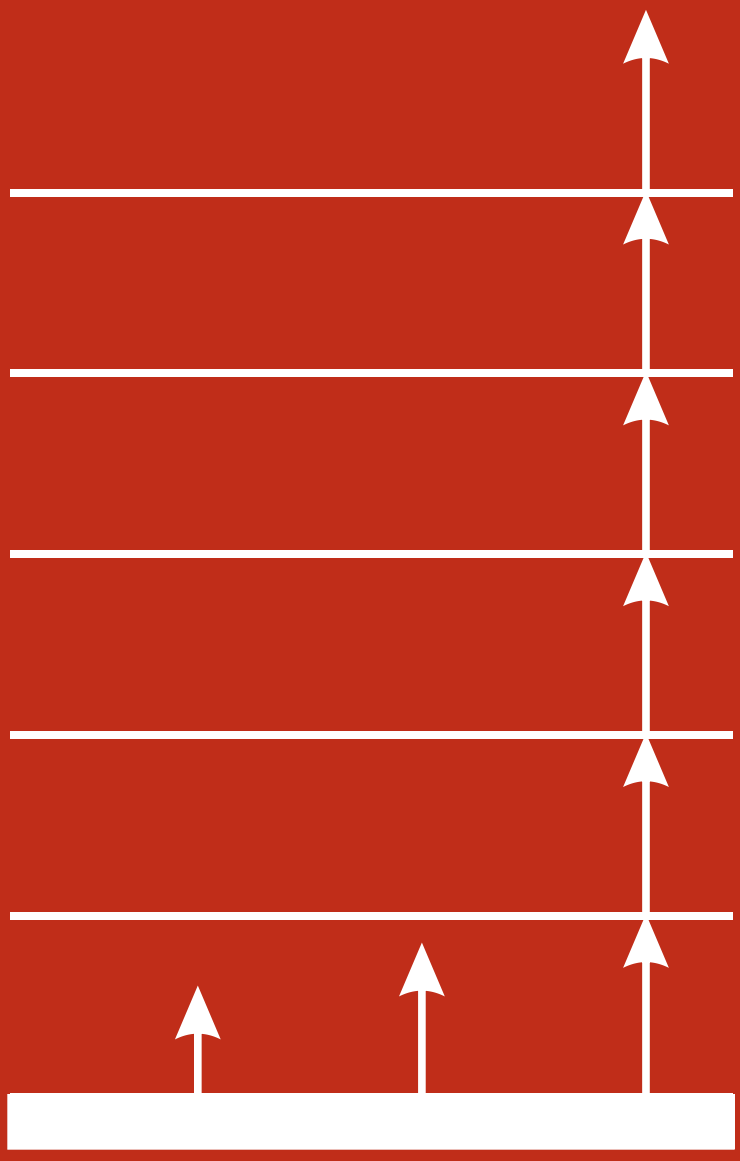
С.М.Абрамов, Л.В.Пармёнова

Метавычисления и их применение. Суперкомпиляция



С.М.Абрамов, Л.В.Пармёнова

Метавычисления и их применение. Суперкомпиляция

Данный материал представляет собой дополнительные главы к курсу «Метавычисления и их применение». Материал подготовлен сотрудниками Института программных систем РАН Абрамовым С.М., Парменовой Л.В.

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

Оглавление

Список иллюстраций

Глава 1

Введение

1.1 Область исследования

Данная работа относится к теории и практике *метавычислений*. Метавычисления—это раздел 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 методы и средства [6, 9, 14, 17, 18, 4, 3]. 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 [6, 8, 9]. Базовыми идеями данных работ являлись:

- применение теории метасистем и метасистемных переходов [7, 9, 20, 7];
- *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 [13] 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) $\mathcal{C}$ данных. По сути, метапрограмма `M` наблюдает за множеством процессов[1] выполнения программы `p` и управляет данными процессами. Таким образом, `M` является метасистемой по отношению к системе `p`. Именно из результатов наблюдения за процессами выполнения

[1] *Процесс-ориентированность*—одна из принципиальных отличительных черт подхода В.Ф.Турчина.

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 самою (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* [8, 12]. Эти усилия оказались плодотворными—в последнее время появились первые реализации суперкомпилятора [11, 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.

Во многом указанный пробел был закрыт работой [3], которая была 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—*универсальноого 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, в работе [3] дан систематическое описание базовых понятий метавычислений и показано, что даже простейшие метапрограммы способны реализовать нетривиальные методы анализа и преобразования программ. Однако, *вопросы теории и практики суперкомпиляции* (из которых и начала формироваться теория метавычисления) *остались за рамками рассмотрения данной работы*.

Цель и задачи исследования. Главной целью данной работы являлось разработка дополнения к [3], которое бы основываясь на стиле, системе обозначений и базовых понятиях из [3] было бы посвящено вопросам теории суперкомпиляции и примерам использования суперкомпилятора.

Данная цель подразумевает решение следующих задач:

1. Разработку, реализацию, описание и обоснование суперкомпилятора для языка TSG (использующего базовые понятия метавычислений над TSG из работы [3]), что включает:
   (a) Разработку алгоритма обнаружения возможного зацикливания.
   (b) Разработку алгоритма обобщения двух конфигураций `cUp` и `cDn`.
   (c) Разработку алгоритма проверки вложенности двух конфигураций $\texttt{cUp} \trianglerighteq \texttt{cDn}$.
   (d) Разработку (на базе упомянутых выше в пунктах 1a–1c функциональных блоков и с использованием алгоритмов из [3]) собственно алгоритма суперкомпилятора.
2. Проведение и описание реальных примеров суперкомпиляции, призванных продемонстрировать способность выполнения суперкомпилятором нетривиальных глубоких преобразований программ.

1.2 Система обозначений и понятий

Как и в [3] в данной работе в качестве языка реализации (для которого строится суперкомпилятор) используется язык TSG—простой и компактный алгоритмически полный язык. Простота TSG и предметной области языка позволяет значительно упростить изложение методов суперкомпиляции для TSG, в то же время не упуская ни одного *существенного* момента.

Заметим, что 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 алгоритмов[2].

В данной работе широго используются терминология, понятия, алгоритмы и утверждения из [3]. В некотором смысле, данная работа является *продолжением (второй частью)* книги [3]. Тем самым, предполагается глубокое знание читателем материала работы [3].

1.3 Структура изложения

В главе 2 описан алгоритм (один из возможных вариантов) `Wh`, реализующий обнаружение возможного зацикливания суперкомпилятора—то есть, обнаружение ситуации *похожей* на построение суперкомпилятором в графе конфигураций бесконечной ветви.

[2] В данной работе во всех построениях используются только следующие свойства языка TSG: *алгоритмическая полнота языка* и определение предметной области языка при помощи конечного набора *жестких* конструкторов над конечным алфавитом атомов.

В нашем проекте реализован алгоритм (функция `whConf`), основанный на *упрощающем сравнении ($\triangleright$) двух конфигураций*. Если на некоторой ветви (начинающейся в корне) графа конфигураций обнаруживаются две вершины с конфигурациями `cUp` и `cDn` (`cUp`—предок `cDn`) такие, что:

```
(cDn ▷ cUp) == True
```

то считают, что конфигурация `cDn` и `cUp` *опасно похожи* (*опасно близки друг к другу*), и следует объявить ситуацию *возможно зацикливание* суперкомпилятора в случае продолжения построения данной ветви.

В главе 3 рассмотрен алгоритм обобщения двух конфигураций `Gener`. Показано, что обобщение двух конфигураций реализуется в две фазы: *обобщение баз* конфигураций и *обобщение рестрикций* конфигураций.

Первая фаза реализована при помощи известного алгоритма *least common generation*, запрограммированного с учетом конструкции конфигураций для языка TSG.

Для реализации второй фазы разработан и обоснован алгоритм обобщения рестрикций, используемых для TSG.

В результате получен алгоритм обобщения, обладающий следующим свойством: *если выполнено* $cUp \succeq cDn$, *то в результате обобщения будет построена конфигурация* $cGen$, *совпадающая с* cUp *с точностью до переименования с-переменных*.

Данное свойство реализованного алгоритма обобщения:

- Демонстрирует, что он строит *тесные* обобщения конфигураций.
- Позволяет легко реализовать проверку условия `cUp` $\succeq$ `cDn` за счет анализа результатов работы алгоритма обобщения (функция `isEqCUpCGen`).

Тем самым, в данном проекте проверка условия `cUp` $\succeq$ `cDn` была отнесена к модулю обобщения, что позволило упростить структуру суперкомпилятора.

В главе 4 введен синтаксис представления графа конфигураций (S-графа) для TSG, рассмотрена операция сведения двух конфигураций в S-графе, определены вспомогательные синтаксические конструкции и функции для суперкомпилятора.

Затем, определен и обсужден собственно суперкомпилятор для языка TSG, основанный на идее выполнения обобщения конфигураций в нижней конфигурации (в ситуации *возможное зацикливание*). Данный суперкомпилятор использует алгоритмы `Wh` и `Gener` (определенные в главах 2 и 3) и алгоритмы, функции и операции из [3].

В завершении главы 4 рассмотрены вспомогательные функции, предназначенные для вывода результата суперкомпиляции (графа конфигураций) в читабельном виде (в виде TSG-программы).

В главе 5 рассмотрены и проанализированы две группы примеров суперкомпиляции TSG-программ разработанным суперкомпилятором:

1. Примеры суперкомпиляции алгоритма `pmatch` на классах вида

   $((\text{[}string\text{, }\mathcal{E}.1), \text{[ ]})$,

   где $string$—некоторая константная строка.

   Рассмотренные примеры позволяют сделать следующий вывод: реализованный по проекту суперкомпилятор способен выполнять нетривиальную специализацию программ.

В частности, программа наивного алгоритма проверки вхождения подстроки в строку при специализации по первой подстроке суперкомпилятором успешно преобразуется в алгоритм Кнута-Морисса-Пратта.

2. Примеры компиляции с языка конечных автоматов в язык TSG. По заданным конечным автоматам и интерпретатору языка конечных автоматов получены программы, функционально эквивалентные заданным конечным автоматам, на языке TSG. То есть, полученные программы есть результат компиляции программ с языка конечных автоматов в язык TSG.

   Рассмотренные примеры позволяют сделать следующий вывод: реализованный по проекту суперкомпилятор способен выполнять нетривиальную первую проекцию Футамуры-Турчина. И тем самым, по заданному интерпретатору `int` некоторого языка `L` суперкомпилятор способен выполнить *компиляцию* любой программы `p` с языка `L` в язык TSG.

1.4 FTP-приложение

В данной работе (как и в книге [3]) используются обозначения, общепринятые в работах по математике и программированию. Для записи алгоритмов метапрограмм использован входной язык системы `Gofer`, являющийся диалектом языка Haskell [23].

Все приводимые в работе тексты метапрограмм являются фрагментами *работающей* Gofer-программы, а все приводимые примеры суперкомпиляции—результатами реального выполнения этой программы.

Полный текст программы и примеров, полный текст данной работы в виде файлов доступны заинтересованному читателю по анонимному FTP:

`ftp://ftp.botik.ru/pub/local/Sergei.Abramov/Scp-proj/`

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 авторы стремились достичь ясности и компактности описания алгоритмов, часто в ущерб эффективности.

Глава 2

Wh—алгоритм обнаружения возможного зацикливания

Цель алгоритма `Wh`—не допустить построения в графе конфигураций бесконечных ветвей и, тем самым—не допустить бесконечно долгой работы суперкомпилятора. Таким образом, наилучшая реализация алгоритма `Wh` связана с точным распознанием ситуации *"программа не останавливается (зацикливается) на заданных входных данных"*. В силу того, что проблема распознания зацикливания программы (проблема останова) алгоритмически неразрешима, мы не можем реализовать (запрограммировать) этот самый сильный вариант алгоритма обнаружения возможного зацикливания. Поэтому, на любом этапе развития технологии суперкомпиляции нам придется довольствоваться алгоритмами обнаружения *возможного* зацикливания.

Алгоритм `Wh` при любой реализации действует следующим образом. Он выполняется после очередного шага развития ветви графа конфигураций и

- либо выдает сигнал о *возможном*[1] зацикливании процесса построения данной ветви;
- либо не выдает такого сигнала.

От любой реализации алгоритма `Wh` требуется выполнения следующего свойства:

Утверждение 1

Основное свойство `Wh`. В любом бесконечном дереве конфигураций на любой бесконечной ветви, начинающейся от корня дерева, обязательно найдутся две конфигурации `cUp` и `cDn`, такие что:

- `cUp` расположенно на пути от корня графа до `cDn` (`cUp`—предок для `cDn`);
- выполнено `(Wh cDn cUp)==True`.

■

Невозможность реализации *точного* алгоритма `Wh` приводит к следующим последствиям:

[1] При этом, в силу упомянутой ранее алгоритмической неразрешимости, нам приходится допускать и истинные сигналы (процесс построение ветви действительно бесконечный), и ложные сигналы о зацикливании.

- существованию многих вариантов различных реализаций `Wh`;
- возможности их сравнения: какой из двух алгоритмов `Wh` "сильнее", то есть выдает меньше ложных сигналов.
- существенно безграничным возможностям улучшения данной компоненты суперкомпилятора.

2.1 Упрощающее отношение

Общие идеи всех известных сегодня различных реализаций Wh основаны на использовании различных формализаций интуитивного понятия "две конфигурации опасно похожи".

В нашем проекте реализован алгоритм `Wh`[2], основанный на так называемом упрощающем сравнении двух термов [1, 2].

Алгоритм `Wh` распознает ситуацию "возможно зацикливание", если в дереве конфигураций встретилась конфигурация, "опасно похожая" с какой-либо построенной ранее конфигурацией. Конфигурации "опасно похожи" в нашем случае означает, что одну из них можно привести к другой с помощью стирания фрагментов в правых частях с-сред, причем имена (индексы) с-переменных при сравнении в рассмотрение не берутся.

В силу теоремы [2, Tree Theorem (Kruskal), p.273–274] в любом бесконечном дереве конфигураций на любой бесконечной ветви, начинающейся от корня дерева, обязательно найдутся две конфигурации `cUp` и `cDn`, такие что:

- `cUp` расположено на пути от корня графа до `cDn` (`cUp`—предок для `cDn`);
- выполнено `(whConf cDn cUp)==True`.

То есть, для данного варианта реализации `Wh` выполнено основное свойство алгоритма распознания возможного зацикливания—выполнено утверждение 1.

2.2 Реализация алгоритма обнаружения возможного зацикливания

В начале описания реализации алгоритма `Wh` рассмотрим функцию `wh` упрощающего сравнения двух с-выражений.

2.2.1 Функция `wh`—упрощающее сравнение двух с-выражений

Аргументы функции `wh` – два с-выражения `x y`, которые она сравнивает между собой, результат – `True` или `False`, в зависимости от того, связаны ли они упрощающим отношением `x` $\lhd$ `y`.

Функция `wh` выполняет упрощающее сравнение следующим образом:

- Если `y` и `x`—два атома, и их значения равны, то то значение функции `True`.

[2] Везде далее будем рассматривать только эту реализацию Wh.

```
wh::CExp->CExp->Bool
wh  (ATOM y)     (ATOM x)       = (x == y)
wh  (CONS y1 y2) x@(CONS x1 x2) = (wh y1 x1)&&(wh y2 x2) ||
                                  (wh y1 x)               ||
                                  (wh y2 x)
wh  (CONS y1 y2) x              = (wh y1 x) || (wh y2 x)
wh  (CVE y)      (CVE x)        = True
wh  (CVA y)      (CVA x)        = True
wh  y            x              = False
```

Рис. 2.1: Функция `wh`—упрощающее сравнение двух с-выражений

- Если аргументы имеют вид `y = (CONS y1 y2)`, `x = (CONS x1 x2)`, где `y1`, `y2`, `x1`, `x2`—с-выражения, то:

  1. либо сравниваются между собой `y1` и `x1`, `y2` и `x2`;
  2. либо сравниваются `y1` и весь второй аргумент `x` (в `y` “стираем” фрагменты `(CONS` и `y2)`);
  3. либо сравниваются `y2` и весь второй аргумент `x` (в `y` “стираем” фрагменты `(CONS y1` и `)`).

- Если первое выражение имеет вид `y = CONS y1 y2`, а второе с-выражение `x` имеет вид отличный от `(CONS x1 x2)`, то функция:

  1. либо сравнивает `y1` и весь второй аргумент `x` (в `y` “стираем” фрагменты `(CONS` и `y2)`);
  2. либо сравнивает `y2` и весь второй аргумент `x` (в `y` “стираем” фрагменты `(CONS y1` и `)`).

- Если входные выражения оба са-переменные или оба се-переменные, то значение функции `True`.

- Во всех остальных случаях значение функции `wh`—`False`.

2.2.2 Алгоритм `numprog` перенумерации термов исходной программы

Чтобы выполнить упрощающее сравнение двух конфигураций, нужно:

1. проверить равенство в этих конфигурациях программных термов и левых частей с-сред,

2. выполнить упрощающее сравнение с-выражений из правых частей с-сред.

Чтобы сделать первую проверку проще, будем использовать нумерацию термов в исходной программе.

Функция `numterm`**.** Нумерацию программных термов производит функция `numterm`.

На входе функции терм `t` и целое число `i`—свободный номер для нумерации термов. Она рассматривает термы двух видов: `ALT` и `CALL`.

```
numterm :: Term -> Int -> (Term, Int)
numterm t@(ALT c t1 t2) i = ((ALT' c t1' t2' i), i'')
                            where
                              (t1', i') = numterm t1 (i+1)
                              (t2', i'') = numterm t2 i'

numterm t@(CALL fn args) i = ((CALL' fn args i), i+1)

numterm t i = (t, i)
```

Рис. 2.2: Функция `numterm`

Когда (см. первое предложение функции `numterm` на рис. 2.2) на входе терм вида `(ALT c t1 t2)`, где `c`—условие, `t1 t2`—термы, происходит нумерация термов `t1` и `t2`, и всего терма `(ALT c t1 t2)`. В результате получаем терм `(ALT' c t1' t2' i)` и новое значение свободного номера `i''`.

Когда (см. второе предложение функции `numterm` на рис. 2.2) на входе терм вида `(CALL fn args)` и `i`, то возвращается пронумерованный терм `(CALL' fn args i)`, а счетчик `i` увеличивается на единицу.

При термах другого вида значение счетчика не изменяется, терм не нумеруется.

```
numdefs :: [FDef] -> Int -> ([FDef], Int)
numdefs [ ] i = ([ ], i)
numdefs ((DEFINE fn prms t):fds) i = ((DEFINE fn prms t'):fds', i'')
                              where
                                (t', i') = numterm t i
                                (fds', i'') = numdefs fds i'
```

Рис. 2.3: Функция `numdefs`

Функция `numdefs`**.** С помощью функции `numdefs` перенумеровываем определения функций в программе:

- Если на входе пустой список определений функций, то результат—пустой список.
- Когда входной список не пуст, то его первый элемент имеет вид:

  `(DEFINE fn prms t)`.

  Сначала при помощи функции `numterm` выполняется нумерация терма `t` (и его подтермов). Полученный терм `t'` записываеся в качестве тела определения функции `fn` (первый элемент списка-результата `numdefs`):

`(DEFINE fn prms t')`

продолжение `fds'` списка определений функций формируется в результате рекурсивного вызова функции `numdefs`. Общий результат работы функции `numdefs` в рассматриваемом случае:

`((DEFINE fn prms t'):fds', i'')`

где `i''`—новое значение свободного номера для термов.

```
numprog :: Prog -> Prog
numprog p =   p'   where (p',_) = numdefs p 1
```

Рис. 2.4: Функция `numprog`

Функция `numprog`**.** Функция `numprog` выполняет глобальную перенумерацию термов в исходной программе. Функция numprog получает на входе программу `p`, а результатом ее работы является программа `p'`, где `p'`—результат работы функции `numdefs` с аргументами `p 1`.

2.2.3 Функция `theSameTerm`

Функция `theSameTerm` является вспомогательной для функции `whConf`, сравнивающей (упрощающее сравнение) две конфигурации. Функция `theSameTerm` сравнивает два программных терма на равенство. Все термы в программе занумерованы. Таким образом, они равны, если совпадают их номера. Функция рассматривает два вида термов: альтернатива и вызов функции.

```
theSameTerm :: Term -> Term -> Bool
theSameTerm (ALT' _ _ _ i) (ALT' _ _ _ j) = (i==j)
theSameTerm (CALL' _ _ i)  (CALL' _ _ j)  = (i==j)
theSameTerm _              _              = False
```

Рис. 2.5: Функция `theSameTerm`

Заметим, что если в двух некоторых конфигурациях

`cUp = ((tUp, cenvUp), rUp),`
`cDn = ((tDn, cenvDn), rDn)`

перенумерованные термы `tDn` и `tUp` равны,

`theSameTerm tUp tDn == True`,

то это обеспечивает равенство длин списков с-связей `cenvUp` и `cenvDn` и попарное равенство всех левых частей в с-связях из `cenvDn` и `cenvUp`. То есть, в этом случае конфигурации имеют вид:

tDn = tUp = t,
cUp = ((t, cenvUp), rUp),
cDn = ((t, cenvDn), rDn),
(length cenvUp) == (length cenvDn),
cenvUp = [(var_1 := $cexpUp_1$),..., (var_n := $cexpUp_n$)],
cenvDn = [(var_1 := $cexpDn_1$),..., (var_n := $cexpDn_n$)].

2.2.4 Функция whConf

Функция whConf имеет в качестве аргументов две конфигурации cDn и cUp. Она сравнивает (упрощающее сравнение) их между собой, результат ее работы—True или False. Эти конфигурации имеют следующий вид:

cDn = ((tDn, cenvDn), rDn)
cUp = ((tUp, cenvUp), rUp)

где tDn, tUp—термы, cenvDn, cenvUp—списки с-связей, rDn, rUp—рестрикции.

```
whConf:: Conf -> Conf -> Bool
whConf ((tDn, cenvDn), _) ((tUp, cenvUp), _) =
           (theSameTerm tDn tUp) &&
           and (zipWith
                  (\ (varUp:=cexpUp) (varDn:=cexpDn)->(wh cexpDn cexpUp))
                cenvDn cenvUp)
```

Рис. 2.6: Функция whConf

Для сравнения термов используется функция theSameTerm. Если термы tDn, tUp равны, то это обеспечивает (см. раздел 2.2.3) попарное равенство всех левых частей в списках с-связей cenvDn и cenvUp. А именно, если

theSameTerm tUp tDn == True,
tUp = tDn = t

то конфигурации имеют следующий вид:

cUp = ((t, cenvUp), rUp),
cDn = ((t, cenvDn), rDn),
(length cenvUp) == (length cenvDn),
cenvUp = [(var_1 := $cexpUp_1$),..., (var_n := $cexpUp_n$)],
cenvDn = [(var_1 := $cexpDn_1$),..., (var_n := $cexpDn_n$)].

Таким образом, после проверки theSameTerm tDn tUp остается убедится, что все правые части $cexpDn_i$ и $cexpUp_i$ из соответствующих с-связей попарно связаны упрощающим отношением:

$cexpDn_i \triangleright cexpUp_i$.

Данная проверка обеспечивается выполнением проверок

wh $cexpDn_i$ $cexpUp_i$

для всех $i = 1, ..., n$.

2.3 Выводы

В главе 2 описан алгоритм (один из возможных вариантов) `Wh`, реализующий обнаружение возможного зацикливания суперкомпилятора—то есть, обнаружение ситуации *похожей* на построение суперкомпилятором в графе конфигураций бесконечной ветви.

В нашем проекте реализован алгоритм (функция `whConf`), основанный на *упрощающем сравнении (▷) двух конфигураций*. Если на некоторой ветви (начинающейся в корне) графа конфигураций обнаруживаются две вершины с конфигурациями `cUp` и `cDn` (`cUp`—предок `cDn`) такие, что:

```
(cDn ▷ cUp) == True
```

то считают, что конфигурация `cDn` и `cUp` *опасно похожи* (*опасно близки друг к другу*), и следует объявить ситуацию *возможно зацикливание* суперкомпилятора в случае продолжения построения данной ветви.

В силу теоремы [2, Tree Theorem (Kruskal), p.273–274] в любом бесконечном дереве конфигураций на любой бесконечной ветви, начинающейся от корня дерева, обязательно найдутся две конфигурации `cUp` и `cDn`, такие что:

- `cUp` расположено на пути от корня графа до `cDn` (`cUp`—предок для `cDn`);
- выполнено `(cDn ▷ cUp)==True`.

То есть, для данного варианта реализации `Wh` выполнено основное свойство алгоритма распознания возможного зацикливания—выполнено утверждение 1.

Глава 3

Gener—алгоритм построения тесной общей надконфигурации

3.1 Постановка задачи

Алгоритм `Gener` предназначен для построения тесной общей надконфигурации для двух заданных конфигураций.

Пусть имеются две конфигурации `cUp`, `cDn` с одним и тем же термом из перенумерованной программы. То есть, для данных конфигураций (см. раздел 2.2.3) выполнено:

`cUp = ((tUp, cenvUp), rUp)`,
`cDn = ((tDn, cenvDn), rDn)`,
`theSameTerm tUp tDn == True`,
`tUp = tDn = t`,
`(length cenvUp) = (length cenvDn) = n`,
`cenvUp = [(`$\mathtt{var}_1$ `:=` $\mathtt{cexpUp}_1$`),...,(`$\mathtt{var}_n$ `:=` $\mathtt{cexpUp}_n$`)]`,
`cenvDn = [(`$\mathtt{var}_1$ `:=` $\mathtt{cexpDn}_1$`),...,(`$\mathtt{var}_n$ `:=` $\mathtt{cexpDn}_n$`)]`.

Требуется построить такую конфигурацию `cGen = ((t,cenvGen), rGen)`, что:

`cGen` $\succeq$ `cUp`,
`cGen` $\succeq$ `cDn`,

и данное обобщение необходимо сделать *тесным*.

В силу [3, утверждение 22, стр. 40] условия `cGen` $\succeq$ `cUp`, `cGen` $\succeq$ `cDn` равносильны следующему: существуют подстановки `sGenUp`, `sGenDn` и рестрикции `r'Up`, `r'Dn`, такие, что:

```
cUp = cGen /. sGenUp /. r'Up,
cDn = cGen /. sGenDn /. r'Dn.
```

Последние два равенства можно преобразовать к следующим условиям:

`cGen = ((t, cenvGen), rGen)`,
`cenvGen = [(`$\mathtt{var}_1$ `:=` $\mathtt{cexpGen}_1$`),...,(`$\mathtt{var}_n$ `:=` $\mathtt{cexpGen}_n$`)]`,
$\mathtt{cexpGen}_i$ `/. sGenUp ==` $\mathtt{cexpUp}_i$, $\mathtt{cexpGen}_i \succeq \mathtt{cexpUp}_i$,
$\mathtt{cexpGen}_i$ `/. sGenDn ==` $\mathtt{cexpDn}_i$, $\mathtt{cexpGen}_i \succeq \mathtt{cexpDn}_i$,
`rGen /. sGenUp /. r'Up == rUp`,
`rGen /. sGenDn /. r'Dn == rDn`.

Таким образом, для построения обобщения cGen пары конфигураций cUp и cDn необходимо

1. построить *тесное* общее надвыражение $[cexpGen_1, \ldots, cexpGen_n]$ для правых частей с-сред:

   $[cexpUp_1, \ldots, cexpUp_n]$ и
   $[cexpDn_1, \ldots, cexpDn_n]$

   конфигураций cUp, cDn, и

2. выполнить *тесное* обобщение рестрикций—построить наиболее сильные ограничения rGen, удовлетворяющие условию:

   существуют r'Up и r'Dn такие, что:
   rGen /. sGenUp /. r'Up == rUp,
   rGen /. sGenDn /. r'Dn == rDn,

   где, sGenUp и sGenDn—подстановки, определяемые соотношениями:

   $cexpGen_i$ /. sGenUp == $cexpUp_i$,
   $cexpGen_i$ /. sGenDn == $cexpDn_i$.

Первый из этих двух шагов будем везде ниже называть *обобщением с-баз*, второй—*обобщением рестрикций*.

3.2 Обобщение с-баз конфигураций

Для построения тесного общего надвыражения для правых частей с-сред из cUp и cDn воспользуемся известным алгоритмом (*least common generation, LCG*), позволяющем для двух жестких с-баз xUp и xDn построить *минимальную* общую надбазу xGen. То есть, построить xGen такую, что:

xGen $\succeq$ xUp,
xGen $\succeq$ xDn,
для любого x', такого, что x' $\succeq$ xUp и x' $\succeq$ xDn, выполнено x' $\succeq$ xGen.

В нашем случае в качестве жестких с-конструкций будут выступать правые части с-сред:

xUp = $[cexpUp_1, \ldots, cexpUp_n]$,
xDn = $[cexpDn_1, \ldots, cexpDn_n]$,
xGen = $[cexpGen_1, \ldots, cexpGen_n]$).

3.2.1 Вспомогательные конструкции и функции

Таблица обобщений. Основной вспомогательной конструкцией для алгоритма обобщения с-баз двух конфигураций является *таблица обобщений*. Элементы этой таблицы относятся к типу GTabEl. Каждый элемент—это пара, состоящая из с-переменной и двух с-выражений:

```
(cvar, (cexp1, cexp2)).
```

Если в таблице имеется элемент `(cvar, (cexp1, cexp2))`, то это означает, что ранее в процессе обобщения с-баз конфигураций было принято решение фрагмент `cexp1` из `cDn` и фрагмент `cexp2` из `cUp` обобщить с-переменной `cvar`. То есть, данные фрагменты в `cGen` представлены с-переменной `cvar`.

Функция `gLookUp`**.** Функция `gLookUp` проверяет, построено ли обобщение для двух данных с-выражений `cexp1` и `cexp2`. Если обобщение для них уже построено, то есть, если в таблице обобщений имеется элемент

```
(cvar, (cexp1, cexp2))
```

то результат работы функции—список из одной переменной `[cvar]`, которая и является обобщением этих выражений.

Если обобщение для `cexp1` и `cexp2` еще не построено, то результатом функции `gLookUp` будет пустой список `[ ]`.

Получая на входе таблицу `tab` и два с-выражения `cexp1` и `cexp2`, функция выбирает все такие с-переменные `cvar`, что элемент `(cvar, (cexp1,cexp2))` принадлежит таблице `tab`.

```
gLookUp :: GTab -> CExp -> CExp -> [CVar]
gLookUp t cexp1 cexp2 = [ cvar | (cvar,(a,b))<-t, a==cexp1,b==cexp2]
                      -- либо [], либо [cvar]
```

Рис. 3.1: Функция `gLookUp`

Функции `isAExp` **и** `isAtom`**.** Функция `isAExp` проверяет, является ли данное с-выражение са-выражением. Са-выражением являются либо атом, либо са-переменная.

Функция `isAtom` проверяет, является ли данное с-выражение атомом.

```
isCAExp ::  CExp -> Bool
isCAExp  (ATOM _) = True
isCAExp  (CVA _)  = True
isCAExp  _        = False

isAtom :: CExp -> Bool
isAtom  (ATOM _) = True
isAtom  _        = False
```

Рис. 3.2: Функции `isAExp` и `isAtom`

3.2.2 Обобщение с-выражений

Функция `genCExp` обобщает два с-выражения. Функция потребляет на входе таблицу обобщений `t`, два с-выражения, свободный индекс. Результатом является тройка, состоящая из новой таблицы обобщений, с-выражения, которое обобщает данные с-выражения, нового значения свободного индекса.

Функция реализует известный алгоритм (*least common generation, LCG*) построения наименьшего общего обобщения.

```
genCExp::GTab -> CExp -> CExp -> FreeIndx  ->  (GTab,CExp,FreeIndx)

genCExp t a@(ATOM x)  (ATOM y)     i= (t, a, i), if (x==y)

genCExp t (CONS x1 x2)(CONS y1 y2) i= (t'', (CONS z1 z2), i'')
                              where
                                 (t',  z1, i')  = genCExp t  x1 y1 i
                                 (t'', z2, i'') = genCExp t' x2 y2 i'

genCExp t x y i = case (gLookUp t x y) of
                    [z] -> (t , z, i )
                    [ ] -> (t', v, i')
                           where
                             (v, i') = if ((isCAExp x)&&(isCAExp y))
                                       then (mkCAVar i)
                                       else (mkCEVar i)
                             t'      = (v, (x, y)):t
```

Рис. 3.3: Функция `genCExp`. Обобщение с-выражений

Обобщение происходит следующим образом:

1. Если на входе `genCExp` два равных атома, то таблица `t` остается прежней, свободный индекс тот же, а выражение, обобщающее эти атомы, равно им самим.

2. Если на входе `genCExp` вида `(CONS x1 x2)` и `(CONS y1 y2)`, то

   (a) Строится обобщение `z1` для `x1` и `y1`:

   `(t', z1, i') = genCExp t x1 y1 i.`

   (b) Затем строится обобщение `z2` для `x2` и `y2`:

   `(t'', z2, i'') = genCExp t' x2 y2 i'`

   (c) После чего формируется новое значение таблицы, обобщение для с-выражений `(CONS x1 x2)` и `(CONS y1 y2)`, новое значение свободного индекса:

   `(t'', (CONS z1 z2), i'')`

3. При других видах входных с-выражений `x` и `y` их обобщение зависит от работы функции `gLookUp`.

(a) Если результат обобщения данных выражений уже записан в таблице, то есть, существует список из одного элемента, являющегося обобщением `x` и `y`:

`(gLookUp t x y) = [z]`,

то результатом работы `genCExp` является неизмененная таблица, этот элемент `z`, свободный индекс:

`(t, z, i)`

(b) Если в результате работы `gLookUp` имеем пустой список:

`(gLookUp t x y) = [ ]`,

то `x` и `y` будут обобщены новой с-переменной `v`.

А именно, если эти выражения являются са-выражениями (что проверяет функция `isAExp`), то с помощью функции `mkCAVar` заводим новую са-переменную `v`, которая и будет обобщать данные выражения. В противном случае (`x` или `y` не является са-выражением), с помощью функции `mkCEVar` заводим се-переменную `v`, обобщающую выражения `x` и `y`.

После этого к началу таблицы `t` приписываем новый элемент `(v, (x,y))`, и результатом работы `genCExp` в этом случае является

`(t', v, i')`,

где `t'`—измененная таблица, `v`—переменная, обобщающая `x` и `y`, `i'`—свободный индекс.

3.2.3 Обобщение с-сред

Обобщение с-сред выполняется функцией `genCEnv`. Как сказано в разделе 3.1, обобщение с-баз конфигураций с равными термами сводится к обобщению правых частей с-сред из этих конфигураций. Таким образом, функция `genCEnv` является несложным расширением `genCExp` на список с-связей.

Функция `genCEnv` получает на входе таблицу, две с-среды, свободный индекс. На выходе функции—таблица обобщений, обобщенная с-среда, свободный индекс.

```
genCEnv :: GTab -> CEnv -> CEnv -> FreeIndx -> (GTab,CEnv,FreeIndx)
genCEnv t [] [] i = (t, [], i)
genCEnv t ((a1:=cexp1):t1) ((a2:=cexp2):t2) i =
                      (t'', ((a1:=cexp):cenv), i'')
                      where
                         (t',  cexp, i' ) = genCExp t cexp1 cexp2 i
                         (t'', cenv, i'') = genCEnv t' t1 t2 i'
```

Рис. 3.4: Функция `genCEnv`. Обобщение с-сред

Вычисление результата `genCEnv` выполняется следующим образом:

- Если входные с-среды—пустые списки, то обобщенная с-среда—также пустой список, а таблица и свободный индекс остаются неизменными.

- Если на входе с-среды следующего вида:

  ```
  ((a1:=cexp1):t1)
  ((a2:=cexp2):t2),
  ```

  где a1, a2—переменные, cexp1, cexp2—с-выражения, то результат работы функции имеет вид:

  ```
  (t'', ((a1:=cexp):cenv), i'')
  ```

  Здесь t''—измененная таблица обобщений; ((a1:=cexp):cenv)—обобщенная с-среда; i''—свободный индекс; с-выражение cexp, обобщающее выражения cexp1 и cexp2, вычисляется функцией genCExp; с-среда cenv, обобщающая с-среды t1 и t2, вычисляется рекурсивным вызовом genCEnv.

  Заметим, что во входных с-средах переменные a1, a2 равны, поэтому в обобщенную среду в качестве левой части можно поместить любую из этих двух переменных, например a1.

3.3 Функция genTabToSubsts. Построение подстановок sGenUp и sGenDn

Рассмотрим две с-базы (два с-выражения или две с-среды) cxDn и cxUp. Выполним построение (при помощи функций genCExp или genCEnv) для них минимальной общей над-с-базы cxGen:

```
(t, cxGen, i') = genCExp [ ] cDn cUp i
```
или
```
(t, cxGen, i') = genCEnv [ ] cDn cUp i
```

Непосредственно из текстов алгоритмов обобщения с-баз (см. функции genCExp и genCEnv) видно, что:

- все с-переменные в обобщающей в с-базе cxGen (с-выражении, с-среде) это те (и только те) с-переменные v_i, которые представлены в первой позиции элементов таблицы обобщений:

  $t = [\ (v_1, (x_1, y_1)), \ldots, (v_n, (x_n, y_n))\ ]$.

- эти с-переменные v_i связаны в таблице обобщений как раз с теми фрагментами из исходных с-баз, которые и были обобщены этой с-переменной v_i:

  1. с фрагментом x_i из первой обобщаемой с-базы cxDn;
  2. с фрагментом y_i из второй обобщаемой с-базы cxUp.

Таким образом, после выполнения обобщения с-баз таблица обобщений t по сути представляет две подстановки sGenDn и sGenUp:

$t = [\ (v_1, (x_1, y_1)), \ldots, (v_n, (x_n, y_n))\ ]$
$sGenDn = [\ (v_1 :\text{-}> x_1), \ldots, (v_n :\text{-}> x_n)\ ]$
$sGenUp = [\ (v_1 :\text{-}> y_1), \ldots, (v_n :\text{-}> y_n)\ ]$

такие, что:

```
cxGen /. sGenDn = cxDn
cxGen /. sGenUp = cxUp
```

На рисунке 3.5 приведена функция `genTabToSubst`, которая по таблице обобщений `t` строит две упомянутые выше подстановки `sGenDn` и `sGenUp`.

```
genTabToSubsts :: GTab -> (Subst,Subst)
genTabToSubsts t = ([v:->x | (v,(x,y))<-t ], [v:->y| (v,(x,y))<-t ])
```

Рис. 3.5: Функция `genTabToSubsts`. Построение подстановок `sGenUp` и `sGenDn` по таблице обобщений `t`

3.4 Обобщение рестрикций

Пусть имеются две конфигурации `cUp`, `cDn` с одним и тем же термом из перенумерованной программы. То есть, для данных конфигураций (см. раздел 2.2.3) выполнено:

```
cUp = ((tUp, cenvUp), rUp),
cDn = ((tDn, cenvDn), rDn),
theSameTerm tUp tDn == True,
tUp = tDn = t.
```

Требуется построить такую конфигурацию `cGen = ((t,cenvGen), rGen)`, что:

```
cGen ≥ cUp,
cGen ≥ cDn,
```

и данное обобщение необходимо сделать *тесным* в некотором смысле.

В разделе 3.1 показано, что для построения обобщения `cGen` пары конфигураций `cUp` и `cDn` необходимо:

1. *выполнить обобщение с-баз*—построить *тесную* общую над-с-среду `cenvGen` для с-сред `cenvDn` и `cenvUp`:

   ```
   cenvGen ≥ cenvDn,
   cenvGen ≥ cenvUp.
   ```

2. выполнить *тесное* обобщение рестрикций—построить наиболее сильные ограничения `rGen`, удовлетворяющие условию:

   существуют `r'Up` и `r'Dn` такие, что:
   ```
   rGen /. sGenUp /. r'Up == rUp,
   rGen /. sGenDn /. r'Dn == rDn,
   ```

   где `sGenDn` и `sGenUp`—подстановки, определяемые соотношениями:

```
cenvGen /. sGenDn == cenvDn,
cenvGen /. sGenUp == cenvUp.
```

В разделе 3.2 описано решение первой задачи—построения *минимального* обобщения `cenvGen` для двух с-сред `cenvDn` и `cenvUp` заданных конфигураций:

```
(tab, cenvGen, i') = genCEnv [ ] cenvDn cenvUp i
```

В разделе 3.3 описано построение подстановок `sGenDn` и `sGenUp`:

```
(sGenDn, sGenUp) = genTabToSubsts tab
```

удовлетворяющих соотношениям:

```
cenvGen /. sGenDn == cenvDn,
cenvGen /. sGenUp == cenvUp.
```

Таким образом, для завершения построения тесного обобщения конфигураций осталось построить обобщение рестрикций: необходимо построить наиболее сильные ограничения rGen, удовлетворяющие условию:

существуют `r'Up` и `r'Dn` такие, что:
`rGen /. sGenUp /. r'Up == rUp`,
`rGen /. sGenDn /. r'Dn == rDn`.

Это условие распадается на два независимых условия:

Существует `r'Dn` такое, что: `(rGen /. sGenDn) +. r'Dn == rDn`.
Существует `r'Up` такое, что: `(rGen /. sGenUp) +. r'Up == rUp`.

каждое из которых можно упростить следующим образом:

- *Рестрикция `(rGen /. sGenDn)` не сильнее рестрикции `rDn`*: все неравенства из `(rGen /. sGenDn)` входят в `rDn`.
- *Рестрикция `(rGen /. sGenUp)` не сильнее рестрикции `rUp`*: все неравенства из `(rGen /. sGenUp)` входят в `rUp`.

Следовательно, в `rGen` могут входить *только такие неравенства* `(x:=/=:y)`, для которых выполнено:

- После применения подстановки `sGenDn` неравенство `(x:=/=:y)` обращается либо в тавтологию, либо в неравенство из `rDn`.
- После применения подстановки `sGenUp` неравенство `(x:=/=:y)` обращается либо в тавтологию, либо в неравенство из `rUp`.

С другой стороны, `rGen` должно быть как можно более сильным ограничением, то есть содержать как можно больше неравенств.

Таким образом, мы приходим к тому, что `rGen` должно содержать *такие и все такие неравенства* `(x:=/=:y)`, что:

- После применения подстановки `sGenDn` неравенство `(x:=/=:y)` обращается либо в тавтологию, либо в неравенство из `rDn`.

- После применения подстановки `sGenUp` неравенство `(x:=/=:y)` обращается либо в тавтологию, либо в неравенство из `rUp`.

Пусть `r`—одна из рестрикций `rDn` или `rUp`, `s`—соответствующая подстановка `sGenDn` или `sGenUp`. Обозначим через:

```
mkRestr r s
```

рестрикцию, содержащую *такие, и все такие* неравенства `(x:=/=:y)`, что после применения подстановки `s` неравенство `(x:=/=:y)` обращается либо в тавтологию, либо в неравенство из `r`.

Тогда, построение `rGen` сводится к следующему:

- Построение рестрикции `(RESTR rGDn) = mkRestr rDn sGenDn`.
- Построение рестрикции `(RESTR rGUp) = mkRestr rUp sGenUp`.
- Включение в `rGen` тех и только тех неравенств, которые входят и в `rGDn` и в `rGUp` (пересечение множеств неравенств):

```
rGen = RESTR [ n | n<-rGDn, n `elem` rGUp ]
```

3.4.1 Общая схема выполнения обобщения рестрикций

Выше задача построения обобщения рестрикций сведена к следующему:

1. Реализации функции `mkRestr` построения (по заданным `r` и `s`) рестрикции

   `(mkRestr r s)`,

   содержащей *такие, и все такие* неравенства `(x:=/=:y)`, что после применения подстановки `s` неравенство `(x:=/=:y)` обращается либо в тавтологию, либо в неравенство из `r`.

   Здесь, `r` — одна из рестрикций `rDn` или `rUp`, `s` — соответствующая подстановка `sGenDn` или `sGenUp`.

2. Простому вычислению `rGen` при помощи данной функции и вычисления пересечения множеств неравенств:

```
rGen = RESTR [ n | n<-rGDn, n `elem` rGUp ]
where
(RESTR rGDn) = mkRestr rDn sGenDn
(RESTR rGUp) = mkRestr rUp sGenUp
```

3.4.2 Функция `mkRestr`

Пусть, `r`—одна из рестрикций `rDn` или `rUp`, `s`—соответствующая подстановка `sGenDn` или `sGenUp`.

Функция `mkRestr` предназначена для построения (по заданным `r` и `s`) рестрикции (`rG = mkRestr r s`), содержащей *такие, и все такие* неравенства `(x:=/=:y)`, что после

применения подстановки s неравенство (x:=/=:y) обращается либо в тавтологию, либо в неравенство из r.

Таким образом, результат rG функции mkRestr является объединением двух рестрикций:

```
rG = (RESTR r1) +. (RESTR r2)
```

где

- r1—список всех таких неравенств (x:=/=:y), что после применения подстановки s неравенство (x:=/=:y) обращается в неравенство из r.
- r2—список всех таких неравенств (x:=/=:y), что после применения подстановки s неравенство (x:=/=:y) обращается в тавтологию.

Построение r1. Рассмотрим произвольное неравенство (x:=/=:y) такое, что после применения к нему подстановки s оно обращается в некоторое неравенство (a:=/=:b) из r. В силу определения неравенств, x, y, a, b являются са-выражениями, то есть, либо атомами, либо са-переменными. Условие *"после применения подстановки s неравенство (x:=/=:y) обращается в (a:=/=:b)"* можно расписать следующим образом:

```
x /.s = a,
y /.s = b
```

Таким образом, для операции (/.s) са-выражение x является прообразом a, са-выражение y является прообразом b.

То есть, в список r1 входят те и только те неравенства (x:=/=:y) для которых выполнено:

```
x ∈ (invAImg a), y ∈ (invAImg b)
```

где (a:=/=:b) некоторое неравенство из r, (invAImg c)—список всех са-выражений прообразов са-выражения c для операции (/.s).

Вспомогательная функция (invAImg c) определяется просто (см. рис. 3.6):

1. Если c—атом, то в список всех са-выражений прообразов c для операции (/.s) следует включить c.
2. Кроме того в список всех са-выражений прообразов c для операции (/.s) следует включить все такие *са-переменные* x, которые подстановкой s, заменяются на c—то есть, для которых в подстановке s имеется пара (x:->c).

Приведенное выше обсуждение обосновывает следующий способ построения списка r1 всех таких неравенств (x:=/=:y), что после применения подстановки s неравенство (x:=/=:y) обращается в неравенство из r:

```
r1 = [ x:=/=:y | (a:=/=:b) <- r, x<-(invAImg a), y<-(invAImg b) ]
```

```
mkRestr :: Restr -> Subst -> Restr
mkRestr  (RESTR r) s = (RESTR r1) +. (RESTR r2)
         where
            r1=[ x:=/=:y | (a:=/=:b) <- r, x<-(invAImg a),
                                           y<-(invAImg b) ]

            invAImg c = (if (isAtom c) then [c] else [ ]) ++
                        [ x | (x@(CVA _):->z)<-s,  z==c ]

            r2=[ x:=/=:y | (x@(CVA _):-> (ATOM a)) <- s,
                           (y@(CVA _):-> (ATOM b)) <- s, a /= b ]
```

Рис. 3.6: Функция `mkRestr`

Построение `r2`. Рассмотрим произвольное неравенство `(x:=/=:y)` такое, что после применения к нему подстановки `s` оно обращается в некоторую тавтологию `(a:=/=:b)`, причем неравенство `(x:=/=:y)` само тавтологией не является.

В силу определений неравенств и тавтологий, `x` и `y` являются са-выражениями, то есть, либо атомами, либо са-переменными, `a` и `b` являются различными `(a /= b)` атомами и можно записать, что:

```
x /.s = a,
y /.s = b
```

Так как список `r2` является списком рестрикций над обобщающими са-переменными (са-переменными из левых частей подстановки `s`), то можно определить этот список следующим образом: в список `r2` входят все такие неравенства `(x:=/=:y)` и только они, для которых выполнены условия

1. Са-выражения `x` и `y` являются са-переменными;
2. В подстановке `s` имеются пары `(x :-> a)` и `(y :-> b)`, причем `a` и `b` являются различными `(a /= b)` атомами.

Вычисление списка `r2` в соответствии с данными условиями приведено на рисунке 3.6.

Функция mkRestr. Приведенные выше обсуждения построения списков неравенств `r1` и `r2` завершают построение функции `mkRestr`: ее результат является объединением этих двух наборов неравенств (см. рис. 3.6):

```
mkRestr (RESTR r) s = (RESTR r1) +. (RESTR r2)
```

Заметим, что определение `r1` и `r2` приведены в наиболее простом (для понимания) варианте, неэффективном с точки зрения сложности вычислений. Кроме того, данные определения приводят к записи в списки `r1` и `r2` нескольких дубликатов одного и того же неравенства. Однако это не существенно: используемая для окончательного формирования результата операция `(+.)` включает в себя удаление повторных экземпляров одного и того же неравенства.

3.4.3 Функция genRestr. Обобщение рестрикций

По заданным

- рестрикциям rUp и rDn;
- подстановкам sGenUp и sGenDn;

функция genRestr (см. рис. 3.7) выполняет построение тесного обобщения rGen рестрикций в полном соответствии с схемой, приведенной в разделе 3.4.1.

```
genRestr::Restr -> Restr -> Subst -> Subst  ->  Restr

genRestr rDn rUp sGenDn sGenUp = rGen
                      where
                         (RESTR rGDn) = mkRestr rDn sGenDn
                         (RESTR rGUp) = mkRestr rUp sGenUp
                         rGen = RESTR [ n | n<-rGDn, n `elem` rGUp ]
                             -- пересечение списков неравенств
```

Рис. 3.7: Функция genRestr. Обобщение рестрикций

3.5 Функция genConf. Обобщение конфигураций

Пусть имеются две конфигурации cUp, cDn с одним и тем же термом из перенумерованной программы. То есть, для данных конфигураций (см. раздел 2.2.3) выполнено:

```
cUp = ((tUp, cenvUp), rUp),
cDn = ((tDn, cenvDn), rDn),
theSameTerm tUp tDn == True,
tUp = tDn = t.
```

Функция genConf строит тесное обобщение для заданных cDn и cUp: конфигурацию cGen = ((t,cenvGen), rGen) такую, что: cGen $\succeq$ cUp, cGen $\succeq$ cDn. Построение выполняется в две фазы (см. раздел 3):

1. *Обобщение с-сред.* Вычисляется обобщение cenvGen для с-сред, таблица обобщений tab и подстановки sGenUp, sGenDn:

   ```
   (tab, cenvGen, i') = genCEnv [] cenvDn cenvUp i
   (sGenDn, sGenUp) = genTabToSubsts tab
   ```

2. *Обобщение рестрикций.* Вычисляется обобщение rGen для рестрикций rDn, rUp:

   ```
   rGen = genRestr rDn rUp sGenDn sGenUp
   ```

Окончательный результат функции — упорядоченная тройка: таблица обобщений tab, обобщающая конфигурация cGen, свободный индекс i с-переменных — формируется из результатов данных вычислений:

```
(tab, ((t, cenvGen), rGen), i').
```

```
genConf::Conf -> Conf -> FreeIndx -> (GTab, Conf, FreeIndx)

genConf ((tDn, cenvDn), rDn) ((tUp, cenvUp), rUp) i =
             (tab, ((tDn, cenvGen), rGen), i')
             where
                (tab, cenvGen, i') = genCEnv [] cenvDn cenvUp i
                (sGenDn, sGenUp)   = genTabToSubsts tab
                rGen               = genRestr rDn rUp sGenDn sGenUp
```

Рис. 3.8: Функция genConf. Обобщение конфигураций

3.6 Функция isEqCUpCGen. Проверка вложения конфигураций cUp $\succeq$ cDn

Пусть имеются две конфигурации cUp, cDn с одним и тем же термом из перенумерованной программы. То есть, для данных конфигураций (см. раздел 2.2.3) выполнено:

```
cUp = ((tUp, cenvUp), rUp),
cDn = ((tDn, cenvDn), rDn),
theSameTerm tUp tDn == True,
tUp = tDn = t.
```

Алгоритм обобщения конфигураций позволяет по данным конфигурациям cUp и cDn вычислить обобщающую конфигурацию cGen=((t, cenvGen), rGen) и подстановки sGenUp и sGenDn

```
(tab, cGen, i') = genConf cDn cUp i
(sGenDn, sGenUp) = genTabToSubsts tab
```

такие, что:

```
cGen ≥ cUp,
cGen ≥ cDn.
```

В силу построения *тесного* обобщения имеется возможность при помощи анализа результатов обобщения cDn и cUp проверить вложения конфигураций: проверить условие cUp $\succeq$ cDn. А именно, если выполнено условие cUp $\succeq$ cDn то в результате обобщения будет построена конфигурация cGen совпадающая с cUp, с точностью до переименовки переменных.

Таким образом, проверка условия cUp $\succeq$ cDn может быть сведено к проверке условия *"cUp совпадает с cGen с точностью до переименовки переменных"*, или, что равносильно следующему условию:

Существует подстановка-перестановка s, *такая, что выполнено*

cenvGen /. s == cenvUp и rGen /. s == rUp.

Заметим, что:

- Подстановка `sGenUp` по построению определена из соотношения:

  `cenvGen /. sGenUp == cenvUp`.

- Так как, `cGen` $\succeq$ `cUp`, то рестрикция `(rGen /. sGenUp)` не сильнее рестрикции `rUp`, то есть, каждое неравенство из `(rGen /. sGenUp)` входит в `rUp`.

Это позволяет проверку условия `cUp` $\succeq$ `cDn` свести к проверке следующих двух условий (см. рис. 3.9):

1. Подстановка

   `sGenUp = [ (`v_1`:->`x_1`),...,(`v_n`:->`x_n`)]`

   является подстановкой-перестановкой, то есть

   (a) все пары `(`v_i`:->`x_i`)` в `sGenUp` имеют следующий вид: v_i и x_i—с-переменные с *одинаковыми типами* (либо обе са-переменные, либо обе се-переменные);

   (b) все с-переменные в списке `xs = [`x_1`,...,`x_n`]` различны (или, другими словами, в `xs` нет повторных переменных: `xs == nub xs`).

2. Каждое неравенство из `rUp` входит в `(rGen /. sGenUp)` (или, другими словами, список `missed` всех тех неравенств из `rUp`, которые не вошли в `rGen /. sGenUp`, является пустым: `null missed`).

На рисунке 3.9 дано описание функции `isEqCUpCGen`, осуществляющей проверку вложения конфигураций: `cUp` $\succeq$ `cDn`, или, что то же самое, проверку совпадения с точностью до переименования с-переменных конфигураций `cGen` и `cUp`. Функция реализована в полном соответствии с приведенным выше анализом.

```
isEqCUpCGen :: Conf -> Conf -> Subst   ->   Bool
isEqCUpCGen cUp cGen sGenUp = isRenaming && (null missed)
        where
            isRenaming = (all (sametypes) sGenUp) && (xs == nub xs)

            sametypes ((CVA i) :-> (CVA j)) = True
            sametypes ((CVE i) :-> (CVE j)) = True
            sametypes (     _  :->    _   ) = False

            xs = [ x | (v:->x) <- sGenUp ]

            (_, RESTR rGen) = cGen
            (_, RESTR rUp)  = cUp
            r' = rGen /. sGenUp
            missed = [ x | x<-rUp, not (x `elem` r') ]
```

Рис. 3.9: Функция `isEqCUpCGen`. Проверка вложения конфигураций: `cUp` $\succeq$ `cDn`

3.6.1 Функция mkSubstUpDn

В функции `mkSubstUpDn` выполняется построение (для случая `cUp` $\succeq$ `cDn`) подстановки `sUpDn`, приводящей конфигурацию `cUp` к `cDn`

В том случае, когда `cUp` $\succeq$ `cDn`, то есть

```
isEqCUpCGen cUp cGen sGenUp == True
```

в суперкомпиляторе будет необходимо знать подстановку `sUpDn`, приводящую конфигурацию `cUp` к `cDn`. Заметим, что эта подстановка может быть легко построена из подстановок `sGenUp` и `sGenDn`. Действительно:

- Подстановка `sGenUp`

  $$\texttt{sGenUp = [(v}_1\texttt{:->x}_1\texttt{),...,(v}_n\texttt{:->x}_n\texttt{)]}$$

  приводит конфигурацию `cGen` к `cUp` и данная подстановка является подстановкой переименовкой, `cGen` совпадает с `cUp` с точностью до переименования `sGenUp` с-переменных. Следовательно, прямая инверсия `sUpGen` подстановки `sGenUp`:

  $$\texttt{sUpGen = (invSubst sGenUp) = [(x}_1\texttt{:->v}_1\texttt{),...,(x}_n\texttt{:->v}_n\texttt{)]}$$

  является подстановкой-переименовкой и *подстановка* ***sUpGen*** *приводит конфигурацию* ***cUp*** *к* ***cGen****.*

- *Подстановка* ***sGenDn*** *приводит конфигурацию* ***cGen*** *к* ***cDn****.*

- Из предыдущих двух пунктов следует, что суперпозиция `sUpDn` двух подстановок `sUpGen = (invSubst sGenUp)` и `sGenDn`:

  ```
  sUpDn = (invSubst sGenUp).*.sGenDn
  ```

  является подстановкой, которая приводит конфигурацию `cUp` к `cDn`.

На рисунке 3.10 дано описание функций `mkSubstUpDn` и `invSubst`, осуществляющих (для случая `cUp` $\succeq$ `cDn`) построения подстановки `sUpDn`, приводящей конфигурацию `cUp` к `cDn`. Функции реализованы в полном соответствии с приведенным выше анализом.

```
mkSubstUpDn :: Subst -> Subst -> Subst
mkSubstUpDn sGenUp sGenDn = (invSubst sGenUp).*.sGenDn

invSubst :: Subst -> Subst
invSubst sGenUp = [ x:->v | (v:->x)<-sGenUp ]
```

Рис. 3.10: Функция `mkSubstUpDn`. Построение подстановки `sUpDn`, приводящей конфигурацию `cUp` к `cDn` (для случая `cUp` $\succeq$ `cDn`)

3.7 Выводы

В главе 3 рассмотрен алгоритм `Gener`. Показано, что обобщение двух конфигураций реализуется в две фазы: *обобщение баз* конфигураций и *обобщение рестрикций* конфигураций.

Первая фаза реализована при помощи известного алгоритма *least common generation*, запрограммированного с учетом конструкции конфигураций для языка TSG.

Для реализации второй фазы разработан и обоснован алгоритм обобщения рестрикций, используемых для TSG.

В результате получен алгоритм обобщения, обладающий следующим свойством: *если выполнено* $\texttt{cUp} \succeq \texttt{cDn}$, *то в результате обобщения будет построена конфигурация* $\texttt{cGen}$, *совпадающая с* $\texttt{cUp}$ *с точностью до переименования с-переменных*.

Данное свойство реализованного алгоритма обобщения:

- Демонстрирует, что он строит *тесные* обобщения конфигураций.
- Позволяет легко реализовать проверку условия $\texttt{cUp} \succeq \texttt{cDn}$ за счет анализа результатов работы алгоритма обобщения.

Тем самым, в данном проекте проверка условия $\texttt{cUp} \succeq \texttt{cDn}$ была отнесена к модулю обобщения, что позволило упростить структуру суперкомпилятора.

Глава 4

Суперкомпилятор для языка TSG

4.1 Структура графа конфигураций и основные типы данных

4.1.1 Сведение конфигураций в S-графе

Граф конфигураций (S-граф) является расширением понятия *"дерево конфигураций* [3]. Отличия (S-граф) от дерева конфигураций в том, что в S-графе введена возможность сводить одну конфигурацию (`cX`) к другой (`cY`).

Пусть в S-графе имеются конфигурации `cX`, `cY`, и выполнено `cX` $\preceq$ `cY`. Тогда:

- конфигурация `cX` является частным случаем конфигурации `cY`;
- функции, определяемые конфигурациями связаны соотношениями:

  $\mathtt{F}_{cX}$`(vars cX) =` $\mathtt{F}_{cY}$`(vars cY /.` $\mathtt{s}_{cY \rightarrow cX}$`)`

  где `vars cX`, `vars cX`—списки с-переменных, входящих в `cX` и `cY`; $\mathtt{s}_{cY \rightarrow cX}$`)`—подстановка, сводящая `cY` к `cX`.

Таким образом, можно вершину `cX` свести к вершине `cY`, а функцию $\mathtt{F}_{cX}$`(vars`$_{cX}$`)` определить на языке TSG следующим образом:

`(DEFINE` $\mathtt{F}_{cX}$ `vars cX (CALL` $\mathtt{F}_{cY}$ `((vars cY) /.` $\mathtt{s}_{cY \rightarrow cX}$`)))`

На рисунке 4.1 приведены функции `mkArgs` и `mkParms`, определяющие для рассмотренного выше случая:

- аргументы функции $\mathtt{F}_c$, определенной конфигурацией `c`:

  `mkArgs c = vars c.`

- параметры функции $\mathtt{F}_{cY}$, при сведении к ней функции $\mathtt{F}_{cX}$:

  `mkParms cY` $\mathtt{s}_{cY \rightarrow cX}$ `= (mkArgs cY) /.` $\mathtt{s}_{cY \rightarrow cX}$

```
mkArgs  :: Conf -> [CVar]
mkArgs  c   = (cvars c)

mkParms :: Conf -> Subst -> [CExp]
mkParms c s = (mkArgs c)/.s
```

Рис. 4.1: Вспомогательные функции для сведения конфигурации

4.1.2 Синтаксис графа конфигураций

Граф конфигураций имеет один из следующих видов:

- Пассивная вершина

  `(PASSIVE k c cexp)`,

  где `k`—уникальный номер вершины; `c`—пассивная конфигурация вида

  `c = ((exp, cenv), r)`;

  `exp`—выражение; `cenv`—с-среда; `cexp=exp/.cenv`—с-выражение, обобщенный результат вычисления конфигурации `c`.

- Вершина вида

  `(WHISTLE kDn kUp sgraph)`,

  где `kDn` и `kUp`—уникальные номера вершин, `sgraph`—S-граф. Данная вершина введена для отладки и визуального анализа результатов суперкомпиляции. Ее всегда можно заменить на `sgraph`. Вершина порождается если во время суперкомпиляции алгоритмом `Wh` обнаружена ситуация *конфигурация с номером* ***kDn*** *"опасно похожа" на конфигурацию с номером* ***kUp***.

- Вершина сведения одной конфигурации к другой:

  `CALLC string kX cX kY cexps sgraphs`,

  где `kX`—номер вершины с конфигурацией `cX`, сведенной суперкомпилятором к вершине с некоторой конфигурацией (`cY`) и с номером `kY`, `cexp`—аргументы сведения: `cexp = vars`$_{cY}$ `/.` $s_{cY} \rightarrow {}_{cX}$; `str`—строка-коментарий (либо `"UP"`, либо `"DN"`), `str` вместе с `sgraphs` определяют тип сведения:

  1. либо `str = "Up"`, `sgraphs = [ ]`—сведение вершины `kX` к вершине-предку `kY`;
  2. либо `str = "Dn"`, `sgraphs = [sgraph]`—сведение вершине `kX` к вершине `kY`, которая является корнем графа конфигураций `sgraph`.

- Drive-вершина

(DRIVE k c sbranches),
sbranches = [(cnt$_1$, sgr$_1$),..., (cnt$_n$, sgr$_2$)],

где k—номер вершины с конфигурацией c; sgr$_i$—S-графы с некоторыми конфигурациями c$_i$ в корневой вершине; cnt$_i$—сужения, такие, что конфигурация c /. cnt$_i$ при обобщенном вычислении переходит в конфигурацию c$_i$.

4.1.3 Nodeinfo—информация о вершине

Развитие графа конфигураций из некоторой вершины однозначно определяется упорядоченной тройкой с типом Nodeinfo—следующей информацией об этой вершине:

ni = (k, c, h)

где k—номер вершины (номер конфигурации); c—конфигурация в данной вершине; h = [(k$_1$, c$_1$), ..., (k$_1$, c$_1$)]—история вершины—список всех номеров k$_i$ и конфигураций c$_i$, предшествующих рассматриваемой вершине (номера и конфигурации из вершин-предков).

4.2 Вспомогательные функции

Функция isPassive. Функция-предикат isPassive проверяет, является ли терм пассивным—то есть, отличается ли он от CALL- и ALT-терма (является ли он выражением).

```
isPassive :: Term -> Bool
isPassive (CALL' f args _)    = False
isPassive (ALT' cnd t1 t2 _)  = False
isPassive exp                 = True

isWh :: Conf -> History -> [(Int, Conf)] -- либо [ ], либо [(kUp, cUp)]
isWh _ [ ]        = [ ]
isWh c ((i, h):t) = if   (whConf c h)
                    then [(i,h)]
                    else isWh c t
```

Рис. 4.2: Вспомогательные функции суперкомпилятора

Функция isWh. Функция isWh, просматривая историю h некоторой конфигурации c, проверяет, нет ли в истории h конфигурации c' (с номером k') "опасно близкой" с конфигурацией c.

Если h будет найдена такая конфигурация c' с номером k', то функция isWh возвратит список с этой (одной) парой [(k', c')].

Если в h такой конфигурации не будет найдено, то функция isWh возвратит пустой список [ ].

4.3 Входные данные суперкомпилятора, начальная вершина

На вход суперкомпилятору (функция `scp`, см. рисунок 4.3) подается программа `p` и класс `cl = (ces, r)`—обобщенное данное для программы `p`, где `ces`—список с-выражений, `r`—рестрикция.

Все алгоритмы суперкомпилятора рассчитаны на работу с программой с перенумерованными термами, поэтому, в начале своей работы, функция `scp` выполняет перенумерацию программы:

```
pn = numprog p.
```

Затем выполняется построение стартовой конфигурации `c` и вычисляется свободный индекс `i`—точно так же, как это выполнялось в алгоритме построения дерева процессов и алгоритме окрестностного анализа [3]:

```
(DEFINE f prms _) : p' = pn
ce = mkEnv prms ces
c = ((CALL' f prms 0, ce), r)
i = freeindx 1 cl
```

Из результата развития (функция `evalG`) в графа конфигураций начальной вершины `ni = (i,c,[])`:

```
(gr, i') = evalG (i,c,[]) pn (i+1)
```

функция `scp` извлекает S-граф `gr`, который и является результатом суперкомпиляции программы `p` на обобщеных данных `cl`.

```
scp :: ProgR -> Class -> SGraph
scp   p cl@(ces, r) = gr
                      where
                         pn = numprog p
                         (DEFINE f prms _) : p' = pn
                         ce = mkEnv prms ces
                         c  = ((CALL' f prms 0, ce), r)
                         i  = freeindx 1 cl
                         (gr, i') = evalG (i,c,[]) pn (i+1)
```

Рис. 4.3: Суперкомпилятор. Функция `scp`

4.4 Развитие вершины

4.4.1 Функция `evalG`. Построение пассивной вершины

Функция `evalG` является входной в группе функций, осуществляющих развитие (преобразование) информации о вершине `ni` в граф конфигураций.

```
evalG ::Nodeinfo->ProgR->FreeIndx -> (SGraph, FreeIndx)

evalG ni@(k, c@((t, ce),r), h) p i =
                        ((PASSIVE k c (t/.ce)),i) , if (isPassive t)

evalG ni@(k, c@((t, ce),r), h) p i =
                        procDR ni dbrs p i'
                        where
                           (dbrs, i') = drive ni p i
```

Рис. 4.4: Суперкомпилятор. Функция `evalg`

Функция `evalG` анализирует терм `t` в информации о вершине `ni=(k, c, h)`, где `k`—номер вершины, `h`—история вершины, `c=((t, ce),r)`—конфигурация, `t`—терм, `ce`—с-среда, `r`—рестрикция.

Если `t`—пассивный терм, то формируется пассивная вершина (так, как это определено в разделе 4.1.2).

Если `t`—непассивный терм, то выполняется *прогонка* вершины `ni` при помощи функции `drive`. Полученный список `dbrs` вариантов прогонки вершины анализируется функцией `procDR`, которая и определяет дальнейшее развитие непассивной вершины `ni`.

4.4.2 Функция `drive`. Прогонка непассивной вершины на один шаг

Функция `drive` выполняет прогонку непассивной вершины `ni = (k, c, h)` на один шаг, где `k`—номер вершины, `h`—история вершины, `c = ((t, ce),r)`—конфигурация, `t`—терм, `ce`—с-среда, `r`—рестрикция.

Выполнение функции (см. рисунок 4.5) организована точно по той же схеме, что и в фрагментах алгоритмов построения дерева конфигураций и окрестностного анализа [3], а функция обобщенного вычисления условия в `ALT`-термах заимствуется из этих алгоритмов.

Результатом работы функции `drive` является список `dbrs` вариантов прогонки вершины `ni`:

$$\texttt{dbrs} = [(\texttt{cnt}_1, \texttt{c}_1), \ldots, (\texttt{cnt}_2, \texttt{c}_2)]$$

где $\texttt{cnt}_i$—альтернативные сужения (объединение всех $\texttt{<c/.cnt}_i\texttt{>}$ совпадает с `<c>`), $\texttt{c}_i$—конфигурации, поученные выполнением одного шага обобщенного вычисления конфигурации $\texttt{c/.cnt}_i$.

Существенным отличием функции `drive` от соответствующих фрагментов упомянутых алгоритмов из [3] является немедленное применения к списку `dbrs` *операции удаления сухих ветвей*, реализованной функцией `delEmptyDBr`.

Функция `delEmptyDBr` просматривает список вариантов прогонки вершины `ni`

$$\texttt{dbrs} = [(\texttt{cnt}_1, \texttt{c}_1), \ldots, (\texttt{cnt}_2, \texttt{c}_2)]$$

```
drive ::Nodeinfo->ProgR->FreeIndx -> ([DBranch], FreeIndx)
drive ni@(k, c@(( CALL' f args _, ce), r), h) p i =
                  ([(idC, c')] , i)
                  where
                     DEFINE _ prms t' = getDef f p
                     ce' = mkEnv prms (args/.ce)
                     c'  = ((t',ce'),r)

drive ni@(k, c@(( ALT' cnd t1 t2 _, ce), r), h) p i =
                  ( (delEmptyDBr [(cnt1,c1),(cnt2,c2)]), 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)
```

Рис. 4.5: Суперкомпилятор. Функция `drive`

```
delEmptyDBr :: [DBranch] -> [DBranch]

delEmptyDBr [ ]                          = [ ]
delEmptyDBr (dbr:dbrs)                   =
                  case dbr of
                     (_,(_,INCONSISTENT)) -> delEmptyDBr dbrs
                     _                    -> dbr:(delEmptyDBr dbrs)
```

Рис. 4.6: Функция `delEmptyDBr`—удаление сухих ветвей в списке вариантов прогонки вершины

и удаляет из него ветви $(\texttt{cnt}_i, \texttt{c}_i)$ с заведомо пустыми ($\texttt{<c}_i\texttt{>} = \emptyset$) конфигурациями, у которых рестрикции несовместны (`INCONSISTENT`). Список оставшихся (после удаления сухих ветвей) вариантов прогонки вершины `ni` и возвращается как окончательный вариант функции `drive`.

4.4.3 Функция `procDR`. Анализ результатов прогонки

Функция `procDR` вызывается из функции `evalG` (см. рисунок 4.4). Среди своих аргументов, `procDR` получает

- непассивную вершину `ni = (k, c, h)`, где `k`—номер вершины, `h`—история вершины, `c`—конфигурация вершины;
- список `dbrs` оставшихся (после удаления сухих ветвей) вариантов прогонки на один шаг вершины `ni` (результат функции `drive`, см. раздел `s:SDrive`):

dbrs = [(cnt$_1$, c$_1$), ..., (cnt$_n$, c$_n$)]

где cnt$_i$—альтернативные сужения (объединение всех <c/.cnt$_i$> совпадает с <c>), c$_i$—конфигурации, поученные выполнением одного шага обобщенного вычисления конфигурации c/.cnt$_i$.

Функция procDR выполняет анализ (см. рисунок) результатов прогонки на один шаг вершины ni и строит ее развитие.

```
procDR ::Nodeinfo->[DBranch]->ProgR->FreeIndx -> (SGraph, FreeIndx)

procDR ni@(k,c,h) dbrs@[ ] p i       = ((CALLC "Z" k c 0 [] []),i)

procDR ni@(k,c,h) dbrs@[(cn,c')] p i = evalG (k,c',h) p i

procDR ni@(k,c,h) dbrs p i =
    case (isWh c h) of
       [ ]         -> mkDbr ni dbrs p i
       [(kUp,cUp)] -> ((WHISTLE k kUp gr), i')
                      where (gr, i') = procWh ni kUp cUp p i
```

Рис. 4.7: Функция procDR. Анализ результатов прогонки

Вершина без продолжения. Если у вершины ni нет вариантов развития

dbrs = [ ],

то в этом случае функцию конфигурации c

F$_k$ (vars$_c$)

следует положить нигде не определенной.

Обозначим через F0 [ ] нигде не определенную функцию с пустым списком аргументов. Тогда, можно дать следующее определение для F$_k$

(DEFINE F$_k$ (vars$_c$) (CALL F0 [ ]))

В обозначениях синтаксиса графа конфигураций то же самое определение задается вершиной:

((CALLC "Z"k c 0 [] []),i)

Именно такую вершину и строит функция procDR, если у ni нет ни одного варианта развития (см. первое предложение на рисунке 4.7).

Замечание. Непосредственно из текстов алгоритмов нетрудно обнаружить, что ситуация *вершина без продолжения* не реализуема для TSG—первое предложение в функции procDR никогда не будет выполняться. Однако мы умышленно оставили для полноты текста, чтобы полно высветить общую структуру суперкомпилятора (без привязки к конкретному языку реализации).

Транзитный шаг. Если у вершины `ni` один вариант развития—`dbrs = [(cn, c')]`, то в этом случае происходит бесусловный (транзитный) переход от конфигурации `c` к конфигурации `c'`. В реализованном в нашем проекте суперкомпилятора (как и в большинстве суперкомпиляторов) такой переход выполняется (см. второе предложение на рисунке 4.7) по принципу *"без проверки на зацикливание, без порождения транзитной вершины (update in place) и наращивания истории"*. То есть, результат развития вершины `ni = (k, c, h)` полагается равным развитию вершины `(k, c', h)`. Именно такой вариант и приведен на рисунке 4.7.

Вершина с несколькими продолжениями. Если у вершины `ni` несколько вариантов развития:

`dbrs = [(cnt`$_1$`, c`$_1$`), ..., (cnt`$_n$`, c`$_n$`)]`

то в этом случае, прежде всего в функции `procDR` (см. второе предложение на рисунке 4.7) выполняется проверка возможного зацикливания—вычисляется `(isWh c h)`.

Если в истории `h` вершины `ni` не нашлось конфигураций опасно близких к конфигурации `c`:

`(isWh c h) == [ ]`,

то, при помощи функции `mkDbr` (см. раздел 4.4.4 строится вершина для `ni` с ветвями продолжения, соответствующими списку `dbrs`:

`(DRIVE k c brs)`,
`brs = [((cnt`$_1$`,gr`$_1$`), ..., (cnt`$_n$`,gr`$_n$`))]`

где `gr`$_i$—графы конфигураций, построенные для `c`$_i$.

Если в истории `h` вершины `ni` нашлась конфигурация `cUp` (с номером `kUp`) опасно близкая к конфигурации `c`:

`(isWh c h) == [(kUp,cUp)]`,

то порождают вершину с отладочной информацией и "свистке"

`(WHISTLE k kUp gr)`,

а собственно S-граф `gr` для вершины `ni` определяется функцией `procWh` (см. раздел 4.4.5):

```
procWh ni kUp cUp p i
```

Замечание о раличных стратегиях обработки транзитных вершин. Выше было сказано, что в нашем проекте суперкомпилятора (как и в большинстве суперкомпиляторов) транзитные вершины обрабатываются (см. второе предложение на рисунке 4.7) *"без проверки на зацикливание, без порождения транзитной вершины (update in place) и наращивания истории"*.

В общем случае, существует три осмысленные стратегии обработки транзитных вершин, которые можно описать тремя раличными состояниями двух булевских переменных (флагов) `trIsSpecCase` и `trInPlace`:

1. *Транзитные вершины не надо проверять на зацикливание, их не надо включать в историю развития вершины (не стоит в дальнейшем искать циклы на транзитные), а значит, они не нужны в графе конфигураций—вполне допустим "update in place"* (вариант, приведеный на рисунке 4.7):

```
trIsSpecCase = True
trInPlace = True
```

2. *Транзитные вершины не надо проверять на зацикливание, однако возможны циклы на транзитные вершины—их надо включать в историю (а значит и в граф конфигураций)*:

```
trIsSpecCase = True
trInPlace = False
```

3. *Не следует ничем отличать анализ транзитных переходов от анализа многовариантных переходов*:

```
trIsSpecCase = False
```

Вариант функции procDR, поддерживающий все три стратегии приведен на рисунке 4.8.

```
-- Имените эти две строки, для выбора стратегии обработки
-- транзитных вершин:

trIsSpecCase = True   -- или False
trInPlace    = True   -- или False

procDR ::Nodeinfo->[DBranch]->ProgR->FreeIndx -> (SGraph, FreeIndx)

procDR ni@(k,c,h) dbrs@[ ] p i       = ((CALLC "Z" k c 0 [] []),i)

procDR ni@(k,c,h) dbrs@[(cn,c')] p i
                     | trIsSpecCase = if trInPlace
                                      then (evalG (k,c',h) p i)
                                      else (mkDbr ni dbrs p i)

procDR ni@(k,c,h) dbrs p i =
    case (isWh c h) of
       [ ]         -> mkDbr ni dbrs p i
       [(kUp,cUp)] -> ((WHISTLE k kUp gr), i')
                      where (gr, i') = procWh ni kUp cUp p i
```

Рис. 4.8: Вариант функции procDR, поддерживающий различные стратегии работы с транзитными вершинами

4.4.4 Функция mkDbr. Построение ветвей продолжений

Пусть

1. ni = (k, c, h)—непассивная вершина, k—номер этой вершины, h—ее история, c—конфигурация;

2. для вершины ni прогонкой построено несколько вариантов развития

   dbrs = [(cnt$_1$, c$_1$), ..., (cnt$_n$, c$_n$)]

   где cnt$_i$—альтернативные сужения (объединение всех <c/.cnt$_i$> совпадает с <c>), c$_i$—конфигурации, поученные выполнением одного шага обобщенного вычисления конфигурации c/.cnt$_i$.

Функция mkDbr строит для ni вершину с ветвями продолжения, соответствующими списку dbrs:

(DRIVE k c brs),
brs = [((cnt$_1$,gr$_1$), ..., (cnt$_n$,gr$_n$))]

где gr$_i$—графы конфигураций, построенные для c$_i$ при помощи функции evalG:

(gr$_i$, _) = evalG (i, c$_i$, h') p i''

где ni$_i$ = (i, c$_i$, h')—информация о вершине с конфигурацией c$_i$; i—уникальный номер для вершины ni$_i$, определяемый в функции mkDbr различными значениями свободного индекса c-переменных; h'—история вершины ni$_i$.

Обратим внимание, что все вершины ni$_i$—непосредственные сыновья ni, и история h' у них всех одинаковая—это история h самой вершины ni, в которую добавлен номер k и конфигурация c из ni:

h' = (k,c):h

```
mkDbr ::Nodeinfo->[DBranch]->ProgR->FreeIndx -> (SGraph, FreeIndx)

mkDbr ni@(k,c,h) dbrs p i = ((DRIVE k c brs), i')
      where
         h' = (k,c):h
         (brs,i') = mkBrs dbrs i
         mkBrs :: [DBranch]->FreeIndx  -> (SBranches,FreeIndx)
         mkBrs [ ]              i = ([ ],i)
         mkBrs ((cn,c'):dbrs)  i = ((cn,gr):brs, i2)
                                   where
                                     (gr, i1) = evalG (i,c',h') p (i+1)
                                     (brs,i2) = mkBrs dbrs i1
```

Рис. 4.9: Функция mkDbr. Построение ветвей продолжений

4.4.5 Функция `procWh`. Обработка возможного зацикливания

Пусть

1. `ni = (kDn, cDn, hDn)`—непассивная вершина, `kDn`—номер этой вершины, `hDn`—ее история, `cDn`—конфигурация;

2. в истории `hDn` вершины `ni` нашлась конфигурация `cUp` (с номером `kUp`) опасно близкая к конфигурации `cDn`:

   ```
   (isWh cDn hDn) == [(kUp,cUp)].
   ```

Функция `procWh` определяет граф `gr` конфигураций для вершины `ni` следующим образом (см. рисунок 4.10):

1. Выполняется обобщение (см. раздел 3.5) конфигураций `cDn` и `cUp`:

   ```
   (gTab, cGen, i1) = genConf cDn cUp i
   (sGenDn, sGenUp) = genTabToSubsts gTab
   ```

2. По результататам обобщения (см. раздел 3.6) проверяется условие `(cUp` $\succeq$ `cDn)`:

   ```
   (isEqCUpCGen cUp cGen sGenUp)
   ```

   (a) Если условие `(cUp` $\succeq$ `cDn)` выполнено, то вершина `ni` может быть сведена к вершине-предку `(kUp, cUp, _)`. В качестве графа для вершины `ni` в этом случае строится вершина Up-сведения:

   ```
   gr = (CALLC "Up"kDn cDn kUp (mkParms cUp sUpDn) [ ])
   ```

   где `sUpDn = (mkSubstUpDn sGenUp sGenDn)`—подстановка приводящая конфигурацию `cUp` к `cDn` (см. раздел 3.6.1).

   (b) Если условие `(cUp` $\succeq$ `cDn)` не выполнено, то вершину `ni` необходимо сводить к новой вершине `(_, cGen, _)`, так как развитие вершины `ni` связано с опасностью зацикливания. Сведение `ni` к `(_, cGen, _)` всегда возможно, так как по построению `(cGen` $\succeq$ `cDn)`.

   В качестве графа для вершины `ni` в этом случае строится вершина Dn-сведения:

   ```
   gr = (CALLC "Dn"kDn cDn i1 (mkParms cGen sGenDn) [grGen])
   ```

   где

   - `grGen`—граф конфигураций, для новой вершины `nGen`:

     ```
     nGen = (i1, cGen, hGen)
     (grGen,i') = evalG (i1, cGen, hGen) p (i1+1)
     ```

   - `i1`—уникальный номер вершины `nGen` (определяется при помощи свободного индекса с-переменных);

   - `hGen`—история вершины `nGen`: эта вершина—непосредственный сын `ni` и история `hGen`—это история `hDn` самой вершины `ni`, в которую добавлен номер `kDn` и конфигурация `cDn` из `ni`:

     ```
     hGen = ((kDn, cDn):hDn)
     ```

```
procWh ::Nodeinfo->Int->Conf->ProgR->FreeIndx -> (SGraph, FreeIndx)

procWh ni@(kDn,cDn,hDn) kUp cUp p i =
   if   (isEqCUpCGen cUp cGen sGenUp)    -- вкладывается cDn в cUp ?
   then ( (CALLC "Up" kDn cDn kUp (mkParms cUp sUpDn)  [    ]),  i1)
   else ( (CALLC "Dn" kDn cDn i1 (mkParms cGen sGenDn) [grGen]), i')
   where
      (gTab, cGen, i1) = genConf cDn cUp i
      (sGenDn, sGenUp) = genTabToSubsts gTab
      sUpDn = mkSubstUpDn sGenUp sGenDn
      hGen = ((kDn, cDn):hDn)
      (grGen,i') = evalG (i1, cGen, hGen) p (i1+1)
```

Рис. 4.10: Функция `procWh`. Обработка возможного зацикливания

4.5 Функции вывода результата суперкомпиляции

В модуле `scp.gs` присутствуют описание двух вспомогательных функций, предназначенных для преобразования графа конфигураций в программу на языке TSG:

- Функция `extrProg :: SGraph -> ProgR` — преобразует граф конфигураций в программу на языке TSG.

  Преобразование написано по принципу: каждая вершина определяет функцию. Поэтому, в получаемой (при помощи `extrProg`) программе очень много функций, причем большинство из них вызываются по одному разу в тексте программы (функции, соответствующие небазисным конфигурациям).

- Функция `optimize :: ProgR -> ProgR`—простое преобразование программы: если функция вызывается в программе один раз, то ее можно удалить из програмы, подставив вместо вызова этой функции ее тело (с заменой аргументов на параметры вызова).

  Результатом применения функции `optimize` к TSG-программе, получаемой при помощи функции `extrProg` является программа, в которой каждая функция соответствует базисной конфигурации.

Так как эти функции носят вспомогательный характер, их описание не анализируется в данной работе.

Все примеры суперкомпиляции, рассматриваемые в главе `s:Samples` посчитаны по следующей схеме:

```
p' = optimize (extrProg (scp p cl)))
```

где `p`—суперкомпилируемая программа, `cl`—класс, обобщенное данное для `p`, `p'`—остаточная программа, результат суперкомпиляции.

4.6 Выводы

В главе 4 введен синтаксис представления графа конфигураций (S-графа) для TSG, рассмотрена операция сведения двух конфигураций в S-графе, определены вспомогательные синтаксические конструкции и функции для суперкомпилятора.

Затем, определен и обсужден собственно суперкомпилятор для языка TSG, основанный на идее выполнения обобщения конфигураций в нижней конфигурации (в ситуации *возможное зацикливание*). Данный суперкомпилятор использует алгоритмы `Wh` и `Gener` (определенные в главах 2 и 3) и алгоритмы, функции и операции из [3].

В завершении главы 4 рассмотрены вспомогательные функции, предназначенные для вывода результата суперкомпиляции (графа конфигураций) в читабельном виде (в виде TSG-программы).

Глава 5

Примеры суперкомпиляции

5.1 Специализация программ

Пусть `p [x,y]`—программа на `R` от двух аргументов. Зафиксируем один аргумент, например `x`, положив его равным `A`. Полученную функцию $\texttt{p}_{[A,\_]}$ одного аргумента:

$$\texttt{p}_{[A,\_]}\ \texttt{[y] = p [A,y]},$$

будем называть проекцией `p` на `[A,_]`.

Аналогичным образом определяется проекция $\texttt{p}_{[\_,B]}$ для случая фиксации второго аргумента функции:

$$\texttt{p}_{[\_,B]}\ \texttt{[x] = p [x,B]},$$

и проекции по нескольким аргументам программы с произвольной арностью.

Определение 1
Специализатором для языка `R` называют программу `spec`, которая

1. по тексту программы `p`∈`R` и
2. по некоторому описанию проекции—что включает определение позиций в списке аргументов `p`, подлежащих фиксации, и задание значений фиксируемых аргументов;

выполняет *специализацию* программы `p`—строит эффективную программу `p'`, реализующую соответствующую проекцию функции `p`.

Сегодня несколько технологий позволяют выполнять на практике специализацию программ. Среди них наиболее успешно справляются с данной задачей:

- "Scp"—суперкомпиляция—один из методов метавычислений и
- "Pe"—частичное выполнение (смешанные вычисления).

5.1.1 Суперкомпиляция как метод специализации программ

Суперкомпиляция—один из разделов метавычислений,—совокупность понятий, методов и алгоритмов метавычислений, позволяющих выполнять глубокие преобразования программ на языке реализации R.

Аргументами суперкомпилятора являются: p∈R—программа на языке реализации и cl—класс, обобщенные данные для p.

Суперкомпилятор построен по следующей схеме:

1. Выполняется построение перфектного дерева процессов tree программы p на классе cl, так как это описано [3];

2. Процессы развития бесконечных ветвей дерева tree гарантированно *приостанавливаются* в некоторой вершине (точка обнаружения *возможного* зацикливания).

   Это выполняется при помощи той или иной реализации алгоритма wh обнаружения *возможного* зацикливания вычислительного процесса. работы.

   При приостановке процесса развития ветви, алгоритм wh указывает две вершины на данной ветви:

   - $\mathtt{n}_{dn}$—нижняя вершина ветви—точка обнаружения *возможного* зацикливания;
   - $\mathtt{n}_{up}$—одна из вершин на той же ветви, что и $\mathtt{n}_{dn}$;

   Эти две вершины характеризуются тем, что их конфигурации $\mathtt{c}_{up}$ и $\mathtt{c}_{dn}$ "опасно похожи", что и является причиной *возможного* бесконечного развития данной ветви.

3. Выполняется анализ "приостановленной ветви" дерева.

   (a) Если $\mathtt{c}_{dn} \preceq \mathtt{c}_{up}$, то восстанавливается сужение, приводящее $\mathtt{c}_{up}$ к $\mathtt{c}_{dn}$:

   $$\mathtt{c}_{up}\mathtt{/.(S\ s)/.(R\ r)=c}_{dn}$$

   и оформляется *цикл*—ребро $(\mathtt{c}_{dn} \xrightarrow{s} \mathtt{c}_{up})$ от нижней вершины $\mathtt{n}_{dn}$ к верхней $\mathtt{n}_{up}$, с приписанной данному ребру подстановкой s. На этом развитие данной ветви завершается.

   (b) Если условие $\mathtt{c}_{dn} \preceq \mathtt{c}_{up}$ не выполнено, то выполняется *обобщение конфигураций* $\mathtt{c}_{dn}$ и $\mathtt{c}_{up}$—построение при помощи некоторого алгоритма gener общей для $\mathtt{c}_{dn}$ и $\mathtt{c}_{up}$ надконфигурации $\mathtt{c}_{gen}$: $\mathtt{c}_{dn} \preceq \mathtt{c}_{gen}$, $\mathtt{c}_{up} \preceq \mathtt{c}_{gen}$.

   После этого применяется *нижняя перестройка к вершине* $\mathtt{n}_{dn}$: ветвь удлиняется на одно ребро и одну вершину: $(\mathtt{c}_{dn} \xrightarrow{s} \mathtt{c}_{gen})$, где s—подстановка из сужения, приводящего $\mathtt{c}_{gen}$ к $\mathtt{c}_{dn}$.

   После чего с конфигурации $\mathtt{c}_{gen}$ возобновляется процесс развития данной ветви (конечно же, под контролем алгоритма wh).

В результате описанных выше действий суперкомпилятор scp "сворачивает" потенциально бесконечное перфектное дерево процессов в *гарантированно* конечный граф конфигураций:

$$\mathtt{scp\ p\ cl} \overset{*}{\Rightarrow} \mathtt{graph}$$

представляющий все процессы вычисления p на классе cl:

$\mathcal{P}$(p,$\mathcal{C}$)⊆<graph>

Таким образом, граф описывает вычислительное поведение p на <cl>.

Данный граф конфигураций является программой[1] graph=p_{cl} с параметрами

[v_1,...,v_n] = cvars cl = $\mathcal{X}$

и со следующим свойством:

Пусть cl=(ces,r)=(e($\mathcal{X}$),r).

Рассмотрим произвольное данное d∈<cl>.

Пусть s∈FVS(cl)*—подстановка, такая, что* cl/.s=(d,RESTR[]). *То есть:*

$s=[v_i:\text{->}d_i,\ldots,v_n:\text{->}d_n]=(\mathcal{X}:\text{->}\vec{d})\in FVS(cl)$
$d=ces/.s=e(\mathcal{X})/.(\mathcal{X}:\text{->}\vec{d})=e(\vec{d})$

Тогда выполнено:

$$p_{cl}\ [d_1,\ldots,d_n] = p\ d = p\ e([d_1,\ldots,d_n]) \tag{5.1}$$

или, в неформальной векторной нотации:

$$p_{cl}\ \vec{d} = p\ e(\vec{d}) \tag{5.2}$$

Таким образом, суперкомпилятор вычисляет программу p_{cl}, которая является результатом *специализации p на класс* cl, то есть специальной версией программы p для случая, когда входные данные имеют вид d=e($\vec{d}$)∈<cl>. При этом выполняются глубокие преобразования исходной программы p.

Заметим, что *при помощи суперкомпиляции можно выполнить любую классическую* (введенную определением 1) *специализацию.*

Например, пусть p [x,y]—программа на языке R от двух аргументов, которую надо специализировать: вычислить проекцию на [A,_], где A∈EVal—некоторая константа. Определим класс cl следующим образом: cl=([A,$\mathcal{E}$.i], RESTR[]) и выполним суперкомпиляцию:

scp p cl = p_{cl}

В силу основного свойства (см. отношения 5.1 и 5.2) суперкомпилятора, для любого x∈EVal будет выполнено:

p_{cl} [x] = p [A,x] = $p_{[A,\_]}$ [x]

Таким образом, при помощи суперкомпиляции выполнена классическая специализация: построена программа p_{cl}, реализующая функцию $p_{[A,\_]}$.

Данный пример легко обобщается на любое описание заказа mask на классическую специализацию: чтобы выполнить при помощи суперкомпилятора проекцию программы p [x_1,...,x_n] на

[1] Точнее—некоторой формой записи программы [9, 10, 15, 16, 19]. Эта запись настолько близка к языкам S-Graph [19] и TSG, что graph легко может быть преобразован в программу на TSG.

mask=[mask$_1$,...,mask$_n$],
где mask$_i$—либо константа—mask$_i$∈EVal, либо "прочерк"—mask$_i$="_",

надо выполнить суперкомпиляцию scp p cl$_{mask}$, где

cl$_{mask}$=(ces, RESTR[]),
ces=[ces$_1$,...,ces$_n$],
ces$_i$=mask$_i$, если mask$_i$—константа, mask$_i$∈EVal и
ces$_i$—се-переменная, если mask$_i$="_", причем, все се-переменные в ces—различные.

Другими словами, ces—это mask, в которой все "прочерки" заменены на различные се-переменные.

Заметим, что суперкомпилятор позволяет строить специальные версии программ, выходящие за рамки классического определения специализации. Рассмотрим самый простой пример. Пусть p [x,y]—программа на языке R от двух аргументов. Рассмотрим класс cl=([$\mathcal{E}$.1,$\mathcal{E}$.1], RESTR[]), и выполним суперкомпиляцию:

scp p cl = p$_{cl}$

В силу основного свойства (см. отношения 5.1 и 5.2) суперкомпилятора, для любого x∈EVal будет выполнено:

p$_{cl}$ [x] = p [x,x]

Таким образом, при помощи суперкомпиляции выполнена *неклассическая* "диагональная" специализация: построена программа p$_{cl}$, реализующая функцию f[x] = p[x,x].

Выполняемая суперкомпилятором специализация программы p на класс cl, введенная определением 1, является понятием более широким, чем классическая специализация.

Замечание Существенно следующее свойство теории суперкомпиляции: в ней часто приходится иметь дело с алгоритмически неразрешимыми задачами и в этом случае довольствоваться алгоритмическими "приближенными" решениями данных задач. Один из примеров такой ситуации—алгоритм wh распознавания *возможного* зацикливания вычислительного процесса. Да и в целом, цель суперкомпиляции—построение перфектного графа конфигураций—алгоритмически неразрешимая проблема [19]. Поэтому на каждом этапе развития теории суперкомпиляции имеют дело с приближенным алгоритмическим решением данной проблемы—суперкомпилятор выполняет построение "почти перфектного" графа конфигураций. А прогресс в теории суперкомпиляции—это получение все более точных приближений к перфектному графу.

Следствием данной ситуации является то, что данный прогресс не имеет формальных ограничений.

Совершенствовать методы суперкомпиляции можно беспредельно. Суперкомпиляция и метавычисления в целом, как область научных исследований, является неограниченой областью.■

5.2 Специализация при помощи суперкомпилятора. От pmatch к KMP-алгоритму

Известно, что сложность алгоритма наивного поиска подстроки в строке $O(N \times M)$, где N—длина строки, M—длина подстроки, а сложность KMP- алгоритма—$O(M)$. Почти все существующие сегодня суперкомпиляторы могут заменять наивный поиск подстроки в строке на KMP-алгоритм. Рассмотрим примеры построения KMP-алгоритма суперкомпилятором, представленным в этой работе.

5.2.1 Пример 1. Поиск подстроки “AAB” в произвольной строке

Программа поиска подстроки “AB” в произвольной строке написана на языке `TSG`, ее аргументом является строка `E1`. Программа строит две функции: `"F2"`, которая находит атом в строке, и `"F11"`, которая производит сравнение найденного атома с нужным символом подстроки.

Функция `"F2".` В первой строке функции проверяется, имеет ли переменная `E1` вид:

`E1 = E3:E4`,

- если нет, то `E1 = A5`, и проверка неуспешна, подстрока не найдена;
- если да, то происходит проверка, имеет ли переменная `E3` вид `E7:E8`, и `E1` принимает вид `E1 = (E7:E8):E4`,
- если `E3 = E7:E8`, то подстрока не найдена.
- если эта проверка неуспешна, то это означает, что `E1` имеет вид `E1 = A9:E4`, и происходит вызов функции `F11[A9, E4]`:

  ```
  (call "F11"[ A9, E4]).
  ```

  Это означает, что найден какой-то символ, и нужно проверить, равен ли он первому символу искомой подстроки. Это делает следующая функция.

Функция `"F11"` В тот момент, когда начинает работу функция `F11`, известно, что строка `E1` имеет вид:

`E1 = A9:E4`.

Происходит проверка, равен ли найденный атом `A9` атому `A`.

- Если проверка не успешна, то есть `A9 /= 'A`, то происходит вызов функции `"F2"[ E4 ]`. Это означает, что теперь будет происходить поиск подстроки “AB”, начиная с `E4`.
- Если проверка оказалась успешной, `A9 = 'A`, то становится известно, что

  `E1 = 'A:E4`,

```
(define "F2"[ E1]=
   (alt (cons?  E1 E3 E4 A5)
   -- E1 = E3:E4
      (alt (cons?  E3 E7 E8 A9)
      -- E1 = (E7:E8):E4
         'FAILURE
      -- E1 = A9:E4
         (call "F11"[ A9, E4])
      )
   -- E1 = A5
      'FAILURE
)  )
```

Рис. 5.1: KMP для строки "AB". Функция "F2".

и начинается разбор E4.

Функция проверяет, имеет ли E4 вид: E4 = E13:E14 или E4 = A15.

- Если E4 = A15, то E1 = 'A:A15, неуспех.
- Если E4 = E13:E14, и E1 имеет вид: E1 = 'A:E13:E14, то происходит разбор E13.
- Если E13 = E17:E18, и E1 = 'A:E17:E18:E14, то подстрока не найдена.
- Если проверка не успешна, то E1 имеет вид: E1 = 'A:'A19:E14, и нужно проверить, равен ли атом A19 атому B.
- Если A19 = 'B, то подстрока найдена.
- Если A19 /= 'B, то происходит вызов функции F11 [A19, E14]. Это означает, что программа будет искать подстроку "AB", начиная с символа A19.

5.2.2 Пример 2. Поиск подстроки "AABAAC" в произвольной строке

Работу начинает функция "F2"[E1]. Сначала функция проверяет, какой вид имеет строка E1.

- Если E1 = A5, то подстрока не найдена.
- Если E1 = E3:E4, то происходит проверка E3.
- Если E3 = E7:E8, то подстрока не найдена.
- Если E3 = A9, и E1 = A9:E4, то проверяется, равен ли атом A9 атому A.
- Если A9 /= 'A, и E1 = A9:E4, то происходит вызов функции "F2"[E4].

```
(define "F11"[ A9, E4]=
-- E1 = A9:E4
   (alt (eqa?  A9 'A)
   -- E1 = 'A:E4
      (alt (cons?  E4 E13 E14 A15)
      -- E1 = 'A:E13:E14
         (alt (cons?  E13 E17 E18 A19)
         -- E1 = 'A:(E17:E18):E14
            'FAILURE
         -- E1 = 'A:A19:E14
            (alt (eqa?  A19 'B)
            -- E1 = 'A:'B:E14
               'SUCCESS
            -- E1 = 'A:A19:E14, A19 /= 'B
               (call "F11"[ A19, E14])
         )  )
      -- E1 = 'A:A15
         'FAILURE
      )
   -- E1 = A9:E4, A9 /= 'A
      (call "F2"[ E4])
)  )
```

Рис. 5.2: KMP для строки `"AB"`, функция `"F11"`

- Если `A9 = 'A`, то `E1 = 'A:E4`, и начинается разбор `E4`.
- Если `E4 = A15`, то подстрока не найдена.
- Если `E4 = E13:E14`, а `E1 = 'A:E13:E14`, то происходит разбор `E13`.
- Если `E13 = E17:E18`, а `E1 = 'A:E17:E18:E14`, то подстрока не найдена.
- Если `E13 = A19`, то `E1 = 'A:A19:E14`, и происходит вызов функции

  `"F21"[ A19, E14 ]`.

Функция `"F21"` В момент начала работы функции `"F21"` известно, что `E1` имеет вид:

`E1 = 'A:A19:E14`.

Проверяется равенство атома `A19` атому `A`.

- Если `A19 /= 'A`, и `E1 = 'A:A19:E14`, то вызывается функция `"F2"[ E14 ]`.
- Если `A19 = 'A`, и `E1 = 'A:'A:E14`, то происходит разбор `E14`.
- Если `E14 = A25`, то есть `E1 = 'A:'A:A25`, то подстрока не найдена.

```
(define "F2"[ E1]=
   (alt (cons?  E1 E3 E4 A5)
   -- E1 = E3:E4
      (alt (cons?  E3 E7 E8 A9)
      -- E1 = (E7:E8):E4
         'FAILURE
      -- E1 = A9:E4
         (alt (eqa?  A9 'A)
         -- E1 = 'A:E4
            (alt (cons?  E4 E13 E14 A15)
            -- E1 = 'A:(E13:E14)
               (alt (cons?  E13 E17 E18 A19)
               -- E1 = 'A:((E17:E18):E14)
                  'FAILURE
               -- E1 = 'A:(A19:E14)
                  (call "F21"[ A19, E14])    )
            -- E1 = 'A:A15
               'FAILURE    )
         -- E1 = A9:E4, A9 /= 'A
            (call "F2"[ E4])  )  )
   -- E1 = A5
      'FAILURE
)  )
```

Рис. 5.3: КМР для строки `"AABAAC"`. Функция `"F2"`

- Если `E14 = E23:E24`, `E1 = 'A:'A:E23:E24`, то разбирается `E23`.
- Если `E23 = E27:E28`, `E1 = 'A:'A:E27:E28:E24`, то подстрока не найдена.
- Если `E23 = A29`, то `E1 = 'A:'A:A29:E24`, и происходит вызов функции

  `"F31"[ A29, E24 ]`.

Функция `"F31"` Когда вызвана функция `"F31"`, `E1` имеет следующий вид:

`E1 = 'A:('A:(A29:E24))`.

Происходит проверка, равен ли атом `A29` атому `B`.

- Если `A29 /= 'B`, то вызываеся функция `"F21"[ A29, E24 ]`:

  `(call "F21"[ A29, E24])`.

- Если `A29 = 'B`, и `E1` принимает вид:

  `E1 = 'A:('A:('B:E24))`,

```
(define "F21"[ A19, E14]=
-- E1 = 'A:(A19:E14)
   (alt (eqa?  A19 'A)
   -- E1 = 'A:('A:E14)
      (alt (cons?  E14 E23 E24 A25)
      -- E1 = 'A:('A:(E23:E24))
         (alt (cons?  E23 E27 E28 A29)
         -- E1 = 'A:('A:((E27:E28):E24))
            'FAILURE
         -- E1 = 'A:('A:(A29:E24))
            (call "F31"[ A29, E24])
         )
      -- E1 = 'A:('A:A25)
         'FAILURE
      )
   -- E1 = 'A:(A19:E14), A19 /= 'A
      (call "F2"[ E14])
)  )
```

Рис. 5.4: KMP для строки "AABAAC". Функция "F21"

то начинается разбор E24.

- Если E24 = A35, то E1 = 'A:('A:('B:A35)), и подстрока не найдена.
- Если E24 = E33:E34, E1 = 'A:('A:('B:(E33:E34))), то происходит разбор E33.
- Когда E33 = E37:E38, а E1 = 'A:('A:('B:((E37:E38):E34))), то подстрока не найдена.
- Когда E33 = A39, а E1 = 'A:('A:('B:(A39:E34))), происходит проверка равенства атомов A39 и A.
- Если A39 /= 'A, и E1 = 'A:('A:('B:(A39:E34))), то происходит вызов функции "F2"[E34]:

  (call "F2"[E34]).

- Если A39 = 'A, E1 = 'A:('A:('B:('A:E34))), то начинается разбор E34.
- Если E34 = A45, E1 = 'A:('A:('B:('A:A45))), то подстрока не найдена.
- Если E34 = E43:E44, E1 = 'A:('A:('B:('A:(E43:E44)))), то рассматриватеся E43.
- Если E43 = E47:E48, и E1 = 'A:('A:('B:('A:((E47:E48):E44)))), то подстрока не найдена.

- Если `E43 = A49`, и `E1 = 'A:('A:('B:('A:(A49:E44))))`, то происходит проверка равенства атомов `A49` и `A`.
- Если `A49 /= 'A`, и `E1 = 'A:('A:('B:('A:(A49:E44))))`, то происходит вызов функции `"F2"[ E44 ]`:

  ```
  (call "F2"[ E44]).
  ```

- Если `A49 = 'A`, `E1 = 'A:('A:('B:('A:('A:E44))))`, то происходит разбор `E44`.
- Если `E44 = A55`, `E1 = 'A:('A:('B:('A:('A:A55))))`, то подстрока не найдена.
- Если `E44 = E53:E54`, и `E1 = 'A:('A:('B:('A:('A:(E53:E54)))))`, то разбирается `E53`.
- Если `E53 = E57:E58`, и `E1 = 'A:('A:('B:('A:('A:((E57:E58):E54)))))`, то подстрока не найдена.
- Если `E53 = A59`, и `E1='A:('A:('B:('A:('A:(A59:E54)))))`, то проверяется равенство атомов `A59` и `'C`.
- Если `A59 = 'C`, `E1 = 'A:('A:('B:('A:('A:('C:E54)))))`, то подстрока найдена.
- Если `A59 /= 'C`, и `E1='A:('A:('B:('A:('A:(A59:E54)))))`, то вызывается функция `"F31"[ A59, E54 ]`:

  ```
  (call "F31"[ A59, E54])
  ```

  Это означает, что далее будет происходить поиск подстроки, начиная с символа `A59`.

5.2.3 Выводы

В разделе 5.2 рассмотрены примеры суперкомпиляции алгоритма `pmatch` на классах вида `([`*string*`, `$\mathcal{E}$`.1), [ ])`, где *string*—некоторая константная строка. Рассмотренные примеры позволяют сделать следующий вывод: реализованный по проекту суперкомпилятор способен выполнять нетривиальную специализацию программ. В частности, программа наивного алгоритма проверки вхождения подстроки в строку при специализации по первой подстроке им успешно преобразуется в алгоритм Кнута-Мориса-Пратта.

5.3 Проекции Футамуры-Турчина

5.3.1 Определение проекций Футамуры-Турчина

Предположим, что `s` записан на языке `R`. Это дает возможность применять `s` к самому себе, что позволяет выполнять цепочку специализаций (метасистемных переходов) и автоматически строить по заданным программам новые, реализующие весьма нетривиальные функции.

```
(define "F31"[ A29, E24]=
-- E1 = 'A:('A:(A29:E24))
   (alt (eqa?  A29 'B)
   -- E1 = 'A:('A:('B:E24))
      (alt (cons?  E24 E33 E34 A35)
      -- E1 = 'A:('A:('B:(E33:E34)))
         (alt (cons?  E33 E37 E38 A39)
            'FAILURE
         -- E1 = 'A:('A:('B:(A39:E34)))
            (alt (eqa?  A39 'A)
            -- E1 = 'A:('A:('B:('A:E34)))
               (alt (cons?  E34 E43 E44 A45)
               -- E1 = 'A:('A:('B:('A:(E43:E44))))
                  (alt (cons?  E43 E47 E48 A49)
                     'FAILURE
                  -- E1 = 'A:('A:('B:('A:(A49:E44))))
                     (alt (eqa?  A49 'A)
                     -- E1 = 'A:('A:('B:('A:('A:E44))))
                        (alt (cons?  E44 E53 E54 A55)
                        -- E1 = 'A:('A:('B:('A:('A:(E53:E54)))))
                           (alt (cons?  E53 E57 E58 A59)
                              'FAILURE
                           -- E1 = 'A:('A:('B:('A:('A:(A59:E54)))))
                                 (alt (eqa?  A59 'C)
                                 -- E1 = 'A:('A:('B:('A:('A:('C:E54)))))
                                    'SUCCESS
                                 -- E1 = 'A:('A:('B:('A:('A:(A59:E54))))),
                                 -- A59 /= 'C
                                    (call "F31"[ A59, E54])    )     )
                           'FAILURE    )
                     -- E1 = 'A:('A:('B:('A:(A49:E44)))), A49 /= 'A
                        (call "F2"[ E44])    )  )
                  'FAILURE    )
            -- E1 = 'A:('A:('B:(A39:E34))), A39 /= 'A
               (call "F2"[ E34])    )  )
         'FAILURE    )
   -- E1 = 'A:('A:(A29:E24)), A29 /= 'B
      (call "F21"[ A29, E24])
)  )
```

Рис. 5.5: KMP для строки "AABAAC". Функция "F31"

Рассмотрим знаменитый пример [5]. Пусть p—программа на языке L, d—данные для p, p [d] $\overset{*}{\underset{\mathtt{L}}{\Rightarrow}}$ res, intL—интерпретатор языка L, написанный на R. Используя несколько раз основное свойство специализатора, получим, что для любой p∈L и любых d выпол-

нено:

```
res = intL [p,d]
    = s[intL^SD,p] [d]
    = s[s^SD,intL^SD] [p] [d]
    = s[s^SD,s^SD] [intL^SD][p] [d]
```

Построены три новые программы на языке R:

```
pR = s[intL^SD,p]
compL = s[s^SD,intL^SD]
gencomp = s[s^SD,s^SD]
```

Основываясь на функциональных свойствах данных программ, можно дать им привычное для теории программирования наименование:

1. Программа pR=s[intL^{SD},p] на языке R обладает тем свойством, что для любого d выполнено pR [d] = intL [p,d] = p 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 compL=s[s^{SD},intL^{SD}] обладает тем свойством, что для любой p∈L выполнено compL [p]=pR.

   То есть, compL—*компилятор с языка L на язык R*.

3. Программа gencomp=s[s^{SD},s^{SD}] обладает тем свойством, что для любого языка L, для которого представлено "его описание" в виде интерпретатора intL, выполнено gencomp[intL^{SD}]=compL.

   То есть, gencom 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 повторено построение трех знаменитых проекций Футамуры-Турчина, первые упоминания о которых в публикациях относятся к 70-ым годам [5].

5.3.2 Компиляция конечных автоматов auto1 и auto2 в TSG

На рисунках 5.6, 5.7, 5.8 приведен интерпретатор языка конечных автоматов.

В данном языке программы—конечные автоматы—имеют следующий синтаксис:

automaton ::= 'NIL | (CONS *state-def automaton*)

state-def ::= (CONS *state-name def*)

def ::= 'NIL | (CONS *char-def def*)

char-def ::= (CONS *char state-name*)

state-name ::= *atom*

char ::= *atom*

```
prog_auto :: ProgR
prog_auto =
   [  (DEFINE "auto" [auto,x]
         (ALT (CONS' auto state_def tauto a)
         -- auto=(state_def:tauto)
            (CALL "loop" [state_def, x, auto])
         -- auto=a
            (error_in_auto auto)
         )
      ),
      (DEFINE "loop" [state_def, x, auto]
         (ALT (CONS' state_def state def a)
         -- state_def=(state:def)
            (ALT (CONS' x hx tx ax)
            -- x=(hx:tx)
               (ALT (CONS' hx e1 e2 ax)
               -- x=((e1:e2):tx)
                  (error_in_data x)
               -- x=(ax:tx), state_def:(state:def)
                  (CALL "NewState"[ax,def,tx,auto])
               )
            -- x=ax -- The End
               state
            )
         -- state_def=a
            (error_in_auto state_def)
         )
      ),
```

Рис. 5.6: Интерпретатор языка конечных автоматов, функции `auto` и `loop`

```
(DEFINE "NewState"[ax,def,x,auto]
   (ALT (CONS' def chardef tdef a)
   -- def=(chardef:tdef)
      (ALT (CONS' chardef char statename a)
      -- chardef=(char:statename)
         (ALT (CONS' char e1 e2 achar)
         -- chardef=((e1:e2):state)
            (error_in_auto chardef)
         -- chardef=(achar:statename)
            (ALT (EQA' ax achar)
            -- chardef=(ax:statename)
               (ALT (CONS' statename e1 e2 astatename)
                  -- chardef=(ax:(e1:e2))
                  (error_in_auto statename)
                  -- chardef=(ax:astatename)
                  (CALL "NewStateDef"[astatename,auto,x,auto])
               )
            -- chardef=(achar:statename), achar /= ax
               (CALL "NewState"[ax,tdef,x,auto])
            )
         )
      -- chardef=a
         (error_in_auto chardef)
      )
   -- def=a -- определение переходов кончилось, но ax не найден
      (error_in_data ax)
   )
),
```

Рис. 5.7: Интерпретатор языка конечных автоматов, функция `NewState`

Первое определение *state-def* в записи конечного автомата задает начальное состояние автомата. В интерпретаторе по текущему состоянию и входной строке выполняется обычный порядок интерпретации конечного автомата:

1. Если входная строка `x` исчерпана, то в качестве результата выдается имя текущего состояния `state` (функция `"loop"`, рисунок 5.6).

2. Если от входной строки можно отделить входной символ `ax`— `x = ax:tx`,—то выполняются следующие действия:

   (a) Для текущего состояния `(CONS` *`state-name def`*`)` в определениях *def* ищется пара вида `(CONS ax statename)`, где `ax`—первый символ в `x`, `statename`—имя нового состояния автомата (функция `"NewState"`, рисунок 5.7).

   (b) В конечном автомате ищется определение `state_def` для нового состояния (функция `"NewStateDef"`, рисунок 5.8).

```
(DEFINE "NewStateDef"[astatename,defs,x,auto]
   (ALT (CONS' defs state_def tdefs a)
   -- defs=(state_def:tdefs)
      (ALT (CONS' state_def state def a)
      -- state_def=(state:def)
         (ALT (CONS' state e1 e2 astate)
         -- state_def=((e1:e2):def)
            (error_in_auto state_def)
         -- state_def=(astate:def)
            (ALT (EQA' astate astatename)
            -- state_def=(astatename:def)
               (CALL "loop" [state_def, x, auto])
            -- state_def=(astate:def), astate /= astatename
               (CALL "NewStateDef"[astatename,tdefs,x,auto])
            )
         )
      -- state_def=a
         (error_in_auto state_def)
      )
   -- auto=a
      (error_in_auto auto)
   )
)
```

Рис. 5.8: Интерпретатор языка конечных автоматов, функция `NewStateDef`

(c) Выполняется переход автомата в новое состояние, удаляется начальный символ `ax` из входной строки `x`—рекурсивно вызывается функция `"loop"` для состояния `state_def` и строки `tx`.

Рассмотрим два конечных автомата, приведенных на рисунке 5.9.

Автомат `auto1` допускает строки из одних нулей и (если строка допустима) возвращает в качестве результата атом `'E`.

Автомат `auto2` допускает строки из нулей и единиц и (если строка допустима) возвращает в качестве результата:

- атом `'E` – если во входной строке было четное число единиц;
- атом `'O` – если во входной строке было нечетное число единиц.

Вспомогательные функции. На рисунке 5.10 приведены функции для записи выражений. Функция `at` строит терм `ATOM x` по заданной строке `x`.

Функция `cA` по списку символов `(x:y:[])` строит терм вида

```
CONS (ATOM [x] (ATOM [y])).
```

```
auto1   = list[ list[ at"E", cA"0E" ] ]

auto2   = list[ list[ at"E", cA"0E", cA"10" ],
                list[ at"0", cA"00", cA"1E" ] ]
```

Рис. 5.9: Конечные автоматы auto1 и auto2

```
at          :: String -> Term
at x = (ATOM x)

cA          :: [Char] -> Term
cA (x:y:[]) = CONS (ATOM [x]) (ATOM [y])

sT          :: [Char] -> Term
sT []        = ATOM "NIL"
sT (x:xs)    = CONS (ATOM [x]) (sT xs)

list          :: [Term] -> Term
list x        = ht x (ATOM "NIL")

ht          :: [Term] -> Term -> Term
ht []     y = y
ht (x:xs) y = CONS x (ht xs y)
```

Рис. 5.10: Функции для записи выражений

Функция sT формирует строку.

Функция list создает список, дописывая полученный терм в конец имеющегося списка. Здесь используется функция ht, которая формирует голову и хвост списка.

Компиляция автомата auto1 в TSG . Компиляция выполнена вычислением первой проекции Футамуры-Турчина:

```
scp prog_auto ([auto1, ℰ.1], RESTR[])
```

где prog_auto—интерпретатор конечных автоматов, auto1—определение конечного автомата.

Для автомата auto1 результатом компиляции в TSG является рассмотренная ниже функция "F2".

Аргументом функции является произвольная строка E1.

Если эта строка—атом: E1 = A5, то результат работы функции— атом 'E.

Если E1 = E3:E4, проверяется E3.

Если E3 не является атомом, то функция сигнализирует об ошибке.

Если E3 является атомом, то есть E1 = A9:E4, то проверяется равенство этого атома нулю.

```
[(define "F2"[ E1]=
   (alt (cons?  E1 E3 E4 A5)
   -- E1 = E3:E4
      (alt (cons?  E3 E7 E8 A9)
      -- E1 = (E7:E8):E4
         (CONS 'error_in_data(CONS(CONS E7 E8) E4))
      -- E1 = A9:E4
         (alt (eqa?  A9 '0)
         -- E1 = '0:E4
            (call "F2"[ E4])
         -- E1 = A9:E4, A9 /= 0
            (CONS 'error_in_data A9)
      )  )
   -- E1 = A5
      'E
)  )
```

Рис. 5.11: Функция "F2" для автомата auto1

Если этот атом равен нулю, то есть E1 = '0:E4, то далее вызывается функция "F2"[E4].

```
(call "F2"[ E4])
```

Если этот атом нулю не равен, то функция выдает сигнал об ошибке (автомат допускает только нули).

Компиляция автомата auto2 в TSG . Компиляция выполнена вычислением первой проекции Футамуры-Турчина:

```
scp prog_auto ([auto2, ℰ.1], RESTR[])
```

где prog_auto—интерпретатор конечных автоматов, auto2—определение конечного автомата.

Для автомата auto2 результатом компиляции в TSG являются две ниже приведенные функции "F2"[E1] и "F18"[E4].

Функция "F2"[E1] для автомата auto2 отличается от функции "F2"[E1] для автомата auto1 только тем, что найденный в данной строке символ проверяется на равенство нулю или единице (автомат допускает нули и единицы). В тот момент, когда находится 0 в данной строке, вызывается функция "F2"[E4]

```
(call "F2"[ E4]),
```

а когда найденный символ равен единице, вызывается функция "F18"[E4]

```
(call "F18"[ E4]).
```

Функция "F18" работает так же, как "F2", только если в строке найденный символ—0, вызывает функцию "F18" для дальнейшего разбора, а если найденный символ—1, вызывает функцию "F2".

```
[(define "F2"[ E1]=
   (alt (cons?  E1 E3 E4 A5)
   -- E1 = E3:E4
      (alt (cons?  E3 E7 E8 A9)
      -- E1 = (E7:E8):E4
         (CONS 'error_in_data(CONS(CONS E7 E8) E4))
      -- E1 = A9:E4
         (alt (eqa?  A9 '0)
         -- E1 = '0:E4
            (call "F2"[ E4])
            (alt (eqa?  A9 '1)
            -- E1 = '1:E4
               (call "F18"[ E4])
            -- E1 = A9:E4, A9 /= '0, A9 /= '1
               (CONS 'error_in_data A9)
      )  )  )
   -- E1 = A5
      'E
)  )
```

Рис. 5.12: Функция "F2" для автомата auto2

```
(define "F18"[ E4]=
   (alt (cons?  E4 E19 E20 A21)
   -- E4 = E19:E20
      (alt (cons?  E19 E23 E24 A25)
      -- E4 = (E23:E24):E20
         (CONS 'error_in_data(CONS(CONS E23 E24) E20))
      -- E4 = A25:E20
         (alt (eqa?  A25 '0)
         -- E4 = '0:E20
            (call "F18"[ E20])
            (alt (eqa?  A25 '1)
            -- E4 = '1:E20
               (call "F2"[ E20])
            -- E4 = A25:E20, A25 /= '0, A25 /= '1
               (CONS 'error_in_data A25)
      )  )  )
   -- E4 = A21
      '0
)  )
```

Рис. 5.13: Функция "F18" для автомата auto2

5.3.3 Выводы

В разделе 5.3.1 рассмотрены примеры компиляции с языка конечных автоматов в язык TSG. По заданным конечным автоматам и интерпретатору языка конечных автоматов получены программы, функционально эквивалентные заданным конечным автоматам, на языке TSG. То есть, полученные программы есть результат компиляции программ с языка конечных автоматов в язык TSG. Выполнена первая проекция Футамуры-Турчина.

Глава 6

Заключение

В данной работе разработан, реализован, описан и обоснован суперкомпилятор для языка TSG, использующий базовые понятия метавычислений над TSG из работы [3]. Таким образом, данная работа является прообразом продолжения (второго тома) монографии [3].

Среди прочего в данной работе:

1. Разработан и обоснован алгоритм обнаружения возможного зацикливания.
2. Разработан и обоснован алгоритм обобщения двух конфигураций `cUp` и `cDn`.
3. Разработан и обоснован алгоритм проверки вложенности `cUp` $\succeq$ `cDn` двух конфигураций.
4. Разработан (на базе упомянутых выше—пункты 1–3—функциональных блоков и с использованием алгоритмов из [3]) и обоснован собственно алгоритм суперкомпилятора.
5. Проведены и описаны реальные примеры суперкомпиляции, призванные продемонстрировать способность выполнения суперкомпилятором нетривиальных глубоких преобразований программ.

Все приводимые в работе тексты метапрограмм являются фрагментами *работающей* Gofer-программы, а все приводимые примеры суперкомпиляции—результатами реального выполнения этой программы.

Полный текст программы и примеров, полный текст данной работы в виде файлов доступны заинтересованному читателю по анонимному FTP:

`ftp://ftp.botik.ru/pub/local/Sergei.Abramov/Scp-proj/`

Литература

[1] Kruskal J.B. Well-quasi-ordering, the Tree Theorem and Vazsonyi's Conjecture // Trans. Amer. Math. Soc., 95, 1960, pp.210-225

[2] N. Dershowitz, J.-P. Jouannaud, Rewrite Systems // Jan van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Volume B, "Formal models and semantics", The MIT Press, 1990, pp.243-320

[3] Абрамов С.М. Метавычисления и их применение // Москва, Наука-Физматлит, 1995, 128 с.

[4] Abramov S.M. Metacomputation and program testing // 1st International Workshop on Automated and Algorithmic Debugging. Linköping, Sweden, 3-5 May, 1993, p.121–135, Linköping University 1993.

[5] Futamura Y. Partial evaluation of computation process—an approach to a compiler-compiler // Systems, Computers, Controls, 2(5): 45-50, 1971.

[6] 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.

[7] Turchin V.F. The Phenomenon of Science // Columbia University Press, New York 1977. (Русскоязычный вариант: Турчин В.Ф. Феномен науки: Кибернетический подход к эволюции // М., Наука, 1993, 296 с.)

[8] Turchin V.F. A supercompiler system based on the language Refal // SIGPLAN Notices, 14(2): 46-54, 1979.

[9] 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.

[10] Turchin V.F. Semantic definitions in Refal and automatic production of compilers // Jones N.D. (ed.), Semantics-Directed Compiler Generation. (Aarhus, Denmark). Lecture Notes in Computer Science, Vol. 94, 441–474, Springer-Verlag 1980.

[11] 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.

[12] Turchin V.F. The concept of supercompiler // ACM TOPLAS, 8(3): 292-325, 1986.

[13] Turchin V.F. Refal: a language for linguistic cybernetics // City College of the City University of New York. 1986

[14] 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.

[15] 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.

[16] Glück R. Inside the supercompiler // City University New York. Technical Report 1989.

[17] 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.

[18] Абрамов С.М. Метавычисления и логическое программирование // Программирование No 3, 1991, стр.31–44.

[19] 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.

[20] Turchin V.F. Program transformation with metasystem transitions // Journal of Functional Programming, 11, 1993

[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] 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.