text{ura}' \ p \ x \ y \ i \xRightarrow{*} L'$$

При этом выполнено следующее:

1. Классы x_i —попарно непересекающиеся подклассы класса x .
2. Для любого i и любого данного $d \in \langle x'_i \rangle$ —программа p определена на d и $p \ d = y$;
3. Для любого данного $d \in \langle x \rangle$, на котором программа p определена и $p \ d = y$, существует номер i такой, что $d \in \langle x'_i \rangle$. Кроме того, в данном случае, префикс длины i списка будет построен алгоритмом **ura'** за конечное число шагов.

Таким образом, **ura'** строит представление L' множества $X_{p^{-1}y}$: $X_{p^{-1}y} = \langle L' \rangle$, и тем самым решает проблему инверсного вычисления программы. Будем называть **ura'** алгоритмом инверсного вычисления программ, или *универсальным решающим алгоритмом (УРА)*.

6.3.1 Построение алгоритма инверсного вычисления программ

Рассмотрим табличное представление функции p на множестве $\langle x \rangle$:

$$L = \text{tab } p \ x \ i = [(x_1, \text{ces}_1), (x_2, \text{ces}_2), \dots].$$

Для любого $d \in X$ выполнено:

$$p \ d = \begin{cases} \text{ces}_i / .s & \text{если } d \in \langle x_i \rangle, s \text{—подстановка такая,} \\ & \text{что } x_i / .s = (d, \text{RESTR}[]) \\ \text{неопределено} & \text{если не существует } i \text{ такого, что } d \in \langle x_i \rangle \end{cases}$$

Имеет место следующее утверждение.

Утверждение 51

Обозначим $X'_i = \{ \text{ces}_i / .s \mid s \in \text{FVS}(x_i), \text{ces}_i / .s = y \}$

Тогда $X_{p^{-1}y} = \bigcup_i X'_i$.

Доказательство

$$\begin{aligned}
X_{p^{-1}y} &= \{ d \mid d \in X, p \text{ определена на } d, p d = y \} \\
&= \{ d \mid d \in \bigcup_i \langle x_i \rangle, p d = y \} \\
&= \bigcup_i \{ d \mid d \in \langle x_i \rangle, p_i d = y \} \\
&= \bigcup_i \{ ces_i/.s \mid s \in FVS(x_i), cex_i/.s = y \} = \bigcup_i X'_i \blacksquare
\end{aligned}$$

Множество X'_i представимо в виде подкласса x'_i класса x_i . А именно, рассмотрим $ures = \text{unify } [cex_i] [y]$, тогда справедливо следующее утверждение.

Утверждение 52

1. Если $ures = (\text{False}, [])$, то $X'_i = \emptyset = \langle x'_i \rangle$, где $x'_i = x_i/.emptC$.
2. Если $ures = (\text{True}, s')$, то $X'_i = \langle x'_i \rangle$, где $x'_i = x_i/.s'$.

Доказательство

1. Пусть $ures = (\text{False}, [])$. Тогда, в силу утверждения 6, не существует подстановки s такой, что $cex_i/.s = y$. То есть, $X'_i = \emptyset = \langle x'_i \rangle$, где $x'_i = x_i/.emptC$.

2. Пусть $ures = (\text{True}, s')$. Тогда, в силу утверждения 6, s' — подстановка такая, что $cex/.s' = y$ и $\text{dom } \text{subs} = \text{cvars } cex$. Рассмотрим подкласс $x'_i = x_i/.s'$. Докажем, что $X'_i = \langle x'_i \rangle$.

2.a. Пусть $d \in X'_i$. Тогда, по определению X'_i , существует подстановка s такая, что $s \in FVS(x_i)$, $d = ces_i/.s$, $r_i/.s = \text{RESTR}[]$, $cex_i/.s = y$. Рассмотрим подстановку $sa = s'*.s$. Заметим, что выполнено:

$$cex/.sa = cex/.(s'*.s) = (cex/.s')/.s = y/.s = y = cex/.s$$

Предпоследнее равенство выполнено в силу того, что y не содержит s -переменных. Из равенства $cex/.sa = cex/.s$ следует (утверждения 5), что $v/.sa = v/.s$ для любой s -переменной $v \in (\text{cvars } cex)$. Рассмотрим любую s -переменную $v \notin (\text{cvars } cex)$. Так как (утверждение 6) $(\text{cvars } cex) = (\text{dom } s')$, то получим:

$$v/.sa = v/.(s'*.s) = (v/.s')/.s = v/.s$$

Итак, для любой s -переменной v выполнено: $v/.sa = v/.s$. Следовательно

$$\begin{aligned}
(d, \text{RESTR}[]) &= (ces_i, r_i)/.s = x_i/.s = x_i/.sa = x_i/.(s'*.s) \\
&= x_i/.s'/.s = x'_i/.s, \text{ то есть } d \in \langle x'_i \rangle.
\end{aligned}$$

Доказано, что если $d \in X'_i$, то $d \in \langle x'_i \rangle$.

2.b. Пусть $d \in \langle x'_i \rangle$.

Существует $sa \in FVS(x'_i)$ такая, что $x'_i/.sa = (d, \text{RESTR}[])$. Рассмотрим $s = s'*.sa$. Тогда:

$$\begin{aligned}
x_i/.s &= x_i/.(s'*.sa) = x_i/.s'/.sa = x'_i/.sa = (d, \text{RESTR}[]), \\
cex/.s &= cex/.(s'*.sa) = cex/.s'/.sa = y/.sa = y, \\
\text{то есть } d &= ces_i/.s, r_i/.s = \text{RESTR}[], s \in FVS(x_i), d \in X'_i.
\end{aligned}$$

Доказано, что если $d \in \langle x'_i \rangle$ то $d \in X'_i$. Рассмотренные случаи (2.a и 2.b) завершают доказательство. \blacksquare

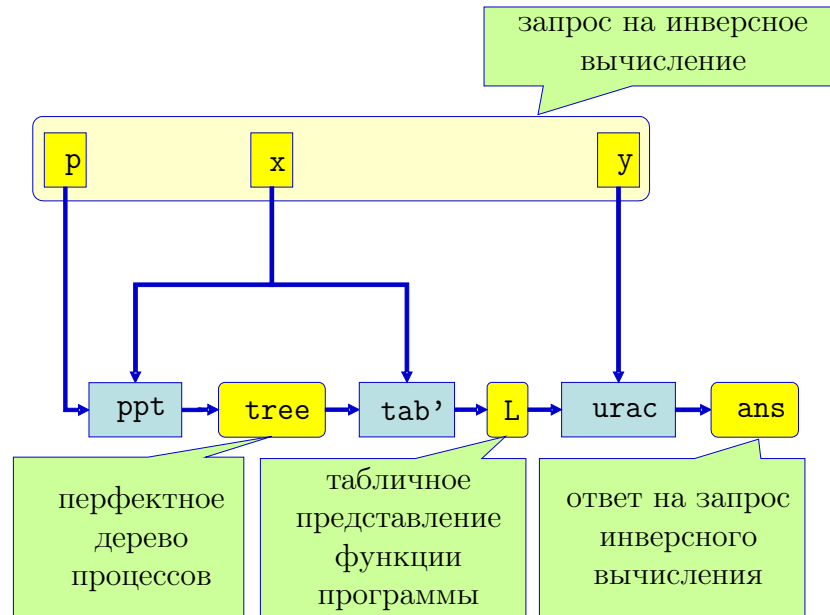


Рис. 6.2: Структура алгоритма инверсного вычисления программ

```

ura' :: ProgR -> Class -> EVal -> Classes
ura' p x y = urac (tab p x) y

urac :: Tab -> EVal -> Classes
urac ((xi, cex):ptab') y =
  case (unify [cex] [y]) of
    (False, _) -> tail
    (True, s) -> case xi' of
      (_, INCONSISTENT) -> tail
      - -> xi':tail
      where xi' = xi/.s
  where tail = urac ptab' y

urac [ ] y = [ ]

```

Рис. 6.3: Алгоритм инверсного вычисления программ

Утверждения 51 и 52 являются основаниями для следующей организации алгоритма `ura'` (текст алгоритма приведен на рисунке 6.3). Пусть `ptab`—табличное представление функции `p` на множестве `<x>` (см. функцию `ura'`):

```

ptab = tab p x i
ptab = [(x1, cex1), (x2, cex2), ...]

```

Просматривая список `ptab`, по каждому элементу (x_i, cex_i) из этого списка выполняют построение (ср. первое предложение `urac` и утверждение 52) класса x'_i , представляющего соответствующее множество X'_i . Если класс x'_i не пуст, его помещают в

```

ura :: ProgR -> Class -> EVal -> [(Subst, Restr)]
ura p x y = map altRepr (ura' p x y)
  where
    altRepr :: Class -> (Subst, Restr)
    altRepr xi = subClassCntr x xi

```

Рис. 6.4: Альтернативное представление результатов инверсного вычисления

строимое перечисление L' . Когда выполнен просмотр всего списка \mathbf{ptab} , то (см. второе предложение функции `urac`) алгоритм завершает перечисление L' классов x'_i . Теорема 47, утверждения 51 и 52 и рассмотренное построение алгоритма `ura` обеспечивают справедливость теоремы 50.

Замечание. В некоторых случаях `ura'` позволяет построить конечное представление множества X_{p-1y} (см. замечание и сноску 2 на стр. 93).■

6.3.2 Альтернативное представление результатов инверсного вычисления

Рассмотрим инверсное вычисление программы p :

`ura' p x y i.`

В с-данном x список с-выражений \mathbf{ces} представляет неполностью заданные аргументы для p : некоторые фрагменты аргументов определены (представлены конструкторами и атомами), а некоторые оставлены неизвестными (представлены с-переменными). Обозначив через \mathbf{a} —список определенных фрагментов в x , \mathcal{X} —список с-переменных, можно записать, что $x = \mathbf{req}[\mathbf{a}, \mathcal{X}]$, где $\mathbf{req}[_, _]$ —синтаксическая конструкция над \mathbf{a} и \mathcal{X} .

То есть, заказ на инверсное вычисление можно рассматривать как уравнение над неизвестными \mathcal{X} : $p \ \mathbf{req}(\mathbf{a}, \mathcal{X}) \xRightarrow{*} y$ и, в этом случае, хотелось бы представлять результат инверсного вычисления в виде перечисления тех значений (то есть, в виде подстановок вида $\mathcal{X} \rightarrow \mathbf{value}$), которые являются решениями этого уравнения. Более точно, речь идет о поиске сужений, то есть пар: подстановка $(\mathcal{X} \rightarrow \mathbf{value})$ и ограничения на \mathcal{X} ,—которые являются решениями в уравнения $p \ \mathbf{req}(\mathbf{a}, \mathcal{X}) \xRightarrow{*} y$.

Алгоритм `ura'`, как он изложен в разделе 6.3.1, перечисляет подклассы x'_i класса x , на которых выполнено $p \ \langle x'_i \rangle = \{y\}$. Однако, не сложно вместо подклассов x_i перечислять сужения, переводящие $x = (\mathbf{ces}, \mathbf{r})$ в $x_i = (\mathbf{ces}_i, \mathbf{r}_i)$. Так как $x_i \preceq x$, то существуют (см. утверждение 22) подстановка \mathbf{s}_i и рестрикция \mathbf{r}'_i такие, что:

$$x_i = x /. (\mathbf{S} \ \mathbf{s}_i) /. (\mathbf{R} \ \mathbf{r}'_i), \quad \mathbf{ces} /. \mathbf{s}_i = \mathbf{ces}_i, \quad \mathbf{r}_i = (\mathbf{r} /. \mathbf{s}_i) + \mathbf{r}'_i$$

Откуда получаем способ вычисления \mathbf{s}_i и \mathbf{r}'_i :

$$\begin{aligned} (\mathbf{True}, \mathbf{s}) &= \mathbf{unify} \ \mathbf{ces} \ \mathbf{ces}_i \\ \mathbf{r}'_i &= [\ \mathbf{ineq} \mid \mathbf{ineq} \in \mathbf{r}_i, \ \mathbf{ineq} \notin (\mathbf{r} /. \mathbf{s}_i) \] \end{aligned}$$

Указанная подстановка \mathbf{s}_i и рестрикция \mathbf{r}'_i являются описанием значений неизвестных фрагментов \mathcal{X} в обобщенном данном x , при которых $p \ \mathbf{req}[\mathbf{a}, \mathcal{X}] \xRightarrow{*} y$.

На рисунке 6.4 приведен модифицированный алгоритм `ura` инверсного вычисления, который, в полном соответствии с вышесказанным, в качестве результата инверсного вычисления перечисляет список всех пар (s_i, r'_i) , таких, что $x'_i = x / (S \ s_i) / (R \ r'_i)$.

6.3.3 Пример инверсного вычисления программ

В данном разделе приведен пример инверсного вычисления программы `prog` проверки вхождения подстроки в строку (см. рисунок 1.2 на стр. 15)³.

Пример 5

Введем следующие обозначения:

```
str = CONS 'A (CONS 'B (CONS 'C 'NIL)),
x = req[str,  $\mathcal{E}.1$ ] = ( [ $\mathcal{E}.1$ , str], RESTR[] ),
true = 'SUCCESS, i = 10.
```

Данное `str` является представлением (см. замечание на стр. 4.3.3) строки ('A 'B 'C). Инверсное вычисление `ura' prog x true i` должно построить представление множества $\langle x \rangle_{prog^{-1}true}$, то есть представление множества всех данных `d` вида: `d = [$\mathcal{E}.1$, str]`, где $\mathcal{E}.1$ —произвольное е-значение, на которых программа `prog` определена и результатом вычисления `prog d` является `true`. Задача заключается в нахождении представления для множества всех подстрок строки `str`, где определение предиката “*строка $\mathcal{E}.1$ является подстрокой строки `str`*” задано программой `prog`.

Вычисление `ura' prog x true i` приводит к следующему:

```
<x>_{prog^{-1}true} = ura' prog x true i =
<[ ([A.12, str], RESTR[]),
  ([CONS 'A A.18, str], RESTR[]),
  ([CONS 'A (CONS 'B A.24), str], RESTR[]),
  ([CONS 'B A.18, str], RESTR[]),
  ([CONS 'A (CONS 'B (CONS 'C A.30)), str], RESTR[]),
  ([CONS 'B (CONS 'C A.24), str], RESTR[]),
  ([CONS 'C A.18, str], RESTR[]) ]>
```

Альтернативное представление результатов инверсного вычисления данного запроса очень наглядно отражает найденные значения неизвестной $\mathcal{E}.1$:

```
ura prog x true i =
[ ([ $\mathcal{E}.1$  :-> A.12], []),
  ([ $\mathcal{E}.1$  :-> CONS 'A A.18], []),
  ([ $\mathcal{E}.1$  :-> CONS 'A (CONS 'B A.24)], []),
  ([ $\mathcal{E}.1$  :-> CONS 'B A.18], []),
  ([ $\mathcal{E}.1$  :-> CONS 'A (CONS 'B (CONS 'C A.30))], []),
  ([ $\mathcal{E}.1$  :-> CONS 'B (CONS 'C A.24)], []),
  ([ $\mathcal{E}.1$  :-> CONS 'C A.18], []) ]
```

³ Рассматриваемый в данном разделе пример является результатом реального выполнения программ в системе `Gofer`.

Итак, в результате данного вычисления получено конечное представление множества $\langle x \rangle_{prog^{-1}true}$ — в виде объединения семи подклассов класса x . Результаты описывают семь множеств представлений (см. замечание о возможности использования произвольного атома в качестве признака конца строки) всех семи подстрок строки **str**:

$()$, $('A)$, $('A 'B)$, $('B)$, $('A 'B 'C)$, $('B 'C)$, $('C)$

В [FTP] приведены другие примеры инверсного вычисления в TSG.

6.4 Развитие универсального решающего алгоритма

В данном разделе будут рассмотрены различные расширения идей, реализованных в алгоритме УРА. Структура изложения в данном разделе:

- В разделе 6.4.1 описан алгоритм **mg** вычисления наиболее общего унификатора — *most general unifier*, — двух SR-баз. На базе алгоритма **mg** в разделе 6.4.2 разработан алгоритм пересечения двух классов. По сути эти разделы расширяют средства и методы представления множеств (глава 2).
- Раздел 6.4.3 посвящен введению *симметричной* версии универсального решающего алгоритма **sura**.
- В разделе 6.4.4 показано, как можно для случая неплюских языков улучшить скорость работы и свойства терминируемости симметричного универсального решающего алгоритма.

6.4.1 MGU: наиболее общей унификатор

Определение 39

Пусть cs_1 и cs_2 — произвольные SR-базы. Подстановку s будем называть *унификатором* для данных cs_1 и cs_2 , если выполнено $cs_1/.s == cs_2/.s$.

Тем самым, унификатор s для заданных cs_1 и cs_2 является решением уравнения (клэша) $cs_1 := cs_2$. Можно с равным успехом рассматривать унификаторы для пар SR-баз (для списка пар SR-баз) или для клэшей (систем клэш).

Обозначим через $unf(cs_1, cs_2)$ множество всех унификаторов cs_1 и cs_2 .

Утверждение 53

Пусть cs_1 и cs_2 — произвольные SR-базы. Обозначим множество подстановок через **Subst**. Тогда:

$$\forall s \in unf(cs_1, cs_2), \forall s' \in \text{Subst} : (s.*s') \in unf(cs_1, cs_2) .$$

Утверждение 53 следуют из определения 39.

Далее мы будем использовать хорошо известные⁴ утверждение 54 и алгоритм **mg** (Рис. 6.5) вычисления *наиболее общего унификатора*.

⁴Описание и список литературы доступны по адресу <http://en.wikipedia.org/wiki/Unification>.

```

mgu :: Clashes -> Maybe Subst
mgu []          = Just []
mgu (eq:eqs) =
  case eq of
    cx1 :=: cx2 | cx1==cx2          -> mgu eqs
    v@(CVE _) :=: cx2                -> mgu' [v:->cx2]
                                     'when' (v 'notOccursIn' cx2)
    cx1:=:v@(CVE _)                 -> mgu' [v:->cx1]
                                     'when' (v 'notOccursIn' cx1)
    (CONS cy1 cy2) :=: (CONS cx1 cx2) -> mgu [cy1:=:cx1, cy2:=:cx2,]++eqs
    (CONS _ _) :=: _                 -> Nothing
    _ :=: (CONS _ _)                 -> Nothing
    (ATOM_) :=: (ATOM_)              -> Nothing
    v@(CVA _) :=: cx2                -> mgu' [v:->cx2]
    cx1:=:v@(CVA _)                 -> mgu' [v:->cx1]
  where
    mgu' s          = fmap (s.*) (mgu (eqs/.s))
    x 'when' p      = if p then x else Nothing
    v 'notOccursIn' ce = v 'notElem' (cvars ce)

```

Рис. 6.5: Алгоритм `mgu` вычисления наиболее общего унификатора для заданной системы клэшей

Утверждение 54

Пусть cs_1 и cs_2 — произвольные SR-базы и пусть множество $\text{unf}(cs_1, cs_2)$ не пусто. Тогда:

- $\exists s_{mgu}: \forall s \in \text{unf}(cs_1, cs_2), \exists s' \in \text{Subst}: s = (s_{mgu}.*s')$.
- Такой s_{mgu} называют *наиболее общим унификатором* для cs_1 и cs_2 .
- Наиболее общий унификатор s_{mgu} единственен, с точностью до переименования переменных в правых частях s_{mgu} и ограничения левых частей набором переменных из cs_1 и cs_2 .

Приведенный на рисунке 6.5 алгоритм для любой системы клэшей за конечное число шагов возвращает:

- либо `Nothing` — если система клэшей не имеет решения в множестве конечных с-выражений;
- либо `Just smgu`, где наиболее общий унификатор для данной системы клэшей.

Слова “система клэшей не имеет решения в множестве конечных с-выражений” поясним на следующем примере. Рассмотрим клэш $\mathcal{E}.1 = (\text{CONS } 'A \ \mathcal{E}.1)$. Очевидно, данный клэш не имеет решения в множестве конечных с-выражений. Это легко понять из

соображений “материального баланса”: пусть решение для $\mathcal{E}.1$ имеется и в нем n конструкторов **CONS**, тогда из клэша следует, что выполнено $n = n + 1$, что не может быть ни для какого конечного числа n .

С другой стороны этот клэш имеет решение в множестве бесконечных s -выражений:

$$[\mathcal{E}.1 \rightarrow (\text{CONS } 'A (\text{CONS } 'A (\text{CONS } 'A \dots)))].$$

Очевидна модификация алгоритма **mgu** для случая, когда допускаются бесконечные s -выражения: надо из алгоритма удалить проверки на повторное вхождение переменных: ‘when’ (... ‘notOccursIn’ ...).

6.4.2 Пересечение классов

Пусть $\mathcal{C}_1 = (\text{ces}_1, \text{rs}_1)$ и $\mathcal{C}_2 = (\text{ces}_2, \text{rs}_2)$ два произвольных класса и пусть в них не используются общие s -переменные⁵: $(\text{cvars } \mathcal{C}_1) \cap (\text{cvars } \mathcal{C}_2) = \emptyset$.

Преобразуем эквивалентным образом условие $d \in \langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_2 \rangle$:

$$\begin{aligned} d \in \langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_2 \rangle &\Leftrightarrow \\ &\Leftrightarrow \exists s_1: \text{ces}_1/.s_1 = d, \text{rs}_1/.s_1 = \text{RESTR[]} \\ &\quad \exists s_2: \text{ces}_2/.s_2 = d, \text{rs}_2/.s_2 = \text{RESTR[]} \\ &\quad \text{здесь } s = s_1 ++ s_2 \\ &\Leftrightarrow \exists s: \text{ces}_1/.s = d = \text{ces}_2/.s, \text{rs}_1/.s = \text{RESTR[]}, \text{rs}_2/.s = \text{RESTR[]} \\ &\quad (\text{rs}_1.\text{rs}_2)/.s = \text{RESTR[]} \\ &\Leftrightarrow \exists s: s \text{ in unf}(\text{ces}_1, \text{ces}_2) \ d = \text{ces}_1/.s = \text{ces}_2/.s, \\ &\quad (\text{rs}_1.\text{rs}_2)/.s = \text{RESTR[]} \\ &\quad \text{здесь } s = s_{mgu} .* s' \\ &\Leftrightarrow \exists s': d = \text{ces}_0/.s', \text{rs}_0/.s' = \text{RESTR[]} \\ &\Leftrightarrow d \in \langle \mathcal{C}_0 \rangle \end{aligned}$$

Таким образом: $d \in \langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_2 \rangle \Leftrightarrow d \in \langle \mathcal{C}_0 \rangle$, то есть: $\langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_2 \rangle = \langle \mathcal{C}_0 \rangle$.

где

- s_{mgu} — наиболее общий унификатор ces_1 и ces_2 ,

$$\text{Just } s_{mgu} = \text{mgu } (\text{zipWith } (:=) \text{ ces}_1 \text{ ces}_2);$$

- $\text{ces}_0 = \text{ces}_1/.s_{mgu} = \text{ces}_2/.s_{mgu}$;
- $\text{rs}_0 = (\text{rs}_1.\text{rs}_2)/.s_{mgu}$.

Тем самым, справедливо следующее утверждение:

Утверждение 55

Пусть $\mathcal{C}_1 = (\text{ces}_1, \text{rs}_1)$ и $\mathcal{C}_2 = (\text{ces}_2, \text{rs}_2)$ два произвольных класса и пусть в них не используются общие s -переменные: $(\text{cvars } \mathcal{C}_1) \cap (\text{cvars } \mathcal{C}_2) = \emptyset$. Тогда:

1. если не существует унификатора для ces_1 и ces_2 , то $\langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_2 \rangle = \emptyset$,
2. иначе $\langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_2 \rangle = \langle \mathcal{C}_0 \rangle$,

⁵Это требование всегда можно выполнить при помощи переименовки s -переменных.


```

( $\cdot$ ^ $\cdot$ ) :: Class -> Class -> [(Subst, Restr)]
(ces1, rs1)  $\cdot$ ^ $\cdot$  (ces', rs') =
  let (ces2, rs2) = rename (ces', rs') (ces1, rs1)
  in case mgu (zipWith (:=:) ces1 ces2) of
    Nothing          -> [ ]
    Just s -> case (rs1+.rs2)/.s of
      INCONSISTENT -> [ ]
      _            -> (s, rs2/.s)

```

Рис. 6.6: Алгоритм пересечения двух классов

где, во втором случае:

- s_{mgu} — наиболее общий унификатор ces_1 и ces_2 ,

$$\text{Just } s_{mgu} = \text{mgu } (\text{zipWith } (:=:) \text{ ces}_1 \text{ ces}_2);$$
- $ces_0 = ces_1/.s_{mgu} = ces_2/.s_{mgu}$;
- $rs_0 = (rs_1+.rs_2)/.s_{mgu} = (rs_1/.s_{mgu}) +. (rs_2/.s_{mgu})$;
- $\mathcal{C}_0 \preceq \mathcal{C}_1, \mathcal{C}_0 = \mathcal{C}_1/.(S \ s_{mgu})/.(R \ rs_2')$, где $rs_2' = rs_2/.s_{mgu}$;
- $\mathcal{C}_0 \preceq \mathcal{C}_2, \mathcal{C}_0 = \mathcal{C}_2/.(S \ s_{mgu})/.(R \ rs_1')$, где $rs_1' = rs_1/.s_{mgu}$.

Утверждение 55 является обоснованием корректности алгоритма (\cdot ^ \cdot) вычисления пересечения двух классов (см. Рис. 6.6), выраженной в следующем утверждении:

Утверждение 56

Пусть $\mathcal{C}_1 = (ces_1, rs_1)$ и $\mathcal{C}_2 = (ces_2, rs_2)$ два произвольных класса. Тогда за конечное число шагов вычисление операции (\cdot ^ \cdot) для \mathcal{C}_1 и \mathcal{C}_2 завершится со следующим результатом:

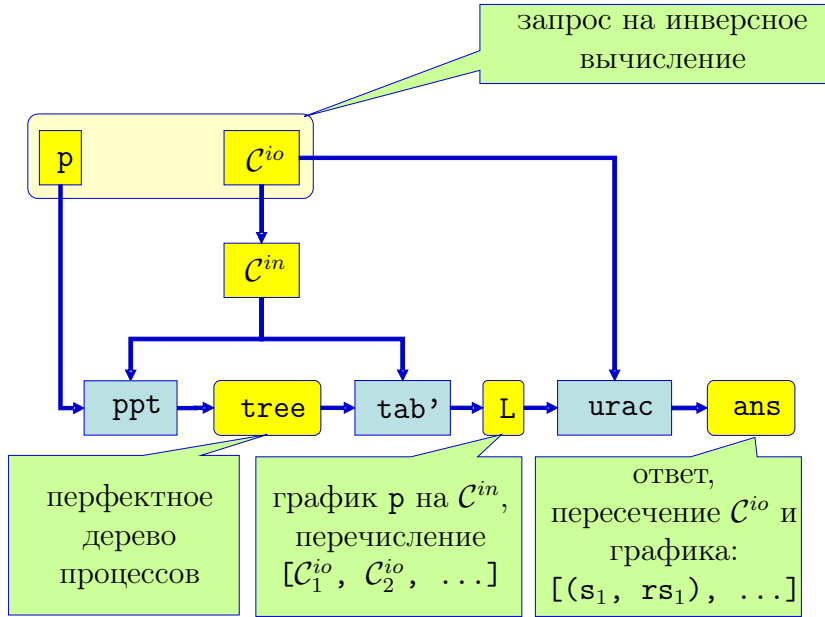
- Либо $[]$, если $\langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_1 \rangle = \emptyset$.
- Либо $[(s, rs)]$, если $\langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_1 \rangle \neq \emptyset$, при этом

$$\langle \mathcal{C}_1 \rangle \cap \langle \mathcal{C}_2 \rangle = \langle \mathcal{C}_0 \rangle, \text{ где } \mathcal{C}_0 = \mathcal{C}_1/.(S \ s)/.(R \ rs).$$

Упомянутая на рисунке 6.6 функция `rename`, не показана здесь, так как она достаточно тривиальна. Она в классе (ces', rs') выполняет переименование с-переменных, таким образом, чтобы получившийся после этого класс (ces_2, rs_2) :

$$(ces_2, rs_2) = \text{rename } (ces', rs') (ces_1, rs_1)$$

не имел бы с классом (ces_1, rs_1) общих переменных.

Рис. 6.7: Структура алгоритма *sura*

6.4.3 СУРА, симметричная версия универсального решающего алгоритма

Имеется некоторое несоответствие между вполне “симметричной” природой понятия *функция* и несимметричной постановкой задачи на инверсное вычисление.

Действительно, по определению, функция $f : A \rightarrow B$ это отношение $f \subseteq A \times B$, то есть множество пар $f = \{ \dots (a, b) \dots \}$, где $a \in A$, $b \in B$, обладающее следующим свойством $((x, y_1) \in f) \wedge (x, y_2) \in f \Rightarrow (y_1 = y_2)$. Обратная функция — а точнее, обратное отношение, — определяется тривиально и вполне симметричным образом: $f^{(-1)} = \{ (b, a) \mid (a, b) \in f \}$. При этом в постановке задачи на инверсное вычисление имеется явная асимметрия: для заданной программой функции f , заданного классом множества X и заданного “желаемого” значения y надо решить (относительно x) систему:

$$\begin{cases} f(x) = y \\ x \in X \end{cases}$$

то есть посчитать пересечение множества f (графика функции), вот с таким множеством пар “вход-выход” (io-пар): $X \times \{y\}$.

Естественное симметричное обобщение данной задачи такое: для заданной программой функции f , заданного io-классом множества $C_{io} = \{ \dots (x, y) \dots \}$ “желаемых” значений (x, y) io-пар надо решить систему:

$$\begin{cases} f(x) = y \\ (x, y) \in C_{io} \end{cases}$$

то есть посчитать пересечение множества f (графика функции), с множеством C_{io} пар “вход-выход” (io-пар).

Умение решать задачу в такой постановке позволит за счет задания различных множеств C_{io} проводить:

```

sura :: ProgR -> IOClass -> [(Subst, Restr)]
sura p cio@((cxs, cy), rs) = concat (map f (tab p cin))
  where cin = (cxs, rs)
        f ((cxs', rs'), cy') = (cy:cxs, rs).^.(cy':cxs', rs')

```

Рис. 6.8: Алгоритма *sura*

- “классическое” инверсное вычисление: $C_{io} = X \times \{y\}$;
- поиск неподвижной точки $f(x) = x$: $C_{io} = \{(x, x) \mid x \in X\}$;
- стандартное (прямое) вычисление: $C_{io} = \{x\} \times Y$.

Определение 40

Пусть *cexps*—список с-выражений, *cexp*—с-выражение, *rs*—рестрикция. Тогда назовем *io*-классом с-конструкцию: $\mathcal{C}^{io} = ((cexps, cexp), rs)$. Данные с-конструкции будут использоваться для определения множеств *io*-пар (вход-выход) TSG-программ. Для *io*-класса $\mathcal{C}^{io} = ((cexps, cexp), rs)$ соответствующий класс $\mathcal{C}^{in} = (cexps, rs)$ определяет множество входов $\langle \mathcal{C}^{in} \rangle$, которое является проекцией множества $\langle \mathcal{C}^{io} \rangle$

Для заданной TSG-программы *p* и заданного *io*-класса \mathcal{C}^{io} задача симметричного инверсного вычисления состоит в нахождении представления множества:

$$\{ (d, y) \mid (d, y) \in \langle \mathcal{C}^{io} \rangle, p \ d \stackrel{*}{\Rightarrow} y \}.$$

которое равно пересечению множества $\langle \mathcal{C}^{io} \rangle$ и графика функции программы *p* для входных данных из множества $\langle \mathcal{C}^{in} \rangle$:

$$\langle \mathcal{C}^{io} \rangle \cap \{ (d, y) \mid d \in \langle \mathcal{C}^{in} \rangle, p \ d \stackrel{*}{\Rightarrow} y \}.$$

Построение графика функции программы *p* для входных данных из множества $\langle \mathcal{C}^{in} \rangle$ в виде перечисления *io*-классов—задача, которая ранее была уже нами решена (алгоритм *tab*, раздел 6.2). Тем самым, нам остается просто вычислить пересечение каждого *io*-класса из данного перечисления с *io*-классом $\langle \mathcal{C}^{io} \rangle$.

Это позволяет нам сформулировать (см. рисунки 6.7 и 6.8) алгоритм *sura*, со следующими свойствами.

Теорема 57

Пусть *p*—произвольная программа на языке TSG, $\mathcal{C}^{io} = ((cexps, cexp), rs)$ —*io*-класс, $\mathcal{C}^{in} = (cexps, rs)$ —с-данное для *p*.

Тогда алгоритм *sura* по заданным *p* и \mathcal{C}^{io} строит перечисление—то есть список $L' = [(s_1, rs_1), (s_2, rs_2), \dots]$, возможно бесконечной длины:

$$sura \ p \ \mathcal{C}^{io} \stackrel{*}{\Rightarrow} L'$$

При этом выполнено следующее:

1. Классы $\mathcal{C}_i^{io} = \mathcal{C}^{io} / (S \ s_i) / (R \ rs_i)$ —попарно непересекающиеся подклассы класса \mathcal{C}^{io} .

2. Для любого i и любой пары $(d, y) \in \langle C_i^{io} \rangle$ — программа p определена на d и $p\ d = y$;
3. Для любой пары $(d, y) \in \langle C_i^{io} \rangle$ такой что, программа p определена на d и $p\ d = y$, существует номер i такой, что $(d, y) \in \langle C_i^{io} \rangle$. Кроме того, в данном случае, префикс длины i списка будет построен алгоритмом *sura* за конечное число шагов.

6.4.4 XURA: расширенная симметричная версия универсального решающего алгоритма для случая неплюских языков

Раздел пока не написан...

6.5 Различные подходы к инверсии программ

Первые эксперименты по созданию и исследованию свойств УРА для языка Рефал проведены В.Ф.Турчиным и С.А.Романенко в 1973 г.

В эти же годы вышла статья [1], которая, по сути, является первой работой, описывающей применение методов метавычислений для инверсии программ. В ней используется иной подход для инверсии программ, основанный на построении графа конфигураций (графа процессов вычисления) программы p на множестве X и на методах интерпретации “в обратном направлении” (прочтения справа налево) графа конфигураций. Речь идет буквально о *синтаксической (текстуальной) инверсии* некоторой программы — графа конфигурации, — построенной при помощи метавычислений над исходной программой. Данный подход позволяет в некоторых случаях строить очень эффективные инверсии программ. Однако, иногда такое построение осложнено тем, что не всякий граф конфигураций, будучи записанным в обратном направлении, является снова корректным графом конфигураций. Последующие исследования данного подхода можно найти в [14].

А.Ю.Романенко в работах [10, 16, 17] разрешает для некоторых случаев указанные сложности текстуальной инверсии программ: в этих работах предлагается диалект Рефала, в котором программы допускают корректное прямое и обратное вычисление и исследуются свойства данного языка.

Автор данной работы придерживался изложенного здесь подхода к проблеме инверсии программ. В 1978 г. он повторил эксперименты В.Ф.Турчина и С.А.Романенко, создав свою реализацию УРА для языка ограниченный Рефал. Однако, как и в более ранних экспериментах с УРА, система являлась скорее простой программой, позволяющей хорошо протестировать основные алгоритмы метавычислений для языка Рефал, нежели средством для исследования инверсии программ. В 1986 г. автор еще раз реализовал УРА, на этот раз для изучения свойств языка Инверсный Рефал, как языка логического программирования и сравнения его с другими языками логического программирования [15].

Помимо подходов, использующих метавычисления для решения проблемы инверсного вычисления программ, существуют работы по инверсии, не связанные с метавычислениями. Так, в работах [24, 25] описываются методы обращения функций программ, основанные на трансформационном подходе.

Таким образом, предложенный выше универсальный решающий алгоритм **ura** конечно *не является единственным* (или самым эффективным) способом инверсного вычисления программ. Пожалуй, он является самым прямым и безотказным (*теоретически* всегда применимым) способом, но на практике, в каждом конкретном случае он вполне *может* уступать по эффективности другим подходам.

Описанный выше УРА показывает, что инверсное вычисление программ—реализуемо. А проблема эффективной практической реализации инверсного вычисления до сих пор остается предметом для дальнейших исследований⁶.

⁶ Один из подходов к повышению эффективности инверсных вычислений программ, основанный на использовании специализации, будет рассмотрен в главе 8.

Глава 7

Инверсное программирование

7.1 Инверсная семантика языка реализации

При наличии эффективных средств инвертирования программ¹ для языка R , программист волен выбирать, как ему выполнять то или иное задание на программирование: возможно, в каких-то случаях проще запрограммировать не требуемую в задании функцию f , а такую функцию p , которая при инверсном вычислении совпадает с функцией f (реализует функцию f). Так, программирование функции перечисления всех подстрок заданной строки представляется задачей существенно более сложной, чем программирование предиката $p \text{ subs } str$, проверяющего, что строка $subs$ является подстрокой строки str . В разделе 6.1 приведен ряд задач, которые проще запрограммировать *инверсно*. Например, можно отказаться от программирования методов решения уравнения: $f[a, x] = b$ а вместо этого написать программу f вычисления значения левой части уравнения по заданным значениям a и x . А собственно решение уравнения возложить на систему инверсного вычисления этой программы. Вероятно, в большинстве случаев такая программа f будет значительно проще для написания, тестирования и верификации, чем программа, реализующая тот или иной алгоритм решения уравнения $f[a, x] = b$.

При использовании средств инверсного вычисления программист имеет дело с языком R , снабженным двумя семантиками—обычной (вычисления при помощи `int`) и инверсной (вычисления при помощи `ura`). Следовательно, в этом случае с каждым текстом программы p можно связать не одну, а две функции программы:

- $p :: D \rightarrow Eval$ —обычная функция программы (обычная семантика):

$$p \ d \stackrel{\text{def}}{=} \text{int } p \ d;$$

- $p^{inv} :: \mathcal{D} \rightarrow \mathcal{R}$ —инверсная функция (инверсная семантика) программы:

$$p^{inv} \ req \stackrel{\text{def}}{=} \text{ura } p \ req;$$

где \mathcal{D} —множество запросов на инверсное вычисление, $req \in \mathcal{D}$, \mathcal{R} —множество результатов инверсного вычисления—списков пар вида $(subst, [ineq])$.

¹ Для конкретности изложения, везде далее будем полагать, что это `ura'` или `ura`, хотя вместо них может быть использован любой иной (см. раздел 6.5) алгоритм инверсного вычисления программ.

Определение 41

Будем называть *инверсным программированием* такую деятельность, при которой программист вместо того чтобы программировать требуемую в задании функцию f , реализует программу p , которая при инверсном вычислении совпадает с функцией f .

При инверсном программировании, на этапе реализации программы p программист имеет дело с обычным языком R и обычными вычислениями в этом языке. Например, при отладке фрагментов p или всей программы p можно выполнять обычные вычисления на тестах. Только когда дело дойдет до реального вычисления функции $f=p^{inv}$, он может обратиться к УРА: задать известные фрагменты в параметрах p , описать неизвестные фрагменты, желаемый результат i , выполнив инверсное вычисление p , найти значения неизвестных фрагментов. При этом ему нет никакой необходимости знать, каким образом осуществляется инверсное вычисление, то есть, как УРА устроен “внутри”.

7.2 Инверсное и логическое программирование

В данном разделе будет рассматриваться один частный случай инверсного программирования, основанный на инверсном вычислении *программ-предикатов*.

Зададимся некоторой константой $true \in D$, которую будем использовать в качестве истинного логического значения при программировании на R программ-предикатов. Будем называть *инверсным языком* R и обозначать R^{inv} следующий язык программирования:

- *синтаксис программ* полностью совпадает с синтаксисом языка R ;
- *предметная область*: в качестве данных программ используются классы-запросы (обобщенные данные программ) $req[x, \mathcal{Y}] \in D$; в качестве результатов вычисления программ—перечисления значений s -переменных $[(\mathcal{Y}:->y_1, r_1), \dots] \in \mathcal{R}$ из классов-запросов; состояния вычислений и промежуточные данные—представлены конфигурациями и s -средами.
- *семантика (вычисление) программ* определяется при помощи ura :

$$p^{inv} req[x, \mathcal{Y}] \stackrel{\text{def}}{=} uraR p req[x, \mathcal{Y}] true i \stackrel{*}{\Rightarrow} [(\mathcal{Y}:->y_1, r_1), \dots] \in \mathcal{R}$$

$$p^{inv} :: D \rightarrow \mathcal{R}, req[x, \mathcal{Y}] \in D.$$

В данном разделе будет показано, что частный случай использования инверсного программирования, а именно—инверсное вычисление программ-предикатов—очень близок к логическому программированию, а язык R^{inv} можно рассматривать как язык логического программирования. Данный тезис обосновывается следующим образом:

- Рассматриваются (раздел 7.2.1) некоторые характерные для языка логического программирования Пролог свойства и обнаруживаются такие же свойства у языка R^{inv} .

- Рассматривается (раздел 7.2.2) конкретный язык R —ограниченный Рефал; реализованный автором для данного языка универсальный решающий алгоритм позволяет сравнить инверсный к ограниченному Рефалу язык R^{inv} с языком микро-Пролог, на примере одной конкретной программы.

7.2.1 Сравнение свойств инверсных и логических языков

При описании семантики Пролога часто опираются на понятие *декларативного смысла программы* [37, 36]. Так, в [36, стр. 43–45] сказано:

Декларативный смысл касается только отношений, описываемых программой. Таким образом, декларативный смысл определяет, что должно быть результатом работы программы. . .

- Программирование на Прологе состоит в определении отношений и в постановке вопросов, касающихся этих отношений.
- Вопросы напоминают запросы к некоторой базе данных. Ответ системы на вопрос представляет собой множество объектов, которые удовлетворяют запросу.

При программировании на Прологе предлагается описать отношение (некий предикат $p(X, Y)$). При исполнении, по запросу $p(x, Y)$, где x —константа (константы), Y —переменная (переменные), система перечисляет такие значения y , что имеет место отношение $p(x, y)$. Таким образом, результат вычисления запроса (декларативная семантика программы) определяется как результат решения уравнения относительно Y : $p(x, Y) = \text{true}$.

Декларативная семантика языка R^{inv} определена точно таким же образом—через понятие *множество решений* такого же уравнения.

Описание операционной (процедурной) семантики языка R^{inv} , то есть описание внутренних процедур УРА, также содержит черты, наблюдаемые в описаниях процедурных семантик языков логического программирования [37]: операции с обобщенными данными (данными, содержащими переменные), в том числе их сравнение (унификация); перебор вариантов, в том числе и для разрешения неоднозначности выполнения операций над обобщенными данными; обход дерева вариантов вычисления предиката над данными с переменными.

Если провести более подробное сравнение всех внутренних механизмов алгоритма *ura* и процедурной семантики Пролога, то можно прийти к следующему заключению—интерпретатор Пролога—частный случай УРА:

1. который рассчитан только на программы, все функции в которых—предикаты;
2. в котором не ведется учет рестрикций на с-переменные;
3. в котором обход дерева процессов выполняется “в глубину”.

Следствием того, что обход дерева процессов в Пролог-системах выполняется в глубину, являются следующие два обстоятельства.

1. Исчезают гарантии нахождения ответа—если ответ на запрос соответствует вершине на правой ветви дерева процессов, а левое поддерево—бесконечное, то ответ системой Пролог не будет найден—система навсегда займется обходом в глубину левого поддерева. Эта ситуация описывается так [36, стр. 83]:

... пролог-система может пытаться решить задачу таким способом, при котором решение никогда не будет достигнуто, хотя оно существует. Такая ситуация не является редкостью при программировании на Прологе. ... Что действительно необычно при сравнении Пролога с другими языками, так это то, что декларативная семантика пролог-программы может быть правильной, но в то же самое время ее процедурная семантика может быть ошибочной в том смысле, что с помощью такой программы нельзя получить правильный ответ на вопрос. В таких случаях система не способна достичь цели потому, что она пытается добраться до ответа, но выбирает при этом неверный путь.

Инверсные языки лишены такой действительно необычной особенности: в силу теоремы 50 у них процедурная семантика в точности совпадает с декларативной.

2. Пользователя Пролога приходится посвящать во внутренние процедуры системы Пролог, приходится объяснять приемы программирования, способствующие тому, чтобы ответы на запросы располагались на левых ветвях дерева перебора, снабжать программиста средствами управления системой Пролог—средствами изменения общего порядка обхода дерева, “насильного подталкивания” системы “на верный путь”.

Если оставить в стороне эти различия (порядок обхода дерева вариантов), то можно сказать, что для языков Пролог и R^{inv} декларативные и процедурные описания семантик очень близки.

Психологическая установка пользователя при работе с языком R^{inv} также аналогична той, что и при работе с языком логического программирования. Цель пользователя—описать, что его интересует (то есть описать предикат p), но не описывать, как построить по данному x такой \mathcal{Y} , что $p[x, \mathcal{Y}] = \text{true}$. Искомое \mathcal{Y} будет найдено алгоритмом `ura`.

При разработке предиката p пользователь может не осознавать существования двух языков R и R^{inv} . Он может создавать свою программу p , полностью находясь в среде понятий и средств (компиляторы, отладчики и т.п.) привычного ему языка R . И только после завершения разработки программы p он может обратиться к некой программе `ura`, которая

$$\text{ura } p \text{ req}[x, \mathcal{Y}] \stackrel{*}{\Rightarrow} [(\mathcal{Y} \rightarrow y_1, r_1), \dots]$$

неведомым для него образом по запросу находит все такие значения y_i неизвестных \mathcal{Y} , что $p \text{ req}[x, y_i] = \text{true}$.

7.2.2 Сравнение пролог-программы и программы на Инверсном Рефале

В этом разделе примем, что R —ограниченный Рефал [8], R^{inv} —инверсный к нему язык. Ниже приводятся реальные тексты программ, которые были написаны (и успеш-

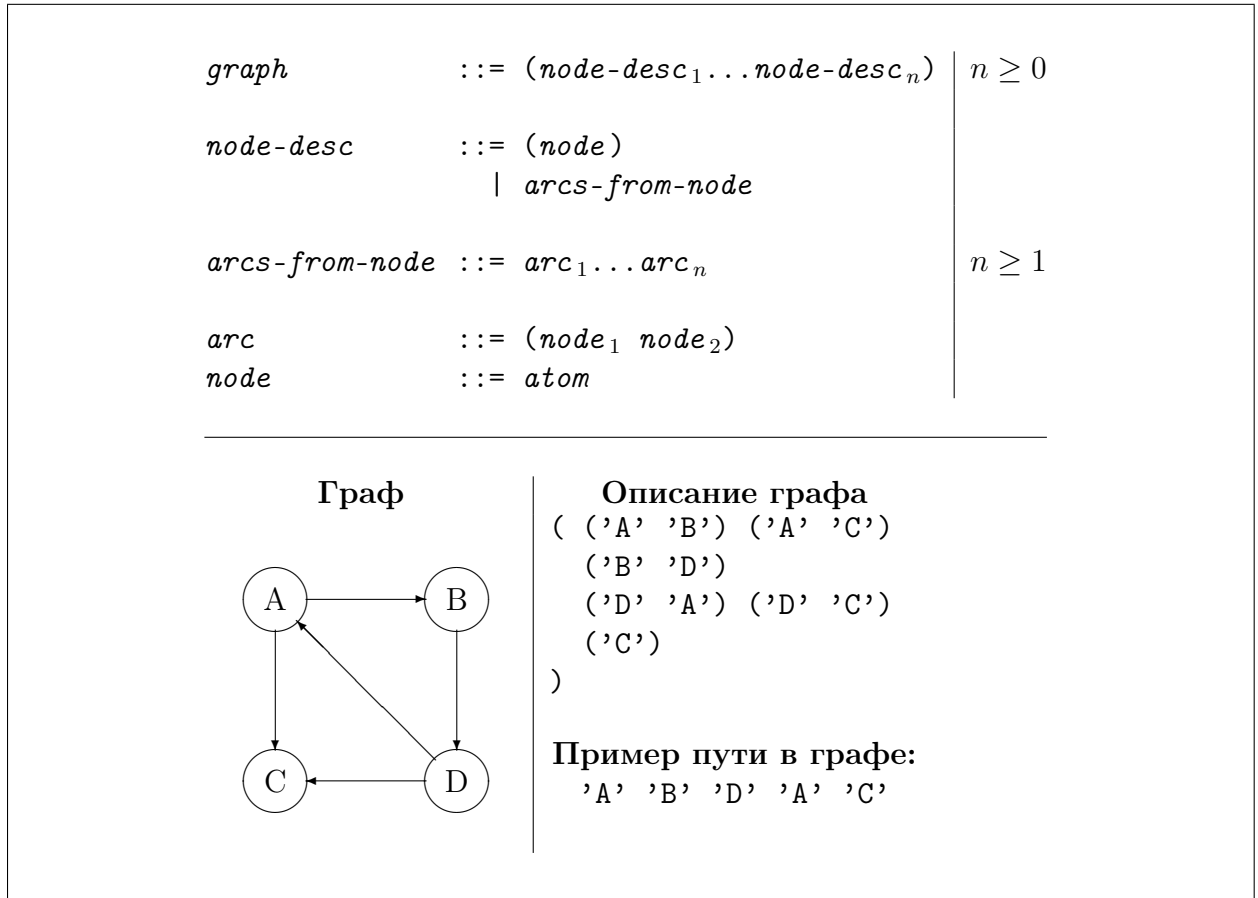


Рис. 7.1: Синтаксис представления графов, пример представления

но выполнялись) для системы микро-Пролог и для системы URA², реализующей Инверсный Рефал. На примере решения одной и той же задачи на языках Пролог и Инверсный Рефал будет завершено сравнение языков логического программирования и инверсного программирования предикатов. Тем самым, будет закончено обоснование тезиса из раздела 7.2: “частный случай использования инверсного программирования, а именно—инверсное вычисление программ-предикатов—очень близок к логическому программированию”.

Рассматривается задача нахождения пути в ориентированном графе. Синтаксис представления графов и пример представления приведены на рисунке 7.1. Для решения таких задач на языке R^{inv} нужно описать предикат³: $\langle \text{путь } e1 \text{ (eZ)} \rangle$, принимающий значение **true** тогда, и только тогда, когда $e1$ —путь в графе eZ . Будет удобно принять, что **true**—пустая строка. Так как конкатенация строк будет пустой тогда, и только тогда, когда все конкатенируемые строки пустые, то данный выбор значения **true** позволяет легко программировать любой предикат Q , равный конъюнкции предикатов $P1, P2, \dots, Pn$:

$$Q \text{ eX} = \langle P1 \text{ eX} \rangle \langle P2 \text{ eX} \rangle \dots \langle Pn \text{ eX} \rangle$$

Будем в данной задаче определять предикаты только на множестве их истинности. Это удобно, так как в данной задаче не возникает необходимости программировать связи

² Реализована автором данной работы в 1986 году. Подробнее—см. раздел 6.5.

³ В Рефале угловые скобки обозначают вызовы функций.

“не” и “или”. Ниже приводятся описания программ для решения нашей задачи на языке R^{inv} —инверсный ограниченный Рефал—и микро-Пролог.

Программа на языке Рефал	Программа на языке микро-Пролог
<p>путь $sX (eZ) = +$ $\langle \text{вершина } sX (eZ) \rangle$ $sX sY e1 (eZ) = +$ $\langle \text{ребро } sX sY (eZ) \rangle +$ $\langle \text{путь } sY e1 (eZ) \rangle$</p>	<p>$((\text{путь } (X) Z)$ $(\text{вершина } X Z))$ $((\text{путь } (X Y Y1) Z)$ $(\text{ребро } X Y Z)$ $(\text{путь } (Y Y1) Z))$</p>
<p>вершина $sX ((sX eY) eZ) =$ $sX ((e1) eZ) = +$ $\langle \text{вершина } sX (eZ) \rangle$</p>	<p>$((\text{вершина } X ((X Y) Z)))$ $((\text{вершина } X (Y1 Z))$ $(\text{вершина } X Z)))$</p>
<p>ребро $sX sY ((sX sY) eZ) =$ $sX sY ((e1) eZ) = +$ $\langle \text{ребро } sX sY (eZ) \rangle$</p>	<p>$((\text{ребро } X Y ((X Y) Z)))$ $((\text{ребро } X Y (Y1 Z))$ $(\text{ребро } X Y Z))$</p>
<p>граф1 $= (('AB') ('AC') +$ $('BD') +$ $('DA') ('DC') +$ $('C') +$ $)$</p>	<p>$((\text{граф1 } ((A B) (A C)$ $(B D)$ $(D A) (D C)$ (C) $)))$</p>
<p>путь-в-Г1 $eX = +$ $\langle \text{путь } eX \langle \text{граф1} \rangle \rangle$</p>	<p>$((\text{путь-в-Г1 } X)$ $(\text{граф1 } Z)$ $(\text{путь } X Z))$</p>

Трудно не заметить текстуальной близости программы на микро-Прологе и программы на Рефале. Однако, Рефал—не язык логического программирования, а данная программа на Рефале—обычная рефал-программа, реализующая предикат в предположении, что истинное булевское значение—пустое выражение. Программа написана в обычном рефальском стиле и предназначена для обычного вычисления в среде рефал-систем. Другое дело, что в системе URA данную программу можно вычислять инверсно, то есть рассматривать как программу на языке Инверсный Рефал. При таком использовании данного текста программы, рефал-программа *обретет* черты программы на языке логического программирования. Но при написании программы автор на это специально не рассчитывал.

После того, как описаны предикаты (задано, “что” нас интересует), необходимо описать запрос $p \text{ req}(x, Y)$. То есть, необходимо указать, какой предикат p считается входным, какая часть аргумента этого предиката задана (x), а какая —нет (Y). В системе URA запрос определялся следующим образом:

- Входной предикат имеет имя *Job*, его описание должно содержать ровно одно предложение. Запрос имеет вид $\text{Job req}(x, Y)$, где $\text{req}(x, Y)$ —левая часть этого предложения, Y —свободные переменные в левой части;

- Каждый раз, получая новое описание предиката `Job`, `URA` выполняет инверсное вычисление, то есть определяет значения переменных, для которых значение `Job` равно `true` (равно пустой строке).

В микро-Прологе эта проблема решается так: на вход системе подается цель (помечено знаками “`==>`”), то есть вызов некоторого предиката, содержащий константы и переменные. В некоторых случаях, в нашей задаче, в микро-Прологе приходилось перед выдачей цели (запроса) добавлять в систему описания вспомогательных предикатов. Часто это было связано с тем, что была необходима печать найденных значений переменных (встроенный предикат `PP`). В других случаях (запросы 4 и 5) сказывался более бедный, чем в Рефале, набор средств конструирования данных в микро-Прологе⁴.

Ниже приведены тексты запросов для нашей задачи.

1. Найти путь в графе “граф1”	
Job <code>eX = <путь-в-Г1 eX></code>	<code>((Job1 X) (путь-в-Г1 X) (PP X)) ==> (Job1 X)</code>
2. Найти путь из вершины 'A' длиной 5 в графе “граф1”	
Job <code>'A' s1 s2 s3 s4 s5 = + <путь-в-Г1 + 'A' s1 s2 s3 s4 s5></code>	<code>==> (Job1 A X1 X2 X3 X4 X5)</code>
3. Найти путь из 'A' в 'C' длиной 4 в графе “граф1”	
Job <code>'A' s1 s2 s3 'C' = + <путь-в-Г1 + 'A' s1 s2 s3 'C'></code>	<code>==> (Job1 A X1 X2 X3 C)</code>
4. Найти путь из 'A' в 'D' в графе “граф1”	
Job <code>'A' eX 'D' = + <путь-в-Г1 'A'eX'D'></code>	<code>((Job2 X Y) (first Z X) (last Z Y) (Job1 Z)) ((first (X Y) X)) ((last (X) X)) ((last (Y Z) X) (last Z X)) ==> (Job2 A D)</code>

⁴ В микро-Прологе (как и в TSG)—один конструктор (`cons`), в Рефале—два конструктора: ассоциативная конкатенация и заключение выражения в скобки.

5. Найти граф, содержащий пути 'ABCABDAE' и 'BCDEA'	
Job eX =	((Job4 Z)
<путь 'ABCABDAE' (eX)>+	(путь (A B C A B D A E) Z)
<путь 'BCDEA' (eX)>	(путь (B C D E A) Z)
	(PP Z))
	=> (Job4 Z)

Сравнение приведенных выше текстов программ и запросов дополнительно обосновывают тезис, высказанный в начале раздела 4 о сходстве языка R^{inv} (в нашем случае—инверсного Рефала) с языком логического программирования.

Это же сравнение выявляет некоторые различия между Инверсным Рефалом и микро-Прологом: описывая предикаты на языке Рефал, пользователь, ранее работавший с Рефалом, находится в привычной среде и может использовать все средства знакомого ему языка. В частности, в тех случаях, когда в процессе описания предиката $p(x, y)$ ему требуется описать *вспомогательную* функцию (не предикат!) $f(z) \rightarrow u$, ему не обязательно представлять ее в виде предиката $pf(z, u)$, описывающего график функции f . Он может описать функцию f в привычной для него манере. В нашем примере, вспомогательная для предиката *путь-в-Г1* функция-константа *граф1* в Рефале представлена функцией, а в микро-Прологе—предикатом-аксиомой, истинным только на одном значении аргумента.

7.2.3 Основные результаты сравнения инверсного и логического программирования

По результатам анализа инверсного программирования программ-предикатов и логического программирования можно сделать следующие выводы.

Свойства инверсного программирования программ-предикатов позволяют рассматривать инверсный язык R^{inv} как язык логического программирования.

Декларативная семантика языка R^{inv} и написанных на нем программ p^{inv} определяется в том же стиле, что и для логических языков. Процедурная семантика языка R^{inv} и написанных на нем программ p^{inv} имеет черты, во многом схожие с тем, что можно найти в описаниях процедурных семантик логических языков. Однако, *в отличие* от логического программирования, процедурная семантика языка R^{inv} и написанных на нем программ p^{inv} всегда в точности совпадает с декларативной семантикой. По этой причине, нет необходимости посвящать пользователя в детали процедурной семантики и снабжать его средствами управления из инверсной программы процессом инверсного выполнения этой программы.

Инверсное программирование программ-предикатов предоставляет в распоряжение программиста больше изобразительных средств, нежели традиционные языки логического программирования.

Программист, при описании программы-предиката на языке R^{inv} , не обязан вспомогательные конструкции описывать так же как предикаты: он свободен в использовании любых средств языка R . Таким образом, взяв произвольный язык R и построив для него алгоритм инверсного вычисления *ura*, можно получить “язык логического программирования” R^{inv} с любым набором изобразительных средств (как курьезный пример—императивный язык логического программирования).

Традиционные системы логического программирования обычно связывают пользователя в средствах, предлагая ему излагать свои мысли только в терминах отношений (предикатов) и других средств, не противоречащих общепринятой схеме Ковальского [37].

Концепция инверсного программирования не исчерпывается инверсным программированием программ-предикатов. Ее можно использовать для более широкого класса задач (см. раздел 6.1).

7.3 Перенос инверсной семантики

Пусть R —язык реализации (например, TSG), для которого разработаны методы метавычислений, включая средства инверсного вычисления программ: средства представления множеств данных языка R —классы и операции над классами, алгоритм инверсного вычисления программ *ura* и т.д. Везде далее не будет предполагаться, что R обязательно является языком TSG. Однако, потребуем наличия всех понятий, операций и выполнения всех утверждений, что были изложены в предыдущих главах для языка TSG.

Рассмотрим произвольный язык L . Как реализовать инверсные вычисления для этого языка? Конечно, всегда имеется “прямой” способ—повторить для языка L все построения, что были сделаны для языка R . Но это довольно-таки большая работа. А нет ли какого-то простого способа переноса на язык L уже реализованных инверсных вычислений для языка R ? Ответ очевиден—конечно, есть. И чтобы осуществить этот перенос, заметим, что “*метапрограмма *ura* реализует инверсное вычисление программы $p \in R$ при помощи наблюдения за развитием процесса вычисления p на множестве $x \dots$* ”. Таким образом, путь к переносу инверсных вычислений на язык L лежит в *представлении процесса вычисления в языке L программы $p \in L$ в виде некоторого процесса вычисления в языке R* .

Представлять процессы вычисления в L в виде процессов вычисления в R проще всего при помощи интерпретатора $\text{int}L \in R$ языка L , написанного на языке R . Подобный способ для переноса окрестностного анализа с языка R на произвольный язык L был использован в главе 5. Ниже будет описан аналогичный способ переноса инверсных вычислений.

Пусть L —произвольный язык программирования, p —программа на языке L , $\text{int}L$ —интерпретатор языка L , написанный на языке R . Будем считать, что выполнены предположения из раздела 5.3.1 о языке L , программах $p \in L$, данных в языке L , интерпретаторе $\text{int}L$.

Рассмотрим множество X данных программы p , представленное классом $\text{req}[a, \mathcal{X}]$: $X = \langle \text{req}[a, \mathcal{X}] \rangle$, \mathcal{X} —с-переменные из $\text{req}[a, \mathcal{X}]$. Рассмотрим произвольное значение $y \in D$. Выполнение инверсного вычисления в языке L для программы p заключается в построении представления для множества:

$$X_{p^{-1}y} = \{ d \mid d \in \langle \text{req}[a, \mathcal{X}] \rangle, p \overset{*}{\underset{L}{\rhd}} y \}$$

или, другими словами, в нахождении всех таких значений x для \mathcal{X} , при которых выполнено $p \text{ req}[a, x] \overset{*}{\underset{L}{\rhd}} y$ (см. раздел 6.1).

Рассмотрим множество $p:X = \{ p:d \mid d \in X \}$. Данное множество представимо в виде класса (обозначим данный класс $p:\text{req}[a, \mathcal{X}]$), который легко построить по классу

$\text{req}[a, \mathcal{X}]$. При этом выполнено, $p:X = \langle p:\text{req}[a, \mathcal{X}] \rangle$. Для языка TSG указанные построения выполняются и обосновываются следующим образом:

$$\text{req}[p, \mathcal{X}] = (\text{ces}, r) = ([\text{ce}_1, \dots, \text{ce}_n], r).$$

Определим:

$$p:\text{req}[p, \mathcal{X}] = (p:\text{ces}, r) = ([p, \text{ce}_1, \dots, \text{ce}_n], r).$$

Тогда:

$$\begin{aligned} \langle p:\text{req}[p, \mathcal{X}] \rangle &= \{ d' \mid s \in \text{FVS}(p:\text{req}[a, \mathcal{X}]), d' = (p:\text{ces})/.s \}, \\ &= \{ d' \mid s \in \text{FVS}(\text{req}[a, \mathcal{X}]), d' = (p/.s) : (\text{ces}/.s) \}, \\ &= \{ d' \mid s \in \text{FVS}(\text{req}[a, \mathcal{X}]), d' = p : (\text{ces}/.s) \}, \\ &= \{ p:d \mid s \in \text{FVS}(\text{req}[a, \mathcal{X}]), d = \text{ces}/.s \}, \\ &= \{ p:d \mid d \in \langle \text{req}[a, \mathcal{X}] \rangle \}, \\ &= \{ p:d \mid d \in X \} = p:X \end{aligned}$$

Здесь использовано, что p не содержит s -переменных, $p/.s = p$.

Рассмотрим следующее инверсное вычисление в языке R :

$$\text{ura intL}(p:\text{req}[a, \mathcal{X}]) \text{ y } i = [(\mathcal{X}:->x_1, r_1), \dots]$$

Рассмотрим следующие подклассы px'_i класса $p:\text{req}[a, \mathcal{X}]$:

$$px'_i = (p:\text{req}[a, \mathcal{X}])/.(S \mathcal{X}:->x_2)/.(R (\text{RESTR } r_i)).$$

В силу теоремы 50 выполнено:

$$\begin{aligned} \langle [px'_1 + px'_2 + \dots] \rangle &= \langle p:\text{req}[p, \mathcal{X}] \rangle_{\text{intL}^{-1}y} = (p:X)_{\text{intL}^{-1}y} \\ &= \{ d' \mid d' \in p:X, \text{intL } d' \xrightarrow{*}_R y \} = \{ p:d \mid d \in X, \text{intL } (p:d) \xrightarrow{*}_R y \} \\ &= p:\{ d \mid d \in X, p \xrightarrow{*}_L y \} = p:(X_{p^{-1}y}) \end{aligned}$$

С другой стороны, так как p не содержит s -переменных, то выполнено следующее:

$$\begin{aligned} px'_i &= (p:\text{req}[a, \mathcal{X}])/.(S \mathcal{X}:->x_i)/.(R (\text{RESTR } r_i)) \\ &= p:(\text{req}[a, \mathcal{X}])/.(S \mathcal{X}:->x_i)/.(R (\text{RESTR } r_i)) = p:x'_i. \end{aligned}$$

где подкласс x'_i класса $\text{req}[a, \mathcal{X}]$ определен как:

$$x'_i = \text{req}[a, \mathcal{X}]/.(S \mathcal{X}:->x_i)/.(R (\text{RESTR } r_i)).$$

Таким образом, имеем:

$$\begin{aligned} p:X_{p^{-1}y} &= \langle [px'_1 + px'_2 + \dots] \rangle = \langle \sum_i px_i \rangle = \bigcup_i \langle px_i \rangle = \bigcup_i \langle p:x'_i \rangle \\ &= \bigcup_i p:\langle x'_i \rangle = p:\bigcup_i \langle x'_i \rangle = p:\langle \sum_i x'_i \rangle = p:\langle [x'_1 + x'_2 + \dots] \rangle \end{aligned}$$

Следовательно, выполнено:

$$X_{p^{-1}y} = \langle [x'_1 + x'_2 + \dots] \rangle.$$

Получен следующий результат.

Теорема 58

О сведении инверсного вычисления в языке L к инверсному вычислению в языке R .

Результат выполнения следующего инверсного вычисления в языке R :


```

inv :: ProgR -> ProgL -> (Class,D) -> [(Subst,Restr)]
inv intL p (x,y) = ura intL (p::x) y
                    where
                        (...) :: ProgL->Class -> Class
                        p::(ces,r) = ((p:ces),r)

```

Рис. 7.2: Алгоритм инверсного вычисления в произвольном языке L

$$\text{ura intL } (p:\text{req}[a, \mathcal{X}]) \text{ y i} = [(\mathcal{X}:->x_1, r_1), \dots]$$

является результатом инверсного вычисления в языке L, а именно:

- Перечисление

$$[(\mathcal{X}:->x_1, r_1), \dots]$$

является перечисление всех значений с-переменных \mathcal{X} , при которых вычисление в языке L программы p на данных $\text{req}[a, \mathcal{X}]$ завершается с результатом y.

- Перечисление

$$[x'_1 + x'_2 + \dots]$$

попарно непересекающихся подклассов x'_i класса $\text{req}[a, \mathcal{X}]$, является представлением множества $X_{p^{-1}y}$, где $x'_i = \text{req}[a, \mathcal{X}] / (S \mathcal{X}:->x_i) / (R (\text{RESTR } r_i))$

Таким образом, чтобы выполнять инверсные вычисления в нескольких языках L_i достаточно: один раз реализовать инверсные вычисления (алгоритм **ura**) для языка R и обеспечить для каждого языка L_i интерпретатор этого языка intL_i , написанный на языке R.

На рисунке 7.2 приведено описание алгоритма **inv**

$$\text{inv intL } p (\text{req}[a, \mathcal{X}], y) \stackrel{*}{\Rightarrow} [(\mathcal{X}:->x_1, r_1), \dots]$$

реализующего (в полном соответствии с теоремой 58) для любого языка L, инверсное вычисление программы $p \in L$.

Глава 8

Нестандартные семантики

8.1 Модификаторы семантик

Окрестностный анализатор `nan` языка R , по сути, определил в языке новую семантику. Помимо обычного $\text{int } p \ d \xRightarrow{*} \text{res}$ вычисления стало возможным выполнять окрестностный анализ—*нестандартное* вычисление в языке R : $\text{nan } p \ d \xRightarrow{*} (\text{res}, \mathcal{O}^d)$. Следующая организация шага окрестностного тестирования программы $p \in L$ (см. раздел 5.3.3):

$$\text{nbh intL } p \ d \xRightarrow{*} (\text{res}, \mathcal{O}^{(p:d)})$$

позволяет выполнить окрестностный анализ процесса вычисления $p \ d \xRightarrow{*}_L \text{res}$ для любой программы p и любого языка L , где

$$\text{nbh intL } p \ d = \text{nan intL } (p:d)$$

По сути, речь идет о переносе окрестностного анализа, изначально реализованного для языка R на произвольный язык L .

Аналогичный перенос инверсной семантики из языка R в произвольный язык L рассмотрен в разделе 7.3: для любого языка L , любой программы p и любого запроса $\text{req}[a, \mathcal{X}]$, следующее вычисление

$$\text{inv intL } p \ (\text{req}[a, \mathcal{X}], y)$$

является инверсным вычислением в языке L программы p на запросе $(\text{req}[a, \mathcal{X}], y)$. Результатом вычисления является перечисление сужений класса $\text{req}[a, \mathcal{X}]$, соответствующих всем решениям уравнения $p \ \text{req}[a, \mathcal{X}] \xRightarrow{*}_L y$.

Становится возможным в любом языке L связать с любой программой $p \in L$ кроме стандартной функции программы p еще две функции p^{nbh} и p^{inv} :

$$\begin{aligned} p \ d &\stackrel{\text{def}}{=} \text{intL } p \ d, & p &:: D \rightarrow D \\ p^{nbh} \ d &\stackrel{\text{def}}{=} \text{nbh intL } p \ d, & p^{nbh} &:: D^{nbh} \rightarrow R^{nbh} \\ p^{inv} \ (\text{req}, y) &\stackrel{\text{def}}{=} \text{inv intL } p \ (\text{req}, y), & p^{inv} &:: D^{inv} \rightarrow R^{inv} \end{aligned}$$

где $D^{nbh}=D$, $R^{nbh}=(D, \text{Class})$, $D^{inv}=(\text{Class}, D)$, $R^{inv}=[(\text{Subst}, [\text{InEq}])]$.

Таким образом, можно рассматривать помимо исходного языка L с его синтаксисом, предметной областью D и стандартной семантикой, новые языки L^{nbh} и L^{inv} , в которых

синтаксис заимствуется у языка L , а семантика является *нестандартной*, модифицированной при помощи \mathbf{nbh} и \mathbf{inv} , соответственно. Модификации подвержена и предметная область языка—типы входных данных и результатов нестандартных вычислений программ могут измениться.

Следующее определение обобщает рассмотренные примеры.

Определение 42

Пусть m произвольный алгоритм, реализующий функцию со следующим типом:

$$m :: \text{ProgR} \rightarrow D \rightarrow D^m \rightarrow R^m$$

Будем называть m *модификатором* семантик языков программирования.

Пусть L —произвольный язык программирования, для которого выполнены все предположения из раздела 5.3.1, в частности, данные и программы языка L представлены данными языка R . Пусть \mathbf{intL} —интерпретатор языка L , написанный на языке R .

Тогда, следующий язык L^m назовем модификацией языка L , порожденной модификатором m :

- *синтаксис программ* полностью совпадает с синтаксисом языка L ;
- *предметная область*: в качестве входных данных программ используются данные с типом D^m , а результаты вычисления программ на этих данных имеют тип R^m ;
- *семантика*: с каждой программой $p \in D$ свяжем функцию p^m , вычисляемую следующим образом:

$$\begin{aligned} p^m d &\stackrel{\text{def}}{=} m \mathbf{intL} p d \\ p^m &:: D^m \rightarrow R^m \end{aligned}$$

Так определенную семантику языка L^m будем называть *нестандартной семантикой языка L , порожденной модификатором m* , или m -нестандартной семантикой.

В соответствии с определением 42, рассмотренные выше языки $L^{\mathbf{nbh}}$ и $L^{\mathbf{inv}}$ являются модификациями языка L , а семантики этих языков—нестандартными семантиками, порожденными модификаторами \mathbf{nbh} и \mathbf{inv} , соответственно.

По определению 42 любой алгоритм m с “подходящим” типом функции вправе рассматриваться как модификатор семантик. Другое дело, что не для всякого такого m модифицированные языки $L^{\mathbf{inv}}$ будут осмысленными, будут иметь практическую ценность. Представляется трудным сузить определение так, чтобы в рассмотрении оставить только осмысленные модификаторы и одна из причин к тому—расплывчивость и изменчивость понятия “практическая ценность”. Поэтому удовлетворимся данным (широким) определением модификатора, но приведем неформальное описание *одной* из методик построения осмысленных (практически ценных) модификаторов. Для демонстрации данной методики построим пару простых осмысленных модификаторов. Примеры будут построены в предположении, что в языке R поддерживаются не только текстовые (как это было для TSG), но и числовые данные и операции с ними. Описание примеров будет неформальным, но, будем надеяться, достаточно ясным.

Оценка ресурсов. Рассмотрим $\mathbf{int} :: \text{ProgR} \rightarrow D \rightarrow D$ —стандартный интерпретатор R . Внесем в его текст следующие изменения. Заведем несколько счетчиков—по одному на

каждый тип (сложение, умножение и т.п.) арифметической операции. В начале работы интерпретатора всем счетчикам присвоим значение 0. Каждый раз, когда в интерпретаторе выполняется арифметическая операция *в интересах интерпретируемой программы* будем, помимо выполнения операции, увеличивать на 1 соответствующий счетчик. Когда интерпретация программы завершена, будем возвращать не посчитанный результат, а список значений счетчиков. Назовем так измененный интерпретатор **rsrR**. Для любой программы $p \in R$ и данным d программа $\text{rsrR} :: \text{ProgR} \rightarrow D \rightarrow [\text{Int}]$ вычисляет— $\text{rsrR } p \ d \stackrel{*}{\Rightarrow} [n_+, n_-, \dots]$ —оценку вычислительных ресурсов: сколько потребуется выполнить арифметических операций при вычислении p на d .

Определим $\text{rsr} :: \text{ProgR} \rightarrow D \rightarrow D \rightarrow [\text{Int}]$ следующим образом:

$$\text{rsr intL } p \ d = \text{rsrR intL } (p:d).$$

По определению 42 **rsr** является модификатором семантик. Для любого языка L , любой программы $p \in L$ и данных d в результате счета $\text{rsr intL } p \ d$ получаем оценку вычислительных ресурсов для вычисления p на d . Данный пример легко изменить на случай оценки других ресурсов (памяти, дискового пространства и т.п.)

С первого взгляда пример кажется парадоксальным: для оценки требуемых ресурсов некоторого вычисления, проводят эти вычисления и подсчитывают по ходу дела затраченные ресурсы. Более разумно стремиться получить оценку ресурсов до (то есть—без) проведения вычисления p на d . Оказывается, что методы эффективной реализации нестандартных семантик, которые будут обсуждаться в разделе 8.3 позволяют надеяться на возможность автоматического преобразования приведенного алгоритма в алгоритм подсчета ресурсов без проведения (полного) вычисления.

Нестандартные вычисления. Рассмотрим стандартный интерпретатор

$$\text{int} :: \text{ProgR} \rightarrow D \rightarrow D$$

и по его тексту внесем следующие изменения. Допустим во входных данных интерпретируемых программ, в тех позициях, где ожидаются числа, указывать не только числа, но и интервалы. Обозначим через D' соответствующее расширение D исходной предметной области D . В тексте интерпретатора, в тех местах, где выполняются операции над числами *в интересах интерпретируемой программы*, внесем изменения: в качестве операндов операций будем принимать как числа, так и интервалы, а операции будем выполнять в стиле интервальных вычислений. Соответственно, при завершении вычисления в результат (в те позиции, где должны быть числа) возможно попадут интервалы. Назовем так измененный интерпретатор **iarR**. По построению, выполнено: $\text{iarR} :: \text{ProgR} \rightarrow D' \rightarrow D'$.

Определим $\text{iar} :: \text{ProgR} \rightarrow D \rightarrow D' \rightarrow D'$ следующим образом:

$$\text{iar intL } p \ d' = \text{iarR intL } (p:d').$$

По определению 42 **iar** является модификатором семантик. Для любого языка L , любой программы $p \in L$ и любых входных (интервальных) данных $d' \in D'$ вычисления $\text{iar intL } p \ d'$ являются интервальными вычислениями в языке L . Таким образом, достаточно один раз реализовать интервальные вычисления для языка R , чтобы иметь возможность выполнения интервальных вычислений в любом языке L , для которого имеется (обычный) интерпретатор $\text{intL} \in R$.

На базе данного примера легко сформулировать класс аналогичных модификаторов, подменяющих обычные вычисления на их (каким-либо образом расширенный) вариант.

Как пример, можно рассматривать аналитические вычисления, для этого, возможно, потребуется выполнение более трудоемких, чем это описано выше, изменений в исходном тексте интерпретатора языка R (например, при расширении реализации операций сравнения чисел).

В приведенных примерах просматривается следующая общая для них методика построения модификаторов:

1. Построение $mR :: ProgR \rightarrow D^m \rightarrow R^m$ —“необычного интерпретатора” для языка R , способного “необычно” исполнять любые программы на языке R (не только интерпретаторы).
2. Определение алгоритма $m :: ProgR \rightarrow D \rightarrow D^m \rightarrow R^m$

```
m intL p d = mR intL p'd
  where p'd = insert p d
        insert :: D -> Dm -> Dm
        insert p d = .....
```

который из своих аргументов компанует аргументы для mR таким образом, чтобы

- (a) получились требуемые аргументы для mR : программа на языке R — $intL$ и данные для $p'd$ нее, полученные “вставкой p в нужную позицию в d ”;
- (b) в процессе “непривычной” интерпретации программа $intL$ получала свои аргументы в формате $(p:d)$.

Заметим, что модификаторы inv и nbh построены по точно такой же схеме—они основаны на “необычных” интерпретаторах (а точнее, метаинтерпретаторах ura и nan) для языка R . Компановка данных $p'd = insert\ p\ d$ в модификаторах nbh , rsr и iar выполнялась тривиально, при помощи обычного конструктора $(:)$. В модификаторе inv соответствующий фрагмент алгоритма не выглядит столь тривиально.

Приведенная в определении $d:m$ схема $m\ intL\ p\ d$ выполнения вычисления в языке L^m программы p на данных d не может быть признанной эффективной. Действительно, как правило, m является “необычным” интерпретатором для языка R . И в этом случае, рассматриваемые вычисления имеют два уровня интерпретации: m выполняет (необычную) интерпретацию $intL$, который интерпретирует p .

В разделе 8.3 будет предложен метод *автоматического* построения эффективных реализаций нестандартных семантик, основанный на применении одного из методов метавычислений—специализации программ.

8.2 Специализация программ

Пусть $p\ [x,y]$ —программа на R от двух аргументов. Зафиксируем один аргумент, скажем x , положив его равным A . Полученную функцию одного аргумента:

$$p_{(A, _)}\ [y] = p\ [A, y],$$

будем называть проекцией p на $(A, _)$. Аналогичным образом определяется проекция для случая фиксации второго аргумента функции— $p_{(_, B)}\ [x] = p\ [x, B]$,—и проекции по нескольким аргументам программы с произвольной арностью.

Специализатором для языка R называют программу **spec**, которая по тексту программы $p \in R$ и по описанию проекции (что включает определение позиций в списке аргументов p , подлежащих фиксации, и задание значений фиксируемых аргументов) строит эффективную программу p' , реализующую соответствующую проекцию функции p .

Суперкомпиляция—один из методов метавычислений,—позволяет выполнять специализацию программ и получать предельно эффективные программы, реализующие проекции. Но, оставляя вопросы теории суперкомпиляции за рамками данной работы, для дальнейшего изложения воспользуемся более простыми понятиями и обозначениями, принятыми в работах по *смешанным вычислениям*.

Можно рассматривать методы и понятия смешанных вычислений как предельно упрощенные методы и понятия метавычислений и, в частности, теории суперкомпиляции. Заметим, что такое (осознанное) упрощение задачи и методов привело к тому, что именно в работах по смешанным вычислениям были впервые [27] реализованы серьезные примеры специализации программ.

В работах [27, 26] описывается специализатор $\text{spec}[p, m, x]$, где:

- p —программа на языке R от n аргументов, подлежащая специализации;
- m —список (маска) из символов n , каждый из которых—'S' или 'D'—от английского: "static", "dynamic". Символ 'S' в i -той позиции означает необходимость проводить специализацию p по i -тому аргументу (значение для этого параметра есть в списке x , то есть задано статически). Символ 'D' в i -той позиции означает, что не надо проводить специализацию p по i -тому аргументу.
- x —список длины k —значения тех аргументов программы p , по которым выполняется специализация (им соответствуют символы 'S' в строке M), k —число символов 'S' в строке M .

Указанный специализатор по своим аргументам выполняет построение текста программы p' , являющейся реализацией соответствующей проекции функции p исходной программы.

То есть, для любой программы $p \in R$, если:

$$\text{spec}[p, M, x] = p'$$

то для любых данных $y = [y_1, \dots, y_{n-k}]$ выполнено *основное свойство специализатора*:

$$(\text{spec}[p, M, x]) y = p' y = p z,$$

где список $z = [z_1, \dots, z_n]$, составлен из элементов списков x и y , располагаемых в списке z в соответствии с маской m . Например, если программа p имеет два аргумента, то выполнено:

$$\begin{aligned} (\text{spec}[p, 'SS', [x, y]]) [] &= p[x, y] \\ (\text{spec}[p, 'SD', [x]]) [y] &= p[x, y] \\ (\text{spec}[p, 'DS', [y]]) [x] &= p[x, y] \\ (\text{spec}[p, 'DD', []]) [x, y] &= p[x, y] \end{aligned}$$

Рассматриваемый специализатор **spec** построен по следующей схеме [27, 26]:

$$\begin{aligned} \text{spec } [p, m, x] &= s \text{ (annotated_p:} x) \\ \text{where annotated_p} &= \text{ann } p \text{ } m \end{aligned}$$

где ann —алгоритм аннотации программы p , s —основной алгоритм, выполняющий смешанные вычисления.

Далее будем использовать обозначение: $p^{\text{mask}} = \text{ann } p \text{ 'mask'}$. Тогда

$$\text{spec } [p, \text{mask}, x] = s(p^{\text{mask}}: x).$$

В этих обозначениях, основное свойство специализатора для программы с двумя аргументами может быть записано следующим образом:

$$\begin{aligned} (s[p^{ss}, x, y])[] &= p[x, y], \quad (s[p^{sd}, x])[y] = p[x, y], \\ (s[p^{ds}, y])[x] &= p[x, y], \quad (s[p^{dd}])[x, y] = p[x, y]. \end{aligned}$$

Предположим, что s записан на языке R . Это дает возможность применять s к самому себе, что позволяет выполнять цепочку специализаций и автоматически строить по заданным программам новые, реализующие весьма нетривиальные функции.

Рассмотрим знаменитый пример. Пусть p —программа на языке L , d —данные для p , $p[d] \xrightarrow{*} \text{res}$, intL —интерпретатор языка L , написанный на R . Используя несколько раз основное свойство специализатора, получим, что для любой $p \in L$ и любых d выполнено:

$$\begin{aligned} \text{res} &= \text{intL } [p, d] \\ &= s[\text{intL}^{sd}, p] [d] \\ &= s[s^{sd}, \text{intL}^{sd}] [p] [d] \\ &= s[s^{sd}, s^{sd}] [\text{intL}^{sd}] [p] [d] \end{aligned}$$

Построены три новые программы:

1. Программа $pR = s[\text{intL}^{sd}, p]$ на языке R обладает тем свойством, что для любого d выполнено $pR[d] = \text{intL } [p, d] = p \text{ } d$. Таким образом, pR —программа на R , функционально эквивалентная программе p на L . То есть, pR —*результат компиляции программы p с языка L на язык R* .
2. Программа $\text{compL} = s[s^{sd}, \text{intL}^{sd}]$ обладает тем свойством, что для любой $p \in L$ выполнено $\text{compL } [p] = pR$. То есть, compL —*компилятор с языка L на язык R* .
3. Программа $\text{gencomp} = s[s^{sd}, s^{sd}]$ обладает тем свойством, что для любого языка L , для которого представлено “его описание” в виде интерпретатора intL , выполнено $\text{gencomp}[\text{intL}^{sd}] = \text{compL}$. То есть, gencomp является генератором компиляторов.

Итак, при помощи специализации из имеющихся программ p , intL и s автоматически (при помощи вычислений) получены три новые программы с весьма нетривиальными функциями.

Выше построены три знаменитые проекции Футамуры-Турчина, первые упоминания о которых в публикациях относятся к 70-ым годам [23]. Таким образом, уже давно было известно, что в случае практической реализации методов специализации программ будет открыта возможность автоматического построения по имеющимся программам нетривиальных новых программ. Первые же выполненные на практике вычисления по данным проекциям относятся к 1985 г. [27]. Сегодня известно несколько специализаторов, способных строить по-настоящему эффективные специализированные программы. А именно, компиляторы, построенные данными специализаторами по проекциям Футамуры-Турчина выглядят так, что могут быть признаны как написанные квалифицированным программистом [26].

8.3 Эффективная реализация нестандартных языков

Рассмотрим произвольный модификатор m семантик. Допустим, что m реализован на языке R . Тогда, в соответствии с определением 42, для любого языка L и любой программы $p \in L$, любых данных $d \in D^m$ выполнено:

$$p^m d = m [\text{int}L, p, d]$$

Как было отмечено в разделе 8.1, вычисления $p^m d$ по данной схеме весьма неэффективны из-за наличия двух уровней интерпретации.

Воспользуемся специализацией для повышения эффективности вычисления $p^m d$:

$$\begin{aligned} p^m d &= m [\text{int}L, p, d] \\ &= s[m^{SD}, \text{int}L] [p, d] \\ &= s[s^{SD}, m^{SD}] [\text{int}L] [p, d] \\ &= s[s^{SD}, s^{SD}] [m^{SD}] [\text{int}L] [p, d] \end{aligned}$$

Построены три новые программы:

- $\text{mi}L = s[m^{SD}, \text{int}L]$, для любой программы $p \in L$ и любых данных $d \in D^m$ выполнено: $\text{mi}L[p, d] = p^m d$. Таким образом, реализована одноуровневая m -нестандартная интерпретация программы p . Назовем $\text{mi}L$ — m -нестандартным интерпретатором языка L .
- $\text{gmi} = s[s^{SD}, m^{SD}]$, для любого языка L , если $\text{int}L \in R$ —интерпретатор L , то выполнено: $\text{gmi}[\text{int}L] = \text{mi}L$. Назовем gmi генератором m -нестандартных интерпретаторов.
- $\text{ggnsi} = s[s^{SD}, s^{SD}]$, для любого модификатора m выполнено следующее:

$$\text{ggnsi}[m^{SD}] = \text{gmi}.$$

Назовем ggnsi —генератором генераторов нестандартных интерпретаторов.

Получены проекции, описывающие построение средств эффективной (одноуровневой) нестандартной интерпретации для произвольных модификаторов m и произвольных языков L .

Еще раз применим основное свойство специализатора к определению нестандартной семантики, но будем при этом выполнять фиксацию аргументов в ином порядке:

$$\begin{aligned} p^m d &= m [\text{int}L, p, d] \\ &= s[m^{SD}, \text{int}L, p] [d] \\ &= s[s^{SD}, m^{SD}, \text{int}L] [p] [d] \\ &= s[s^{SD}, s^{SD}, m^{SD}] [\text{int}L] [p] [d] \\ &= s[s^{SD}, s^{SD}, s^{SD}] [m^{SD}] [\text{int}L] [p] [d] \end{aligned}$$

Построены четыре новые программы:

- $\text{mp}R = s[m^{SD}, \text{int}L, p]$, $\text{mp}R$ —программа на языке R , для любых $d \in D^m$ выполнено: $\text{mp}R d = p^m d$. Таким образом, $\text{mp}R$ —результат m -нестандартной компиляции с языка L в язык R , компиляции, в процессе которой происходит два преобразования: преобразуется функция программы в соответствии с модификатором m и генерируется текст на языке R , реализующий функцию p^m .

- $mcL = s[s^{SSD}, m^{SSD}, intL]$, для любой $p \in L$ выполнено: $mcL[p] = mpR$. Таким образом, mcL — m -нестандартный компилятор с языка L на язык R .
- $gmc = s[s^{SSD}, s^{SSD}, m^{SSD}]$, для любого языка L , если $intL \in R$ —интерпретатор L , то выполнено: $gmc[intL] = mcL$. Назовем gmc генератором m -нестандартных компиляторов.
- $ggnsC = s[s^{SSD}, s^{SSD}, s^{SSD}]$, для любого модификатора m выполнено $ggnsC[m^{SSD}] = gmc$. Таким образом, $ggnsC$ —генератор генераторов нестандартных компиляторов.

Получены проекции, описывающие построение для произвольных модификаторов m средств m -нестандартной компиляции с произвольного языка L в язык R .

Приведенные выше формулы, при их практическом воплощении, должны обеспечить эффективную реализацию нестандартных семантик. Программист, имея на своем рабочем столе эффективные средства метавычислений (инверторы, окрестностные анализаторы, специализаторы и многие другие, не названные здесь или еще не “открытые” исследователями), будет способен не писать, а вычислять новые программы из старых. И именно в этот момент можно будет говорить о том, что мы достигли автоматизации программирования.

Приведенные выше семь проекций пока еще не были осуществлены на практике. И хотя последние примеры выполнения специализации при помощи суперкомпилятора [21, 22] демонстрируют возможность устранения двойных уровней интерпретации, все же практическая реализация данных проекций потребует большего развития методов метавычислений.

К счастью, совершенствовать методы метавычислений можно беспрестанно (одно из следствий теории суперкомпиляции и общей теории рекурсивных функций). Точно так же не видно ограничений в создании все новых и новых методов применения метавычислений в практическом программировании.

Таким образом, метавычисления всегда останутся непобедимым и благодатным полем для новых исследований.

Литература

- [1] Турчин В.Ф. Эквивалентные преобразования рекурсивных функций описанных на Рефале // Теория языков и методы программирования. Труды Симпозиума по теории языков и методам программирования. Киев–Алушта. стр.31–42.
- [2] Turchin V.F. The Phenomenon of Science // Columbia University Press, New York 1977. (Турчин В.Ф. Феномен науки: Кибернетический подход к эволюции // М., Наука, 1993, 296 с.)
- [3] Turchin V.F. A supercompiler system based on the language Refal // SIGPLAN Notices, 14(2): 46-54, 1979.
- [4] Turchin V.F. The language Refal, the theory of compilation and metasystem analysis // Courant Institute of Mathematical Sciences, New York University. Courant Computer Science Report No. 20, 1980.
- [5] Turchin V.F., Nirenberg R., Turchin D.V. Experiments with a supercompiler // Conference Record of the ACM Symposium on Lisp and Functional Programming. p.47–55, ACM Press 1982.
- [6] Turchin V.F. Program transformation by supercompilation // Ganzinger H., Jones N.D. (ed.), Programs as Data Objects. (Copenhagen, Denmark). Lecture Notes in Computer Science, Vol. 217, p.257–281, Springer-Verlag 1985.
- [7] Turchin V.F. The concept of supercompiler // ACM TOPLAS, 8(3): 292-325, 1986.
- [8] Turchin V.F. Refal: a language for linguistic cybernetics // City College of the City University of New York. 1986
- [9] Turchin V.F. The algorithm of generalization in the supercompiler // Børner D., Ershov A.P., Jones N. (ed.), Partial Evaluation and Mixed Computation. (Gammel Avernæs, Denmark). pp.341–353, North-Holland, 1988.
- [10] Romanenko A.Yu. The generation of inverse functions in Refal // Børner D., Ershov A.P., Jones N. (ed.), Partial Evaluation and Mixed Computation. (Gammel Avernæs, Denmark). pp.427–444, North-Holland, 1988.
- [11] Glück R. Transformation of Refal graphs into Refal programs // City University New York. Technical Report (Presented at the International Seminar on Metacomputation. February 21-22, 1989, New York) 1989.
- [12] Glück R. Inside the supercompiler // City University New York. Technical Report 1989.

- [13] Glück R., Turchin V.F. Experiments with a self-applicable supercompiler // City University New York. Technical Report 1989.
- [14] Glück R., Turchin V.F. Application of metasystem transition to function inversion and transformation // Proceedings of the ISSAC '90. (Tokyo, Japan). 286-287, ACM Press 1990.
- [15] Абрамов С.М. Метавычисления и логическое программирование // Программирование No 3, 1991, стр.31–44.
- [16] Romanenko A.Yu. Inversion and metacomputation // Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation. (Yale University, Connecticut). p.12–22, ACM Press 1991.
- [17] Romanenko A.Yu. A language for direct and inverse programming and its semantics // Institute for the Physical Chemistry, Academy of Sciences of the USSR, Moscow. 1992.
- [18] Glück R., Klimov And. Occam's razor in metacomputation: the notion of a perfect process tree // Cousot P., Falaschi M., Filé G., Rauzy A. (ed.), Static Analysis. Proceedings. (Padova, Italy). Lecture Notes in Computer Science, Vol. 724, 112-123, Springer-Verlag 1993.
- [19] Turchin V.F. Program transformation with metasystem transitions // Journal of Functional Programming, 11, 1993
- [20] Glück R., Klimov And. Metacomputation as a Tool for Formal Linguistic Modeling // Cybernetics and Systems Research'94. World Scientific: Singapore 1994
- [21] Nemytykh A.P., Turchin V.F. Metavariables: Their Implementation and Use in Program Transformation // Technical Report TR 95.6.5.002, The City College of New York, 1995.
- [22] Nemytykh A.P., Turchin V.F. A Self-applicable Supercompiler // Technical Report TR 95.6.5.001, The City College of New York, 1995.
- [23] Futamura Y. Partial evaluation of computation process—an approach to a compiler-compiler // Systems, Computers, Controls, 2(5): 45-50, 1971.
- [24] R.M.Burstall, J.Darlington. A transformational system for developing recursive programmes // JACM, 24(1), 1977, pp. 44–67.
- [25] J.Darlington. An experimental programm transformation and synthesis system.—Artifitial intelligense, 16(1), 1981, pp. 1–46.
- [26] Romanenko S. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure // Børner D., Ershov A.P., Jones N. (ed.), Partial Evaluation and Mixed Computation. (Gl. Avernæs, Denmark). pp.445-463, North-Holland, 1988.

- [27] Jones N.D., Sestoft P., Søndergaard H. Mix: a selfapplicable partial evaluator for experiments in compiler generation // *Lisp and Symbolic Computation*, 2(1): 9-50, 1989.
- [28] Кальниньш А.А., Борзов Ю.В. Инвентаризация идей тестирования программ. Методическая разработка по математическому обеспечению ЭВМ // Рига, ЛГУ, 1981.
- [29] Ramamoorthy C.V., Yj S.B.F., Chen W.T. On the automated generation of program test data // *IEEE Transactions on Software Engineering*, 1976, SE-2, No 4, p.293–300.
- [30] Alberts D.S. The economics of software quality asurance // *AFIPS Conf.Proc.*, 1976 NCC, Montvale, 1976, p.433–442.
- [31] Kernighan B.W., Plauger P.J. The elements of programming style. New York, McGraw-Hill, 1974.
- [32] Moranda P.B. Asymptotic limits to program testing // *INFOTECH State of the Art Report: Software Testing*, vol.2, 1979, p.201–212.
- [33] Мессих И.Г.Б., Штрик А.А. Методика и средства анализа структуры и характеристик сложных комплексов программ реального времени // *Синтез, тестирование, верификация и отладка программ*. Рига: ЛГУ, 1981, с.152-153.
- [34] Rault J.-C., Bouissou B. Quantitative measures for software reliability: a review // *INFOTECH State of the Art Report: Software Testing*. 1979, v.2, p.213-229.
- [35] Abramov S.M. Metacomputation and program testing // *1st International Workshop on Automated and Algorithmic Debugging*. (Linköping, Sweden). p.121–135, Linköping University 1993.
- [36] Братко И. Программирование на языке Пролог для искусственного интеллекта // Мискава, “Мир”, 1990. 559 с.
- [37] Kowalski R. Logic for Problem Solving // *Artificial Intelligence Series*, North Holland, 1979.
- [38] Hudak P., Wadler Ph. et al. Report on the programming language Haskell, a non-strict purely functional language (Version 1.1) // *Technical report Yale University/Glasgow University*. August 1991.
- [39] Manna Z. Theory of computation // New York, 1974, 448 p.

Абрамов С.М.

Методы метавычислений и их применение

Оригинал-макет подготовлен С.М.Абрамовым в системе \LaTeX 2 ϵ