ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Алгоритмы, которые оперируют над исходным кодом языка программирования, классически построены таким образом, который предполагает сначала построение абстрактного синтаксического дерева по заданным синтаксическим правилам, а затем его обработку соответственно с семантикой языка. Данный подход находит широкое применение в сфере изучения формальных языков и исторически продемонстрировал свою жизнеспособность. К плюсам разделения стадий парсинга и произведения вычислений над деревом является возможность разобщить эти, как правило, весьма сложные процедуры и иметь возможность раздельно их реализовывать, тестировать, а также обновлять. Так решаются задачи инкапсуляции.

Однако в последние годы, в силу роста количества инструментов для работы с формальными языками, которые достаточно просты для работы с ними широкого круга разработчиков программного обеспечения, набирают популярность предметно-ориентированные языки. Обычно они имеют более простые синтаксис и семантику, чем такие типичные языки общего назначения, как Java или C++. Соответственно, для них часто оказывается нецелесообразной сложная архитектура обрабатывающих эти языки средств, типичная архитектуре программ, оперирующих исходным кодом на языках общего назначения: синтаксис в предметно-ориентированных языках нередко формулируется так, чтобы быть наиболее близким к отражаемой семантике, и, таким образом, построение абстрактного синтаксического дерева становится простой задачей.

Для записи грамматик формльных языков существует большое количество разнообразных нотаций, однако среди наиболее популярных из них нет такой, которая бы позволяла выражать вычисления в привязке к синтаксическим правилам. Работа основана на предположении, что рост популярности предметно-ориентированных языков создаёт условия для того, чтобы такая нотация получила практическое применение.

Задание вычислений в совокупности с грамматикой языка может быть полезно во многих ситуациях, в частности, при разработке алгоритмов для статического анализа. Как показало чтение публикаций по статическому анализу, в список наиболее популярных способов задать правила для статического анализатора входят использование формальных систем вроде лямбда-исчисления или машин

Тьюринга, языков общего назначения вроде FORTRAN или C или упрощённых языков для выражения алгоритмов вроде используемого Дональдом Кнутом в собрании сочинений «Искусство программирования». Использование формальных систем повышает уровень разработчика, требуемый для понимания публикации и реализации алгоритма; привязка алгоритма к языку общего назначения может, в силу наличия большого числа граничных случаев в спецификации этих языков, усложнить перенос алгоритма на другой язык; упрощённые же языки зачастую не обладают теми составляющими семантики, которая обрабатывается некоторым конкретным алгоритмом статического анализа. Возможным решением является создание нового языка, описание его грамматики и вычислений над ней в качестве минимального рабочего примера работы алгоритма статического анализа.

К статическому анализу относится также проверка корректности совокупности языковых конструкций: так, компилятор осуществляет статический анализ кода на предмет соответствия его реализации стандарта языка. Если анализ проходит успешно, компилятор переходит к стадиям оптимизации и трансляции, если нет, то сообщает об ошибке. Таким образом, вычисления над языковыми конструкциями также могут определять корректность или некорректность программы в аспектах, которые невозможно выразить в грамматике: к примеру, вычисление может проверять, происходит ли обращение к переменной до того, как она определена.

В рамках данной работы создана нотация, которая позволяет задать совместно с грамматикой языка правила, по которым осуществляются вычисления над его конструкциями. Также приведён пример того, как в этой нотации выразить несложный алгоритм проверки границ целочисленных значений.

1 РАЗРАБОТКА НОТАЦИИ

1.1 Анализ имеющихся решений

После анализа было выделено несколько классов нотаций и их типичных представителей.

- Формальные нотации. Такими нотациями задаётся набор конструкций, которыми может быть описана грамматика языка. В список таких нотаций входят BNF и его расширения, синтаксическая нотация Вирта[1], грамматика ван Вейнгаардена и другие.
- Нотации, оформленные в конструкциях некоторого языка программирования. Примерами могут служить аппликативные парсеры на языке Haskell[2], модуль языковых грамматик в Perl6[3], синтаксические нотации в языке Coq[4] и многое другое.
- Нотации входных данных для генераторов парсеров. Таковыми являются, к примеру, описанные в инструкциях к утилитам `yacc` и `bison` и библиотеке Marpa на языке Perl.

Разрабатываемая нотация занимает промежуточную позицию между первым и третьим классом: с одной стороны, она не является частью языка программирования и задаёт читаемое человеком, легко форматируемое формальное описание, с другой, позволяет составить использующий её генератор парсеров.

Из уже существующего наиболее близок к нашей разработке модуль Marpa для языка Perl. Он позволяет в удобочитаемом текстовом виде задать грамматику языка, а также даёт возможность задать над языковыми конструкциями вычисления. К сожалению, он не предполагает формализацию вычислений: алгоритмы, оперирующие термами, не входят в грамматику: в ней указывается лишь имя функции в программе-обработчике, в которую будет передана ветвь синтаксического дерева. Таким образом, для задания синтаксически-управляемого анализа дерева недостаточно предоставить входные данные для Marpa, требуется также осуществить транзитивное замыкание по операции вызова функций программы-обработчика. Дополнительной сложностью для использования в наших целях Marpa является то, что функции написаны на языке Perl, который не рассчитан на лёгкость чтения.

1.2 Выбор составляющих нотацию языков

В силу того, как обширен выбор формальных нотаций и языков программирования, было решено, что возможно найти среди них подходящие компоненты для разрабатываемой нотации и не нужно проектировать всё самостоятельно.

В первую очередь необходимо выбрать средство описать грамматику языка. Требования к средству выражения грамматики при этом были таковы:

- Наличие фундаментальных правил, к примеру, для обработки чисел или определённых классов символов Юникода. Это позволит задать в рамках разрабатываемой нотации широкий набор фундаментальных парсеров, которые будут действовать сообразно ожиданиям: к примеру, парсер знаковых и беззнаковых целых чисел.
- Поддержка комментариев: при расширении описания грамматики заданием вычислений над ней получается больше информации, в результате чего усложняется понимание написанного. Комментарии помогают смягчить эту проблему.
- Распространённость: базовая нотация должна быть известна большинству и понятна без дополнительных комментариев.

На основе этих критериев была выбрана нотация Augmented Backus-Naur Form[5]: семейство BNF-нотаций широко распространено, а ABNF предоставляет требуемый нам функционал.

Далее нужно выбрать язык, на котором будут описываться вычисления. Требования к нему такие:

- Популярность в среде исследования языков программирования: пусть целевая аудитория нашей нотации не ограничивается исследователями, важно, что любой желающий написать свой доменный язык так или иначе должен ознакомиться с некоторым теоретическим минимумом, и для этого так или иначе неизбежно погружение, пусть и не глубокое, в академическую среду, окружающую формальные языки.
- Простота написание интерпретатора этого языка: одной из целей проекта является создание нотации, которую пользователь сможет расширить под свои нужды. В силу этого является важной простота реализации утилит, которые разрабатываемую нотацию обрабатывают.
- Отсутствие или малое количество аннотаций типов: понятно, что область

использования каждого синтаксического элемента языка, который представлен в нотации, строго ограничена семантикой этого языка. Таким образом, уже имеется некоторая аннотация типов в виде описываемой грамматики. Задействование дополнительных конструкций для уточнения типов может выглядеть громоздко.

- Простота программирования в чистом функциональном стиле: в силу того, что намеренно не уточняется, какого рода парсеры будут использовать разрабатываемую нотацию, порядок вычислений совершенно неопределённый. Поэтому необходимо, чтобы любое вычисление могло быть отложено, прервано или начато в любой момент времени без влияния на то, как будут вычислены выражения для любых не находящихся в зависимости от данного термов. Таким образом, требуется, чтобы язык не поощрял стиль программирования, при котором программа сохраняет большое количество данных в глобальном состоянии, и предоставлял удобный инструментарий для написания чистых функций.
- Независимость выполнения языка от выравнивания его кода. Так как описания вычислений будут связаны с грамматикой, сложно будет обеспечить консистентное выравнивание программы так, чтобы это не ухудшило в значительной степени читаемость.

Наиболее популярными языками, обладающими заданными свойствами, являются языки семейства LISP. Был выбран язык Scheme версии r6rs как наиболее ориентированный на исследования формальных языков.

1.3 Нотация

Основой нотации будет, как уже отмечено, Augmented Backus-Naur Form. Не поддерживаются только комментарии в треугольных скобках: ожидается, что они обрабатываются человеком и словесно описывают некоторую конструкцию. Так как нотация явным образом рассчитана на обработку программную, эта конструкция не имеет смысла.

В рамках правила, после каждой альтернативы может располагаться точка с запятой, за которой в круглых скобках `()` представлено тело функции на языке Scheme, принимающей данную альтернативу как параметр.

1.3.1 Передача параметров

По спецификации ABNF[5], альтернатива представляет собой некоторое количество сконкатенированных значений. Значение может представлять собой как атомарное правило, так и набор сконкатенированных значений с некоторым модификатором повторений: к примеру, «это значение может повторяться от 5 до 10 раз».

Атомарное правило, если для него заданы правила обработки, передаётся как результат, возвращаемый обработчиком, а если обработчик не задан, то передаётся строковое представление терма, который соответствует этому правилу. Например, если есть правило `bin = "zero"/ "one"` и оно выполнилось для строки `"zero"`, то в функцию для `top = bin : (тело)` придёт строка `"zero"`, а если есть правило `zero = "zero": (identity 0) / "one": (identity 1)`, то придёт `0`.

Если к значению применён модификатор, задающий количество, то будет возвращён список. Например, функция для правила `top = [bin] : (тело)` пример одно из: `'(0)`, `'(1)`, `'()` или соответствующие строки вместо чисел, если вычисление для правила `bin` не определено.

В случае значения, представленного конкатенацией, тоже возвращается список: `top = bin bin : (тело)` примет элемент декартового квадрата множества $\{0, 1\}$. По умолчанию в список включены все аргументы конкатенации, но это можно изменить. Достаточно справа от аргумента указать в фигурных скобках его имя — последовательность цифр и строчных и заглавных латинских букв. Тогда в качестве параметра будет возвращён список, однако в нём будут только те аргументы, у которых есть имя. Имя должно быть уникальным в рамках данного правила за исключением особого случая, когда имя — пустая строка. Аргументов, у которых имя является пустой строкой, может быть любое количество. Пример: `bin{t1} LWSP bin{} LWSP bin` не будет вычислять третий `bin`. В качестве параметра будет передан список из результатов вычисления первого `bin` и второго. Если бы ни к одному аргументу не было приписано фигурных скобок, вернулся бы список из пяти элементов, где нечётные представлены результатами вычислений `bin`, а чётные — строками, которые являются некоторой комбинацией пробельных символов.

Для указания имени составного выражения надо брать выражение в скобки,

даже если это не требуется по стандарту для ABNF.

Если для сложного правила не определены правила обработки, то оно возвращает то, что само бы принимало в функцию, если бы она имелась.

1.3.2 Среда

При исполнении функции принимают неявным параметром среду исполнения: произвольный объект, в который функция может что-то записать и из которого может что-то считать.

Однако если позволить произвольным правилам менять среду, возникает проблема определения порядка вычислений: в зависимости от него обработка грамматики будет давать разный результат.

Таким образом, требуется механизм для определения порядка выполнения функций, которые могут менять среду. При этом вводить порядок в тех альтернативах, где среда не меняется, не нужно: вычисления там будут в любом случае чистыми.

Далее, было решено, что если в альтернативе задействовано только одно правило, но оно может менять среду, порядок требуется указывать. Таким образом можно сразу видеть, является ли правило «чистым». С этой точки зрения, к примеру, в языке Haskell и семействе языков ML, если в качестве среды принять сведения о привязанных в символам значений, выражения для вызова функций являются чистыми, а оператор `let ... in ...` и объявление новых глобальных символов — нет.

Порядок исполнения задаётся таким образом: после имени правила через двоеточие в фигурных скобках указано число от `1`. Если хоть один операнд операции конкатенации имеет номер, то у каждого операнда с именем также должен быть номер. Если у выражения есть номер, то у оборачивающего его правила тоже должен быть номер. Номера могут быть произвольными. Единственное ограничение — им нельзя совпадать. Таким образом помечаются все правила, в которых может изменяться среда, и в их рамках гарантируется, что каждое составляющее выражение будет при любом наборе правил обхода синтаксического дерева иметь конкретный номер.

Рассмотрим конкретные примеры. Пусть правило `bin` меняет среду. Тогда правило

`top = (3*bin{:1}){:2} bin {t1:3} ("thirty four": (34)){:1} : (тест)` выполнит сначала правило `"thirty four"`, которое вернёт `34`, затем правило `3*bin{:1}`, которое вернёт список из трёх элементов, обработанных по правилу для `bin` (для которого нужно обязательно указать номер и в рамках данной вложенной конструкции, поскольку `bin` не является чистым вычислением), затем — правило `bin`. После этого будет исполнено правило `(тест)`.

Иногда бывает нужно провести некоторые дополнительные вычисления между выполнением разных правил. К примеру, в приведённом выше примере может потребоваться выполнить операцию между правилами `"thirty four"` и `3*bin{:0}`. Специального синтаксиса для таких случаев нет: можно задать пустую строку и провести вычисления для неё, например, так:
`top = (3*bin{:1}){:3} bin{:4} (0*"never needs matching": (вычисление)){:2} ("thirty four": (34)){:1} : (тест).`

Также часто появляется необходимость обратиться к правилу, которое меняет среду, но не сохранять результат этих изменений. К примеру, для организации областей видимости можно изменять среду, добавляя туда новые переменные, но было бы неудобно вычищать список привязанных имён вручную. Вместо этого можно объявить, что данное правило получает собственную копию среды и произвольно её меняет, что не сказывается на вычислениях в других правилах. Однако было принято несколько более общее решение: добавлена возможность указывать для каждого правила, от какого правила унаследовать среду. Обозначается это так: после порядкового номера вычисления через ещё одно двоеточие указывается порядковый номер вычисления, из которого требуется взять копию среды. Если нужно взять начальную копию, можно адресовать мнимое правило `0`. За среду, которую получает обработчик данной альтернативы, принимается среда, возвращённая правилом, обладающим наибольшим номером.

К примеру, правило `top = bin{t1:1} bin{t2:2:0} bin{t3:3:1}` выполнит сначала первый `bin`, затем второй, но со средой, в которой нет изменений, привнесённых первым, и, наконец, третий в среде, изменённой первым. Всё правило возвращает среду, которая установлена в третьем `bin`. Таким образом, изменения, которые осуществил в среде второй `bin`, теряются.

1.3.3 Интеграция со Scheme

Scheme — это «диалект Lisp со статической областью видимости, полностью поддерживающий хвостовую рекурсию», в котором «находят удобное выражение функциональный, императивный стили и стиль передачи сообщений»[6].

Эта гибкость, полезная для разработчика под проектируемую нотацию, затрудняет связь между разными функциями. Наивным подходом могло бы стать размещение каждого объявленного вычисления в отдельной функции, которая вызывается с заданными параметрами; среда при этом копируется из некоторой скрытой переменной в локальную перед вызовом, а после завершения функции, если ей позволено менять среду, копировать модифицированную среду в оригинальную переменную. Проблема возникает в связи с тем, что язык Scheme оперирует сущностями под названием «продолжения». Вкратце, они позволяют нарушить обычный ход работы программы, сохранить текущее положение в ней и восстановить его в произвольный момент. К примеру, и это то, что не позволяет применить наивный подход, может произойти такое: некоторое вычисление, которому разрешено менять среду, записывает в неё своё продолжение; затем некоторое иное вычисление, которому менять среду не дозволено, считывает это продолжение из среды и попадает в ту самую первую функцию и уже из этого контекста меняет среду. Очевидно, что для свершения такого трюка требуется углубленное знание Scheme и осознанное решение так поступить, однако среди решений при высокоуровневом проектировании нотации имеется намерение не допускать написание кода, который на разных реализациях нотации выполняется по-разному, даже если это совершается намеренно.

В языке Scheme каждое значение соответствует одному из предикатов:[6]

- `boolean?` — булева переменная;
- `pair?` — пара;
- `symbol?` — идентификаторы;
- `number?` — числа;
- `char?` — символы;
- `string?` — строки;
- `vector?` — векторы;
- `procedure?` — функции;
- `null?` — пустые списки.

Отмечается, что этим исчерпываются все возможные в Scheme типы.

Всё перечисленное, кроме процедур, не может менять ход работы программы; следовательно, достаточно запретить хранение процедур в среде, чтобы стало невозможно выйти за заданные границы из некоторого вычисления.

Удачным способом обеспечить эти условия нам кажется хранение среды в виде строки. В самом деле, все перечисленные типы сериализуются в строку, которую возможно однозначно десериализовать обратно, лишь процедуры десериализовать нельзя. К примеру, в guile сериализованное продолжение выглядит так:

```
> (call/cc (lambda (x) x))
$1 = #<continuation 1b97de0>
```

`#<continuation 1b97de0>` является некорректным синтаксисом для описания объекта в Scheme.

Таким образом, сериализация среды и проверка, является ли полученная строка синтаксически корректной программой на Scheme, позволяет понять, имеются ли в среде процедуры. Однако так как этот подход может оказаться в некоторых случаях неподходящим в силу производительности, такая реализация не обязательна.

К возвращаемым значениям для вычислений таких требований не предъявляется: доступ к ним имеют только стоящие выше по иерархии правила, и если текущее вычисление имеет доступ к смене среды, значит, и у всех значений выше по иерархии имеется строгий порядок выполнения.

В силу этих соображений вводится такой протокол: вычисления должны быть описаны так, будто они располагаются вместо слова «body» в данной программе:

```
(let ([env internal_env] [internal_env '()])
  (bind-args
    (lambda (param) (body))))
```

Здесь функция `bind-args` совершает такие действия: для каждого операнда операции конкатенации верхнего уровня, у которого есть имя, она привязывает к этому имени соответствующее значение.

Следует обратить внимание, что body находится в скобках. На языке Lisp это означает, что body является не скаляром, а некоторым вычислимым списком.

Скаляры же не приравниваются к функциям с нулевой арностью. Соответственно, когда требуется вернуть скаляр, недостаточно написать `(0)`: 0 не является функцией и вычисление его является ошибкой. Вместо этого следует использовать функцию `identity`, которая определена как `lambda (x) x` и написать `(identity 0)`. Это является вычислением и ошибки не произойдёт.

Ещё один важный вопрос — подключение библиотек и введение вспомогательных функций. В описанной на настоящий момент нотации не имеется средств ввести функцию и исполнять её где угодно: в среде хранение кода запрещено, а передача данных вниз по дереву невозможна.

В этих целях вводим дополнительную сущность: перед тем, как начнётся перечисление правил грамматики, можно указать в скобках произвольный код, который будет исполняться каждый раз перед началом любого вычисления. Например:

```
(begin
    (import (rnrs))
    (define (test  s) (* 2 (string-length s)))
    (define (test' s) (* 3 (string-length s))))

bin = "zero" : (test param) / "one" : (test' param)

top = 2bin : (car param)
```

2 РАЗРАБОТКА СТАТИЧЕСКОГО АНАЛИЗАТОРА

2.1 Описание анализатора

Разработан и оформлен в представленной нотации алгоритм статического анализа, который вычисляет границы значений целочисленных переменных, сигнализирует о переполнении чисел, размерность которых задана, и определяет, в машинное слово какого размера гарантированно поместятся все значения числа.

При этом разработка алгоритма осуществлена с попутной формализацией языка, анализ программ на котором осуществляется. Стоял выбор: поступить так или взять некоторый существующий язык и проанализировать его. Было предположено, что тогда, когда целью стоит разработка алгоритма, а не конкретного анализатора под известный язык, такой подход обладает рядом существенных преимуществ, среди которых можно выделить такие:

- Итеративная натура разработки. При использовании существующей языковой инфраструктуры даже первая версия алгоритма должна быть составлена с учётом большинства языковых конструкций, даже если уровень их поддержки незначительный. Тестирование алгоритма в таком случае может быть весьма затруднительно, поскольку в общем случае сложно понять, ошибка содержится в механизме работы анализатора или в неполноте покрытия языка. Когда же язык развивается вместе с алгоритмом, каждая конструкция может быть добавлена после того, как алгоритм протестирован для всех существующих, что позволяет исследователю более быстро обнаруживать источник ошибок.
- Переносимость. Часто к языкам одного семейства можно применить один и тот же алгоритм статического анализа с небольшими изменениями. В случае, когда разработка не привязана к конкретному языку, это сделать проще: иначе может быть затруднительно выделить детали алгоритма, которые определены не семантикой, а деталями структуры языка.
- Более глубокое понимание. При разработке алгоритма для конкретного языка необходимо полностью учитывать его структуру, и это лишает исследователя способности внести в язык мелкие правки для определения, как это повлияет на структуру алгоритма. Такое отсутствие гибкости может помешать обретению понимания, какие именно особенности языка приводят к корректности алгоритма.

- Более простая формальная верификация. Семантика большинства зрелых языков сложна для описания в терминах программных комплексов для автоматической верификации, и это затрудняет степень доверия, которую мы можем иметь к анализу корректности алгоритмов статического анализа. Семантика языков, созданных специально для исследования методик анализа, напротив, часто весьма просто формализуется, что позволяет при формальной верификации уделять основное внимание алгоритму, а не языку.

В силу этих причинпроцесс создания простого механизма для анализа области возможных значений демонстрируется путём задания примитивного языка и постепенного его расширения с сопутствующим усложнением алгоритма.

2.2 Система типов

Первым делом, даже перед определением набора конструкций, которыми будет обладать язык, требуется определить, какими видами данных он должен оперировать.

Поскольку при определении областей допустимых значений в самом деле важны лишь числа, в язык не входят тип-суммы, тип-произведения, строки, хэш-таблицы и другие типы, свойственные языкам, применимым при настоящей разработке. Также не рассматриваются булевы значения как отдельный тип, вместо этого используются ноль в качестве ложного значения и всё остальное в качестве истинного. В языке нет и массивов: они интересны лишь тогда, когда может произойти выход за их границы, и в этих целях можно их представлять как число n элементов в них, а вставку и удаление — как увеличение и уменьшение этого числа; в таких терминах проверка на выход за границы массива выражается как требование того, чтобы запрашиваемый индекс не превосходил n.

Более того, поддерживаются лишь целые числа. Реализация рациональных чисел в виде пары целых лишь усложнила бы алгоритм без нужды: отношения между рациональными числами легко представимы в терминах отношений между целыми. Введение чисел с плавающей точкой в общем виде же сделало бы анализатор значительно более сложным: в силу неточной натуры таких чисел для их формального анализа требуется описание стандарта их реализации,

который, как правило, весьма сложен. В силу этого числа с плавающей точкой выходят за рамки данной работы.

Требуются два типа целых чисел. Первый из них (далее — «бесконечные числа») — это тип произвольных неотрицательных чисел. Для переменных такого типа нам требуется автоматически определять максимальное значение, которое они могут принимать по ходу программы, и находить соответствующий машинный тип, который следует использовать для представления этой переменной в памяти. Второй тип (далее — «числа по модулю») имеют числа по модулю некоторой степени двойки. Их максимальное значение должно быть проверено для обнаружения потенциальных переполнений. Пусть в некоторых случаях переполнение может быть желаемым поведением, формализуемый язык не поддерживает переменные, которые не считают его ошибкой. Однако эта проблема не существенная: в случаях, когда необходимо организовать переполнение, достаточно сконвертировать число в бесконечное, осуществить требуемую операцию, а затем снова взять значение по требуемому модулю.

2.3 Операции над числами

Вполне очевидно, что так как оба типа в нашем языке задают целые числа, описание языка следует начать с задания операций над ними.

В первую очередь надо отметить существование механизма конвертирования между типами. Для этого необходимо лишь рассматривать бесконечные числа как числа по модулю бесконечности. В таком случае конвертирование в число по модулю m — просто взятие старого значения по модулю m.

Далее, арифметические операции сложения, вычитания, умножения и деления работают на бесконечных числах по общепринятым правилам. У чисел по модулю тоже есть эти операции, и деление не обнаруживает число, обратное по модулю, а ведёт себя так же, как и на бесконечных числах. Это поведение привычное и встречается в подавляющем большинстве языков программирования, поддерживающих деление, даже в тех, которые синтаксически определяют, в регистрах какой разрядности содержится число. Вводится ограничение на арифметические операции между значениями разных типов: перед этим необходимо явным образом сконвертировать их.

Важным отличием между бесконечными и взятыми по модулю числами яв-

ляется то, что на последних введены битовые операции. Несложно увидеть, что побитовая инверсия бесконечных чисел не определена: получается бесконечность как значение, а его хранение не предполагается. И, несмотря на то, что конъюнкция и дизъюнкция заданы, без отрицания эти операции не составляют булеву алгебру и, как следствие, имеют ограниченную пользу. Ожидается, что для хранения бинарных структур или битовых флагов, где битовые операции часто применяются, будут использоваться числа заранее известной длины в двоичном представлении.

И, разумеется, числа требуется сравнивать. Для этого вводятся два оператора: проверка на равенство и проверка на отношение «меньше, чем». При сравнении двух чисел, которые взяты по разным модулям, происходит неявное конвертирование обоих чисел к наименьшему из этих модулей. Результат сравнения представлен числами по модулю 2.

Было бы, несомненно, весьма соблазнительной перспективой создать язык, который поддерживает лишь перечисленные операции и присваивание переменных: тогда весь анализ заключался бы в простом выполнении программы: все имеющиеся операции имеют строго определённый результат. В силу этого необходимо ввести источник неопределённости, и потому в язык вводится оператор **rand**, который возвращает произвольное целое число в промежутке $[0; +\infty)$.

Внимательное рассмотрение этого оператора, однако, показывает, что требуется ввести ограничения, чтобы система сохраняла консистентность. Нельзя сохранить **rand** в переменную бесконечного типа, поскольку это бы потребовало бесконечно великого объёма памяти, занимаемого данной ей. Таким образом, необходимо брать модуль **rand** перед использованием. В силу этого поддерживаются лишь случайные числа по модулю степени двойки.

Особая роль оператора **rand** заставляет считать его сущностью, которая вовсе не является числом, — именно в силу этого он называется оператором, а не значением. Очевидно, что это не число по модулю, но и считать его бесконечным числом тоже нет особого смысла. Нужно рассмотреть все операции, чтобы убедиться в этом.

rand/a Если исходное распределение принять за $[0; \infty)$, то после деления на некоторое число оно продолжает быть $[0; \infty)$. Действительно, любое число n встречается в конечном распределении так же часто, как an — в исходном.

$a/$**rand** Для любого конечного a имеется лишь a чисел, деление на которые с дальнейшим округлением вниз даёт ненулевой результат. В то же время имеется бесконечное количество чисел таких, что результат деления a н а них — 0. Соответственно, деление на **rand** всегда равно 0.

rand $\pm a$ Операции сложения и вычитания помогают нам сдвинуть распределение так, чтобы оно стало $[\pm a; \infty)$. Однако так как после этого всё равно придётся взять результат по модулю k, он останется $[0; k)$ в силу того, что $[0; a] \equiv [k; k + a] \mod k$.

rand $\cdot k$ Умножение может быть полезным, но весьма узко. Если k и n взаимно просты, то $kr \mod n$ имеет распределение $[0; n)$. Если же нет, то область допустимых значений исключает некоторые числа. К примеру, если k тоже является степенью 2, то вместо $[0; n)$ распределение результата станет $\{0\} \cup \{k\} \cup \{2k\} \cdots \left\{\left(\frac{n}{k} - 1\right) k\right\}$. Однако несложно убедиться, что таких же результатов можно добиться и за границами начального конвертирования к модульным числам.

В силу того, что ни одна операция не имеет важной роли, без потери общей натуры алгоритма можем объявить **rand** особой сущностью, которая поддерживает лишь конвертирование в модульное число.

Выражения представлены в инфиксной нотации и имеют такие же ассоциативность и приоритет операций, как и в языке C. Разумеется, точно так же работают и скобки, позволяющие задать иной приоритет. У операторов конвертирования приоритет выше, чем у всего остального, причём конвертирование в числа по модулю — операция более приоритетная, чем конвертирование в бесконечные числа.

Построим ABNF-грамматику подмножества языка, которое мы сейчас описали.

⟨*expr_1*⟩ ::= ⟨*expr_2*⟩ '|' ⟨*expr_1*⟩
  | ⟨*expr_2*⟩

⟨*expr_2*⟩ ::= ⟨*expr_3*⟩ '&' ⟨*expr_2*⟩
  | ⟨*expr_3*⟩

⟨*expr_3*⟩ ::= ⟨*expr_4*⟩ '<' ⟨*expr_3*⟩
  | ⟨*expr_4*⟩ '=' ⟨*expr_3*⟩
  | ⟨*expr_4*⟩

⟨*expr_4*⟩ ::= ⟨*expr_5*⟩ ’+’ ⟨*expr_4*⟩
| ⟨*expr_5*⟩ ’-’ ⟨*expr_4*⟩
| ⟨*expr_5*⟩

⟨*expr_5*⟩ ::= ⟨*expr_6*⟩ ’*’ ⟨*expr_5*⟩
| ⟨*expr_6*⟩ ’/’ ⟨*expr_5*⟩
| ⟨*expr_6*⟩

⟨*expr_6*⟩ ::= ’-’ ⟨*aexpr*⟩
| ’~’ ⟨*aexpr*⟩
| ⟨*aexpr*⟩ ASP ‘`bits`’ ASP ⟨*mod-number*⟩
| ⟨*aexpr*⟩

⟨*aexpr*⟩ ::= ⟨*var-name*⟩
| ⟨*number*⟩
| ‘`inf`’ ASP ⟨*expr*⟩
| ’(’ LWSP ⟨*expr*⟩ LWSP ’)’
| ‘`rand`’ ASP ‘`bits`’ ASP ⟨*mod-number*⟩

⟨*var-name*⟩ ::= ‘`@`’ ALPHA *alnum

⟨*alnum*⟩ ::= ALPHA
| DIGIT

⟨*mod-number*⟩ ::= ⟨*non-zero-digit*⟩ [natural-number]

⟨*natural-number*⟩ ::= 1*DIGIT

⟨*non-zero-digit*⟩ ::= %x31-39

⟨*ASP*⟩ ::= WSP / CRLF WSP

Примером выражения, которое совпадает с правилом ‘`<expr>`’, является:

```
inf ~ ((32 & @x bits 6) = 0)
```

Это выражение описывает взятие значения `@x` по модулю 64 (поскольку $2^6 = 64$), производит его побитовую конъюнкцию с 32, таким образом получая значение пятого бита, сравнивает результат с нулём, что даёт число по модулю 2, инвертирует результат и конвертирует в бесконечное число. Таким образом, выражение предоставляет число 1 бесконечного типа, если пятый бит **@x** установлен, и 0 в противном случае.

2.4 Анализ арифметики

Пусть представленное не является особенно полезным языком, об этом уже можно рассуждать.

Важная цель — определить, какие значения могут принимать выражения.

Так как ещё не оформлен механизм записи в переменные, часть языка, связанная с ними, будет игнорироваться.

Наивный способ анализа арифметических выражений — рассмотрение каждого оператора по отдельности и свёртывание дерева, представленного их комбинацией, по правилам, заданным для них, так, чтобы результатом был простой числовой промежуток. И пусть это бы сработало на текущей стадии, для этого бы потребовалось полностью отказаться от такого механизма позже, после ввода переменных.

В качестве примера можно рассмотреть данное выражение:

```
@x - 1 * (@x + 2)
```

Любому наблюдателю очевидно, что для любого значения `@x` результатом данного выражения будет -2. Но наивное решение сообщает, если `@x` $\in [0; 15]$,

```
[0; 15] - [1] * ([0; 15] + [2]) =
[0; 15] - [1] * [2; 17] =
[0; 15] - [2; 17] =
[-15; 13]
```

Очевидно, нужно более продуманное решение.

Единственный источник неопределённости — оператор **rand**. Потому результат выполнения программы можно представить как определённую комбинацию значений **rand**, объединённых арифметическими операциями. В силу этого каждому появлению **rand** в программе предлагается сопоставить уникальный идентификатор. Все выражения, в которых — напрямую или при раскрытии значения переменной — имеется такой идентификатор, должны его сохранять при вычислении результата.

Продемонстрируем пример вычисления выражения, представленного выше, в этих терминах. Пусть `@x` имеет значение $R \mod 16$, где R — идентификатор **rand**. Тогда

```
{R mod 16} - {1} * ({R mod 16} + {2}) =
```

```
{R mod 16} - {1} * {R mod 16 + 2} =
{R mod 16} - {1 * (R mod 16 + 2)} =
{R mod 16 - 1 * (R mod 16 + 2)}
```

Теперь, при подставлении значения $[0; 15]$ в `R mod 16`, в любом случае получается корректный результат: число -2.

Возможно также расширить механизм алгебраическими преобразованиями: к примеру, правилами дистрибутивности умножения и конъюнкции, упрощением умножения на 1 и 0, упрощением $a - a$ до 0.

В силу представленных соображений выражение представляется как тип-сумма с такими конструкторами:

- **ARand** — конструктор, представляющий оператор **rand**; единственный параметр — n в выражении **rand** $\bmod 2^n$.
- **AConst** — конструктор для числовых литералов; единственный параметр — значение литерала.
- **AVar** — конструктор для взятия значения переменной; параметр — её уникальный идентификатор.
- **AInf** — оператор конвертирования в бесконечное число, принимает один параметр — выражение, над которым выполняется операция.
- **AMod** — конвертирование в число по модулю, принимает два параметра: число бит и выражение, над которым выполняется операция.
- **APlus**, **AMinus**, **AMult**, **ADiv**, **AAnd**, **AOr**, **AEq**, **ALt** — инфиксные операторы, каждый принимает два выражения.
- **ANeg** — префиксный оператор для побитового отрицания; параметр — выражение, над отрицание которого осуществляется.

В такой нотации пример выше может быть представлен как

```
AInf (ANeg
  (AEq
    (AAnd
      (AConst 32)
      (AMod 6 (AVar 'x')))
    (AConst 0)))
```

Представление этого дерева в разработанной нотации будет таким:

```
expr_1 = expr_2{a} '|' expr_1{b} :
           (list 'AOr a b)
       / expr_2

expr_2 = expr_3{a} '&' expr_2{b} :
           (list 'AAnd a b)
       / expr_3

expr_3 = expr_4{a} '<' expr_3{b} :
           (list 'ALt a b)
       / expr_4{a} '=' expr_3{b} :
           (list 'AEq a b)
       / expr_4

expr_4 = expr_5{a} '+' expr_4{b} :
           (list 'APlus a b)
       / expr_5{a} '-' expr_4{b} :
           (list 'AMinus a b)
       / expr_5

expr_5 = expr_6{a} '*' expr_5{b} :
           (list 'AMult a b)
       / expr_6{a} '/' expr_5{b} :
           (list 'ADiv a b)
       / expr_6

expr_6 = aexpr{a} ASP 'bits' ASP mod_number{m} :
           (list 'AMod a m)
       / aexpr

expr = expr{a} LWSP op{op} LWSP expr{b} :
         (list a b op)
     / var_name :
         (list 'AVar (car param))
```

```
      / natural_number :
          (list 'AConst (car param))
      / '~' expr_1{a} :
          (list 'ANeg a)
      / '-' expr_1{a} :
          (list 'AMinus 0 m)
      / 'inf' ASP expr{a} :
          (list 'AInf a)
      / '(' LWSP expr_1{a} LWSP ')' :
          (list a)
      / 'rand' ASP 'bits' ASP mod_number{m} :
          (list 'ARand m)

var-name = '@' (ALPHA *alnum){name} :
  (identity name)

alnum = ALPHA [alnum] \ DIGIT [alnum]

op = '<' : (identity 'ALt)
   / '=' : (identity 'AEq)
   / '|' : (identity 'AOr)
   / '&' : (identity 'AAnd)
   / '+' : (identity 'APlus)
   / '-' : (identity 'AMinus)
   / '*' : (identity 'AMult)
   / '/' : (identity 'ADiv)

mod-number = non_zero_digit{st} ([natural_number]){rs} :
  (if (number? rs)
    (+ (* 10 st) rs)
    st)

natural_number = (1*DIGIT){num} :
  (string->number (string-concatenate num))
```

```
non_zero_digit = \%x31-39 :
  (string->number (car param))

ASP = WSP / CRLF WSP
```

Теперь можно проанализировать, как различные конструкции влияют на числовые промежутки, в которых находятся результаты.

- **ARand** — как и указано выше, результат **rand** — любое число в $[0; 2^n)$, где n — параметр.
- **AConst** — промежуток, который может принимать константа, — просто её значение.
- **AInf** — нет влияния на промежуток.
- **AMod** — конвертирование ограничивает промежуток до $[0; 2^n)$, где n — параметр типа. Однако не обязательно в его результате получается полный такой промежуток. Пусть все результаты внитрилежащего выражения входят в промежуток $[a; b]$. Сначала требуется обнаружить наибольший множитель n менее a по формуле $\left\lfloor \frac{a}{n} \right\rfloor \cdot n$. Далее, наименьший множитель n, больший, чем b, обнаруживается по $\left\lceil \frac{b}{n} \right\rceil n$. Теперь промежуток $\left[\left\lfloor \frac{a}{n} \right\rfloor \cdot n; \left\lceil \frac{b}{n} \right\rceil n \right)$ разбивается на промежутки длиной n. Для каждого промежутка создаётся битовый вектор длиной n, причём каждый бит равен единице, если соответствующее ему число находится в промежутке. Тогда дизъюнкция всех таких битовых векторов задаёт результат.

  Пример процедуры таков. Скажем, имеется результат, представленный значениями $[5; 7] \cup \{10\}$ и требуется сконвертировать его в число по модулю 4. Тогда минимальный промежуток, ограниченный множетелями 4 и полностью включающий исходный, — $[4; 12)$. Разбиение его на группы по четыре элемента выглядит таким образом: $[4; 8)$ и $[8; 12)$. Соответствующие им векторы — 0111 и 0010. Их дизъюнкция — 0111. Таким образом, результат — $[1; 3]$.
- **ANeg** — для каждого промежутка в области значений внутрилежащего выражения происходит поиск значения побитового отрицания его границ и их переставление.

  Например, если $[5; 7] \cup [10; 14]$ имеет тип числа по модулю 32, то эти числа можно представить как $[00101_2; 00111_2] \cup [01010_2; 01110_2]$. Применение за-

данного правила даёт промежуток $[\ 00111_2;\ 00101_2] \cup [\ 01110_2;\ 01010_2]$, то есть $[11000_2; 11010_2] \cup [10001_2; 10101_2]$, что в десятичной системе счисления выглядит как $[24; 26] \cup [17; 21]$.

- **AEq** — сравниваются границы операндов. Если их пересечение не пустое, то **AEq** может быть 1, в противном случае оно всегда даёт 0. Если же оба операнда могут принимать лишь одно и одинаковое значение, то **AEq** всегда 1 и не может быть 0.
- **ALt** — следуя схожей логике, если наибольшее возможное значение левого операнда меньше наименьшего возможного значения правого, то **ALt** всегда 1; если наименьшее возможное значение левого операнда больше, чем наибольшее возможное значение правого, то **ALt** всегда 0; в противном случае, оно может быть как 0, так и 1.
- **APlus** и **AMinus** — если в область допустимых значений левого операнда входит $[a_1; a_2]$, а в область значений правого — $[b_1; b_2]$, то в область значений результата при сложении входит $[a_1 + b_1; a_2 + b_2]$, при вычитании — $[a_1 - b_2; a_2 - b_1]$.
- Все остальные операции — арифметические и побитовые от двух переменных, и для них нужно взять картезианское произведение областей значений операндов и применить оператор к каждому результирующему кортежу.

Посмотрим, как эти правила применимы к приведённому выше примеру, если `@x` находится в промежутке $\{0\} \cup [100; 127]$ (в целях данного обсуждения можно проигнорировать принятое ранее решение использовать идентификаторы операторов **rand** вместо конкретных промежутков).

```
inf ~ ((32 & @x bits 6) = 0)
inf ~ ((32 & {0}, [100; 127] bits 6) = 0)
inf ~ ((32 & {0}, [36; 63]) = 0)
inf ~ (({0}, {32}) = 0)
inf ~ [0; 1]
inf [0; 1]
[0; 1]
```

Зачем нужен этот механизм, если в любом случае требуется перебрать все значения случайных чисел? Причина в том, что зачастую конкретный идентификатор **rand** используется лишь в одной части выражения. Тогда не имеет значения, что он является случайным, и можно принять его за набор значений.

К примеру, рассмотрим выражение

(@x + 2) * (@y − 5) < @z + @a

Предполагается, что ни одна переменная не имеет общих идентификаторов **rand** с другой. Пусть каждая из них имеет значение $[0; 2^{32})$. Тогда подход, требующий использования честных идентификаторов **rand**, привёл бы к $\left(2^{32}\right)^4$ итерациям. С другой стороны, использование операций, преобразующих области значений напрямую, потребовало бы лишь $\left(2^{32}\right)^2$ итераций — квадратный корень количества итераций первого подхода.

Теперь необходимо рассмотреть возможность того, что арифметическая операция, выполненная над числом по модулю, приведёт к переполнению или произойдёт деление на ноль. На этот случай нужно отметить набор значений как потенциально некоректные. Можно спроектировать произвольной сложности механизмы, которые оповещали бы пользователя анализатора о возможной ошибке. Однако все они могут быть построены на очень простой концепции хранения булевого значения, описывающего возможность ошибки и проходящего до вершины дерева, представляющего выражение; так, флаг ошибки установлен для узла, если он есть хотя бы у одного из дочерних.

Итоговый алгоритм выглядит так:

а) Выполнение высокоуровневых оптимизаций: сокращения, приведения к простой для анализа форме путём использования ассоциативности, коммутативности, дистрбутивности или иных свойств. Описание этих деталей не приводится, поскольку они уже далеко от заявленной темы, а также потому, что ожидается, что прирост скорости, связанный с использованием этих оптимизаций, незначительный на практически используемых программах. Пусть возможно написать код, который вычитает переменную из неё же, в настоящих программах такая функциональность используется нечасто. Когда идентификатор **rand** встречается в выражении несколько раз, это редко можно решить примитивными оптимизациями.

б) Построение списка всех идентификаторов случайных чисел, используемых в выражении, и подсчёт количества их вхождений.

в) Проход дерева выражения для каждого идентификатора и отметка поддеревьев, в которых он присутствует.

г) Повторный проход по дереву с использованием меток и замена первого узла, у которого все операнды помечены одинаковым идентификатором, на

соответствующие области значений; возможно, в терминах других случайных чисел. На этом же этапе происходит отметка некорректных операций.

д) Когда все идентификаторы случайных значений уничтожены, использование более эффективного алгоритма для обнаружения области значений выражения целиком. Если при этом возникает ошибка, происходит установка флага ошибки у результата.

2.5 Императивные присваивания

Теперь, когда имеются выражения, можно составить язык, позволяющий строить цепочки из операторов присваивания. Пример программы на таком языке может быть таким:

```
@x = rand mod 16;
@y = @x * 2;
@z = @x & @y
```

В первую очередь возникает вопрос того, как представлять состояние переменных. Частично ответ был представлен в прошлой секции: переменные определяются как операции над случайными значениями.

Для этого нужно заменить все операторы определения значения переменных на выражения, сохранённые в качестве значений этих переменных, поскольку требуется императивное, а не реактивное, программирование и не нужно, чтобы при изменении значения переменной значения остальных, зависящих от неё, переменных тоже сменили результат. И, как отмечалось ранее, операторы **rand** заменяются на уникальные идентификаторы.

В силу этого можно использовать для хранения значений переменных тип, заданный для описания выражений, с тем исключением, что **AVar** не должен присутствовать, а для **ARand** требуется хранить уникальный идентификатор соответствующей случайной величины. Решено сделать **ARand** параметрическим и в оригинальном определении, но на стадии построения дерева выражений задавать произвольное значение и игнорировать его, пока оно не заменится на корректный идентификатор на стадии анализа.

Теперь определение языковых конструкций — задача прямолинейная.

- **CAsgn** — операция присваивания. Принимает два параметра: уникальный идентификатор переменной и присваиваемое выражение.

- **CSeq** — построение цепочки присваиваний. Принимает в качестве параметров два высказывания, которые требуется объединить в цепочку.

К примеру, рассмотренный выше пример может быть представлен таким образом:

```
CSeq (CAsgn 'x' (AMod ARand 10))
     (CSeq (CAsgn 'y'
                  (AMult (AVar 'x')
                         (AConst 2)))
           (CAsgn 'z'
                  (AAnd  (AVar 'x')
                         (AVar 'y'))))
```

Поначалу может быть неясным, зачем требуется отдельная конструкция для построения цепочки выражений, когда можно было просто держать их список. Однако последующее расширение языка структурами для контроля исполнения программы — циклами, условными операторами — предполагает унифицированное обращение с единичными высказываниями и их списками. Можно было бы вместо **CSeq** использовать конструктор, который принимает не два параметра, а один список. Но беглое рассмотрение приводит к выводу, что **CSeq** и так представляет собой связный список.

Теперь нужно формализовать операционную семантику языка.

Вводится среда исполнения, которая представляет собой ассоциативный контейнер, сопоставляющий переменные их значениям. В начальный момент времени среда исполнения пустая. В целях данного обсуждения выбор, что предпринимать, если программа обращается к переменной, которая ещё не установлена, не имеет значения, поэтому можно просто объявить таким программы некорректными.

Далее необходимо хранить список значений переменных в каждый момент исполнения, чтобы определить максимальное и минимальное значение каждой из них. Это приводит к ещё одному ограничению: запрещено присваивать в переменную значения типа, отличного от того, что в ней хранилось ранее. Например, если переменная x была числом по модулю 256, то в дальнейшем она не может стать числом по модулю 16 или бесконечным числом.

Определим функцию вычисления, которая принимает дерево, представляющее программу, и среду разработки и рекурсивно вызывает себя, обрабатывая

дерево согласно таким правилам:

- **CAsgn**
  а) Заменить каждый **ARand** на такой, в котором указан корректный уникальный идентификатор.
  б) Пройти по присваемому выражению, заменяя каждый **AVar** на значение соответствующего идентификатора в текущей среде.
  в) Выполнить над результирующем выражением алгоритм, описанный в предыдущей секции. Добавить полученную область в список допустимых для заданной переменной значений.
  г) Проверить бит ошибки; если он установлен, оповестить пользователя.
  д) Поместить результат в среду под именем переменной, присваивание которой осуществляется.
- **CSeq**
  а) Пусть текущая среда имеет имя e_1. Тогда нужно взять первое поддерево и выполнить функцию вычисления на нём со средой e_1 и получить новую среду; пусть она называется e_2.
  б) Взять второе поддерево, запустить функцию вычисления на нём со средой e_2 и получить новую среду e_3.
  в) Вернуть e_3 как новую среду.

Несложно убедиться, что **CSeq** — ассоциативная операция, то есть для любых высказываний a, b, c выполняется CSeq a (CSeq b c) = CSeq (CSeq a b) c. Вследствие этого можно строить дерево, представляющее программу, произвольно организуя дерево **CSeq**.

ABNF результирующего языка такое же, как у предыдущего языка, но с дополнительным правилом:

⟨*statement*⟩ ::= ⟨*var-name*⟩ LWSP ’=’ LWSP ⟨*expr*⟩
| ⟨*statement*⟩ LWSP ’;’ LWSP ⟨*statement*⟩

Для хранения значений переменных нужно использовать механизм передачи среды, определённый в нашей нотации. Значения переменных будем хранить в списке из пар вида «ключ-значение».

Введём функцию `lookup` для поиска значений в такой среде:

```
(define (has-key key)
        (lambda (rec) (string=? key (car rec))))
```

```
(define (lookup env x)
        (car (cdr (find (has-key x) env))))
```

Запись значения в среду тогда является просто функцией `cons`. При этом даже не требуется отбрасывать старые значения: так как они будут дальше по списку, процедура `lookup` не будет до них доходить и вернёт актуальное значение.

```
statement
  = var_name{name} LWSP '=' LWSP expr{value} :
      (begin (set! env (cons '(name value) env))
             env)
  / statement{:0} LWSP ';' LWSP statement{:1} :
      (identity env)
```

Явным образом возвращается среда затем, чтобы в дальнейшем, когда появится необходимость комбинировать разные высказывания, можно было передавать среды, возвращённые некоторыми из них, как значения, не зависящие от текущего состояния основной среды.

2.6 Условные операторы

Пусть этот язык уже можно использовать, в него несложно добавить и условные операторы, что сделает его более похожим на привычные.

Для начала определим новое высказывание **CIf** с тремя параметрами: условным выражением и двумя подвысказываниями. Операционная семантика такова:

а) Вычислить значение условного выражения в текущей среде;

б) Если результат равен 0, вычислить второе высказывание с той же средой. В противном случае, вычислить первое. Результат вычисления **CIf** тот же, что у выбранного высказывания.

Пример программы, основанной на **CIf**:

```
@a = rand mod 16;
@b = rand mod 32;
if (@b mod 16 == @a) then
        @c = @a mod 32 * 2
```

```
else
        @c = @a mod 32 + 1
fi;
@d = @c == 0
```

Несложно заметить, что отдельной области видимости для переменных, инициализированных внутри **CIf**, не вводится. Это может привести к проблеме. Если до **CIf** переменная не была инициализирована и в одной из веток, но не в другой, произошло её присваивание, то программа может быть корректной в одном наборе случаев, но не в другом. В силу этого такие программы объявляются недопустимыми.

Теперь определим такой механизм для проведения анализа **CIf**:

а) Для каждого операнда-высказывания найти все переменные, которые там устанавливаются.

б) Для каждого элемента объединения этих множеств поместить новое значение переменной в среду:
   - Он установлен в обеих ветках и имеет значение c_t в первой и c_e во второй. Тогда результирующее значение в **CIf** — $a \cdot c_t + (a = 0) \cdot c_e$, где a — условие.
   - Он установлен в первой ветке, но не во второй. Тогда результат — $a \cdot c_t + (a = 0) \cdot c_o$, где c_o — значение переменной в начальной среде.
   - Он установлен во второй ветке, но не в первой. Тогда значение равно $a \cdot c_o + (a = 0) \cdot c_e$.

в) Сохранить списки возможных значений, полученные при проходе высказываний, в соответствующие переменным хранилища.

К примеру, предполагая, что переменные `@a` и `@b` заданы, рассмотрим простую релизацию **max**, заданную таким образом:

```
if (@a < @b) then
        @c = @b
else
        @c = @a
fi
```

Анализ раскрывает, что `@c` имеет значение $(a < b) \cdot b + (a \geq b) \cdot a$.

Теперь нужно выразить эту конструкцию в представленной нотации.

В первую очередь вводится функция `key-set`, которая получает из среды

отсортированный список ключей, которые в ней есть.

```
(define (key-set env)
  (delete-duplicates (sort string<? (map car env))))
```

Наконец, описание правила `statement` расширяется таким образом:

```
statement
  = ... // old definitions
  / 'if' ASP '(' LWSP expr{c:1} LWSP ')' ASP
         'then' ASP statement{t:2}   ASP
         'else' ASP statement{e:3:1} ASP 'fi' :
    (let ([st (key-set t)] [se (key-set e)])
         (if (eq st se)
           (begin
              (set! env (map (lambda (x)
                 (let ([va (lookup t x)]
                       [vb (lookup e x)])
                      '(APlus
                         (AMult
                           (ANeg
                             (AEq (AConst 0)
                                  c))
                           va)
                         (AMult
                           (AEq (AConst 0) c)
                           vb)))) st))
              env)
           (error "if"
              "Different number of variables")))
```

2.7 Заведомо завершающиеся циклы

Для выражения подавляющего большинства полезных программ в императивном стиле необходимы циклы. Однако они являются и серьёзным источником проблем: в общем виде они вводят возможность того, что программа никогда не

завершится. Более того, доказано[8], что не может существовать метода определения, какие программы завершаются, а какие нет, в общем случае. Рассуждения можно проводить лишь об определённых классах программ.

В рамках данной работы остановимся на языке, который даёт выражать лишь императивный эквивалент тотального функционального программирования[9]. Основная его особенность в том, что возможно осуществлять обработку только тех данных, которые гарантировано деконструируются за конечное число шагов. Так как представленный язык поддерживает только числа, это означает, что вводятся лишь циклы с иммутабельным счётчиком.

Таким образом, вводится высказывание **CFor** с тремя параметрами: именем переменной-счётчика, выражением, задающим число шагов, и телом цикла.

Семантика высказывания такова:

а) Сохранить число шагов и присвоить 1 переменной в счётчике;
б) Если переменная больше, чем количество шагов, покинуть цикл;
в) Получить среду для тела цикла;
г) Присвоить переменной-счётчику номер следующего шага;
д) Перейти к шагу 2.

Явное отличие от того, как обычно используются переменные-счётчики, заключается в том, что изменение переменной в теле цикла не меняет хода работы программы: количество пройденных и оставшихся шагов хранится внутри и не доступно для изменения, только для чтения. Это даёт уверенность, что программы всегда завершатся.

Пример программы, использующей такой цикл:

```
@j = 0;
for @i to (@n) do
        @j = @j + @i
done
```

Эта программа высчитывает сумму первых `@n` шагов.

Циклы можно анализировать, но это не тривиальная задача. Циклы из нашего подмножества можно представить как рекурсивные определения, и некоторые из них имеют так называемые «закрытые формы», то есть арифметические выражения, которые дают функционально тот же результат, что и последние присваивания в цикле. К примеру, для цикла, представленного сверху, закрытая форма такова:

@i = @n;
@j = @i * (@i + 1) / 2

Существует большое количество методов нахождения закрытых форм рекурсивных определений (конкретные примеры есть в «Конкретной математике»[11]), но они не позволяют осуществлять такой анализ автоматически в общем виде. Лучшее, что можно сделать в таких случаях, — сверять выражения в телах циклов с некоторой библиотекой популярных закрытых форм. Это выходит за рамки данной работы.

Таким образом, для анализа циклов происходит следующее:

а) Определение границ количества шагов (пусть n и N обозначают минимальное и максимальное количество итераций);
б) Проход цикла $n - 1$ раз;
в) Проход цикла $N - n + 1$ раз с сохранением значений всех переменных, присвоенных в телах циклов, в виде кортежа (x, i, e), где x — имя переменной, i — номер итерации, а e — значение переменной по исходу итерации;
г) Составление выражения $\sum_i (i = n) \cdot e_i$ для каждой переменной, которая присваивалась в теле цикла, из сохранённых данных, где e_i — значение переменной в итерации i.

Здесь появляется принципиальная проблемая. Дело в том, что разработанная нотация не позволяет менять среду неизвестное заранее число раз за одно правило. Максимальное число правок среды по ходу исполнения одной альтернативы равно число операндов конкатенации верхнего уровня.

Таким образом, в этой нотации нельзя напрямую выразить идею применения одного и того же правила — в данном случае, правила `statement` — неизвестное заранее число раз. Тело цикла будет обработано лишь единожды при обработке текста, а значит, и вызов соответствующей функции произойдёт лишь однажды.

Поэтому как только программа обретает возможность использовать терм заранее неизвестное число раз, возникает потребность в хранении построенного синтаксического дерева и дальнейшего его использования.

Одним из решений является именно такое: в высказываниях представлять не вычисления, а механизм построения дерева, а обработку производить лишь тогда, когда всё дерево уже построено. Для этого введём функцию, которую назовём `traverse-statements`. Она будет принимать такое дерево и про-

изводить все действия, которые ранее были возложены на обработчики для `statement`. Соответственно, для всех случаев, кроме циклов, код уже написан и повторно не приводится. Общий вид функции и код для цикла будут таковы:

```
(define (traverse-statements env)
  (case-lambda
    (['CNop]       env)
    (['CIf c t e] (traverse-statements-cif env c t e))
    (['CSeq a b]  (traverse-statements-cseq env a b))
    (['CAsgn k v] (traverse-statements-casgn env k v))
    (['CFor vn c bd] (let ([(min max)] (find-range c))
      (do (foldl (lambda (e k)
                   (traverse-statements e
                     (list 'CIf c
                       (list 'CSeq
                         (list 'CAsgn v c) bd) 'CNop)))
                 env (iota max)))))))
```

Тогда грамматика будет выглядеть просто:

```
statement
  = var_name{name} LWSP '=' LWSP expr{value} :
      (list 'CAsgn name value)
  / statement{c1:0} LWSP ';' LWSP statement{c2:1} :
      (list 'CSeq c1 c2)
  / 'if' ASP '(' LWSP expr{c:1} LWSP ')' ASP
         'then' ASP statement{t:2}   ASP
         'else' ASP statement{e:3:1} ASP 'fi' :
      (list 'CIf c t e)
  / 'for' ASP var_name{vn} ASP
         'to' ASP '(' ASP expr{c} ASP ')' ASP
         'do' statement{bd} 'done' :
      (list 'CFor vn c bd)
```

Такое решение, однако, приводит к тому, что нотация оказывается бесполезной: в самом деле, раз вычисления в любом случае осуществляются после построения дерева, нет никаких причин не использовать более устоявшиеся меха-

низмы описания алгоритмов и грамматик по отдельности. Попытки расширить нотацию до того, чтобы она покрывала рекурсивные случаи, тоже оказывается нецелесообразно: внесение такой сложности делает результирующие тексты трудными для понимания, а нотация теряет интуитивность и требует специальной тренировки для написания на ней.

В связи с этим замечается ещё раз, что нотация в первую очередь предназначена для линейных проходов по исходным текстам программ, при которых каждый терм затрагивается не более одного раза. Довольно обширный круг задач, решённый до момента столкновения с циклами, показывает, что даже с такими ограничениями существует широкое применение разработанной нотации, однако не следует забывать и о том, что подавляющее большинство средств для работы с формальными языками этой нотацией не упраздняется, а дополняется.

2.8 Дальнейшие расширения

Результат алгоритма анализа можно сделать более показательным, если сохранять с каждым значением его вероятность. Это позволит определить, какие ветки исполнения посещаются чаще.

Также возможно ввести частичный порядок на множестве возможных значений, такой, что операция сравнения определяется близостью максимального значения к нулю. Это бы позволило рассуждать о более широком классе циклов и определять, завершатся ли они.

Можно определить функции как отдельные программы с некоторыми начальными значениями переменных и заранее заданными типами, а также добавить выражение для вызова функций.

Сложные структуры можно хранить как набор переменных, то есть работать с ними как с синтаксическим сахаром.

Знаковые числа можно реализовать, если у чисел по модулю один бит отвести под знак. Это потребует только добавления новых арифметических правил.

Общий случай для циклов может быть подвержен частичному анализу, если найти переменные, которые меняются с заданным периодом, и определить область значений на одном периоде. Этот метод рассуждения может применяться и к группам переменных, которые образуют компоненту сильной связности по отношению взаимного изменения. Переменные, период которых найти нельзя, в

общем случае не могут рассматриваться.

ЗАКЛЮЧЕНИЕ

Была введена нотация, которая позволяет описывать вычисления над конструкциями языка в совокупности с его грамматикой и состоит из известных в среде языковых исследований существующих формальных систем: описания грамматики ABNF и языка программирования Scheme, диалекта Lisp.

Было продемонстрировано, как введённая нотация может быть использована для описания алгоритма статического анализа, и на этом примере показан итеративный ход разработки, при котором эта нотация способна при частично реализованном алгоритме способствовать тестированию и начальной проверке корректности.

Предполагается, что эта нотация способна найти применение во многих сферах деятельности, в которых используется языко-ориентированный подход: в разработке программного обеспечения, в исследовании алгоритмов над формальными языками, при распространении результатов этих исследований и во многом другом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Niklaus Wirth, “What can we do about the unnecessary diversity of notation for syntactic definitions?“ / Никлаус Вирт. - Communications of the ACM, Volume 20 Issue 11, 1977. - с. 822-823.
2. Daan Leijet, “Parsec, a fast combinator parser“ / Дэн Лэйет. - 2001. - Режим доступа:
   https://hackage.haskell.org/package/parsec
3. “Grammars“ / Стандартная библиотека Perl6. - 2016. Режим доступа:
   https://docs.perl6.org/language/grammars
4. “Syntax extensions and interpretation scopes“ / Руководство к языку Coq. - 2001. - Режим доступа:
   http://flint.cs.yale.edu/cs428/coq/doc/Reference-Manual013.html
5. RFC 4234 “Augmented BNF for Syntax Specifications: ABNF“. - 2005. - Режим доступа:
   https://tools.ietf.org/html/rfc4234
6. «The Revised^6 Report on the Algorithmic Language Scheme.» / Майкл Спербер, Р. Кент Дибвиг, Мэттью Флэт, под редакторством Антона ван Штратена, - 2007. - Режим доступа:
   http://www.r6rs.org/
7. Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catalin Hritcu, Vilhelm Sjöberg, and Brent Yorgey, “Software Foundations“ / Бенджамин Пирс и другие. - 2016. - Версия 4.0. - Режим доступа:
   http://www.cis.upenn.edu/ bcpierce/sf.
8. Alan Turing, On “computable numbers, with an application to the Entscheidungsproblem“ / Алан Тьюринг. - Proceedings of the London Mathematical Society, Series 2, Volume 42 (1937), pp 230–265, doi:10.1112/plms/s2-42.1.230.
9. Turner, D.A., “Total Functional Programming“ / Тюрнер Д. А. - Journal of Universal Computer Science, 10 (7): 751–768. - 2004.
10. Birch, Johnnie; van Engelen, Robert; Gallivan, Kyle, “Value Range Analysis of Conditionally Updated Variables and Pointers“ / Бирш, ван Энгелен, Галливан.

- Режим доступа:
http://www.cs.fsu.edu/ engelen/cpcpaper.pdf

11. Ronald L. Graham, Donald E. Knuth, Oren Patashnik, “Concrete Mathematics: A Foundation for Computer Science“ / Грэхэм, Кнут, Паташик. - Второе издание. - Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1994.
12. Patrick Cousot, Nicolas Halbwachs, “Automatic discovery of linear restraints among variables of a program“. In Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York.
13. Ilya Sergey, “Programs and Proofs“ / Илья Сергей. Режим доступа: http://ilyasergey.net/pnp/