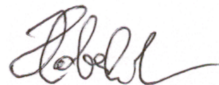


Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И. Ульянова (Ленина)» (СПбГЭТУ «ЛЭТИ»)

На правах рукописи



**Хаберланд Рене**

**Логический язык программирования как  
инструмент  
спецификации и верификации  
динамической памяти**

**05.13.11 – Математическое и программное обеспечение  
вычислительных машин, комплексов и компьютерных сетей**

Диссертация на соискание учёной степени  
кандидата технических наук

Научный руководитель  
к.т.н. Кринкин Кирилл Владимирович

Санкт Петербург — 2020

# Оглавление

<b>1</b>	<b>Введение</b>	<b>3</b>
1.1	Вычисление Хора . . . . .	12
1.1.1	Тройка Хора . . . . .	13
1.1.2	Логический вывод . . . . .	17
1.1.3	Автоматизация логического вывода . . . . .	25
1.1.4	Альтернативные подходы . . . . .	33
1.2	Объектные вычисления . . . . .	40
1.3	Модели динамических куч . . . . .	54
1.3.1	Преобразование в стек . . . . .	54
1.3.2	Анализ образов . . . . .	54
1.3.3	Ротация указателей . . . . .	55
1.3.4	Файловая система . . . . .	56
1.4	Побочные области . . . . .	58
1.4.1	Анализ псевдонимов . . . . .	58
1.4.2	Сбор мусора . . . . .	59
1.4.3	Интроспекция кода . . . . .	61
1.5	Существующие среды . . . . .	62
<b>2</b>	<b>Проблемы динамической памяти</b>	<b>66</b>
2.1	Мотивация . . . . .	68
2.2	Проблемы в связи с корректностью . . . . .	72
2.3	Проблемы в связи с полнотой . . . . .	75
2.4	Проблемы в связи с оптимальностью . . . . .	77
<b>3</b>	<b>Выразимость формул куч</b>	<b>79</b>
3.1	Граф над кучами . . . . .	83
3.2	Предикаты . . . . .	89
<b>4</b>	<b>Логическое программирование и доказательство</b>	<b>99</b>
4.1	Пролог как система логического вывода . . . . .	99

4.2	Логический вывод как поиск доказательства . . . . .	107
4.3	Совместимость языков . . . . .	121
4.4	Представление знаний . . . . .	123
4.5	Архитектура системы верификации . . . . .	126
4.6	Объектные экземпляры . . . . .	130
<b>5</b>	<b>Ужесточение выразимости куч</b>	<b>132</b>
5.1	Мотивация . . . . .	133
5.2	Многозначимость операторов . . . . .	135
5.3	Ужесточение операторов . . . . .	139
5.4	Классовый экземпляр как куча . . . . .	148
5.5	Частичная спецификация куч . . . . .	151
5.6	Обсуждения . . . . .	153
<b>6</b>	<b>Автоматическая верификация с предикатами</b>	<b>156</b>
6.1	Сжатие и развёртывание . . . . .	157
6.2	Предикатное расширение . . . . .	158
6.3	Предикаты как логические правила . . . . .	160
6.4	Интерпретация предикатов над кучами . . . . .	166
6.5	Перевод правил Хорна . . . . .	169
6.6	Синтаксический перебор как верификация куч . . . . .	170
6.7	Свойства . . . . .	174
6.8	Реализация . . . . .	175
6.9	Выводы . . . . .	194
<b>7</b>	<b>Заключение</b>	<b>198</b>
	<b>Список сокращений</b>	<b>204</b>
	<b>Список терминов</b>	<b>205</b>
	<b>Литература</b>	<b>221</b>
	<b>Список определений</b>	<b>246</b>
	<b>Список иллюстраций</b>	<b>249</b>
	<b>Приложение: Предметный указатель</b>	<b>252</b>

# 1 Введение

## Актуальность

Ошибки использования динамической памяти являются одними из самых дорогостоящих при разработке программного обеспечения на протяжении уже нескольких десятилетий. Локализация таких ошибок является трудной задачей, поскольку, довольно часто, видимое проявление некорректного поведения программы и реальный участок исходного кода его вызвавшего отстоят далеко друг от друга. Следовательно, разработка подходов и методов, упрощающих поиск и предотвращение ситуаций некорректного использования динамической памяти является актуальной задачей.

Выразимость утверждений, полнота правил, и автоматизация в большой степени определяют успех верификации динамической памяти. Выразимость утверждений касается формализма утверждений и правил верификации. В существующей на текущий момент практике, описания динамической памяти сложны и часто неинтуитивны. Возможности логического вывода зачастую ограничиваются корректностью и полнотой правил верификации и представляют собой лишь «*островные реализации*», которые нельзя использовать на практике. Из-за большого количества соглашений (конвенций) в языках спецификации и верификации возможности верификации динамической памяти сильно ограничены.

Верификация часто осуществляется нетривиально, объяснения принятия или отказа доказательства, как правило, контр-интуитивны.

Диссертационная работа опирается на фундаментальные и прикладные исследования таких ученых как: Floyd R.W., Hoare C.A.R., Burstall R.M., Kowalski R.A., Clarke E.M. Jr., Apt K.R., Suzuki N., Warren D.H.D., Horwitz S., Miller B.P., Fredriksen, L. So, B., Steinbach J., Tofte M., Talpin J.-P., Abadi M., Hutton G., Reynolds J.C., Berdine J., Calcagno C., O’Hearn P.W. Clarke E.M. Jr., Parkinson M.J., Burstall R.M., Hurlin C., Distefano D., Birkedal L., Matthews C., Bertot Y., Gregoire B., Leroy X., Dodds M., Jacobs B., Smans J., Philippaerts P., Vogels F., Penninck W., Piessens F. Bornat R., Meyer B., Parduhn S., Wilhelm R. Leino K.R.M.

## Цель работы

Повышение степени выразимости средств описания вариантов использования динамической памяти за счёт исключения многозначности между языками спецификации и верификации, а также разра-

ботка программных средств автоматической верификации.

Работа была осуществлена при поддержке программы №2.136.2014/К от министерства образования и науки Российской Федерации.

## Задачи исследования

Для достижения поставленной цели в диссертационной работе решаются следующие задачи:

1. Исследование и анализ существующих ограничений выразимости и выявление причин несоответствия языков спецификации и верификации динамической памяти.
2. Исследование декларативности в утверждениях, которые основываются на предикатной логике без избыточных конвенций и анализ реляционной модели, основанной на ужесточении пространственных операторов и абстрактных предикатов, для описания куч (от англ. *Heap*<sup>1</sup>). Сравнение реляционной модели предикатов с другими моделями.
3. Исследование критериев реализуемости верификации динамической памяти на основе выбранного логического языка программирования (использующего абстрактную машину Уоррена) и автоматизированного вывода над абстрактными предикатами. Исследование возможностей и ограничений при представлении куч, а также правил верификации с помощью термов.
4. Разработка архитектуры верификатора динамической памяти и его программная реализация.
5. Исследование представления куч в таком унифицированном виде, который даёт возможность обнаружить синтаксически различные, но семантически одинаковые представления и свести число необходимых сравнений к минимуму. Исследование возможности отделения выражений куч от остальных логических правил и возможность подключения «SAT»-решателей для решения отдельных теорий.
6. Исследование возможности автоматического сравнения динамической памяти с имеющимся экземпляром памяти при условии изменения входных условий или представления модели памяти.

## Методы исследования

Автоматизированное доказательство теорем, вычисление Хора, формальная логика, вычислимые логики, семантики программ, теория типов, теория объектов, логическое программирование, функциональное программирование, объектно-ориентированное программирование, формальные языки,

---

<sup>1</sup>В работе используется русское название структуры данных „куча“, предложенной Дж. Вильямсом (J. W. J. Williams) в 1964

теория языков программирования, теория программирования, теория графов,  $\lambda$ -вычисления, абстрактная алгебра, теория вычислимости, языковые процессоры, теория компиляции, логика распределенной памяти, архитектуры процессоров ЭВМ, организация памяти процессов ОС, встроенные системы, современные технологии моделирования, представление и обработка знаний.

## Научная новизна работы

### В области автоматизации доказательств:

1. Впервые исследован разрыв между языками спецификации и верификации динамической памяти. Впервые предложен логический язык программирования как инструмент, который способен преодолеть все выявленные проблемы выразимости, автоматизации и полноты с помощью унификации языков.
2. Впервые проведены сравнения выразимости при трансформации термов в логическом и функциональном представлениях, выявлено с помощью множества примеров при использовании метрик, что логический язык практически без исключений превосходит представления функционалов. Альтернативно к функционалам рассматривались императивные программные операторы с побочными эффектами. Оба случая встречаются в имеющихся подходах от Reynolds, Parkinson, Berdine, Hurlin, Jacobs, Tofte, Hutton и Bertot, они приводили к разрыву языков спецификации и верификации.
3. Впервые предложено, термы диалекта языка Пролог использовать непосредственно для представления данных полного конвейера статического анализа динамической памяти, в отличие от используемого в настоящее время комбинированного подхода Meyer, Leroy, либо подхода основанного на 3-адресном представлении от Reynolds/Berdine, O'Hearn, Bornat, либо подходы основаны на ассемблере как у Parkinson.
4. Предложен новый метод автоматизации процесса верификации памяти, как синтаксического перебора абстрактных предикатов, при использовании генеричных интерпретаторов основанных на пошаговой обработке граней графа кучи, в отличие от (полу-)ручного преобразования структур куч как у Berdine, O'Hearn, а также в отличие от введения определений узкого круга тактик верификации Bertot и Jacobs.

### В области выразимости языков спецификации и верификации:

1. Сняты ограничения выразимости переменных символов, термов и рекурсивных предикатов, удовлетворяющих правилам Хора для спецификации и верификации динамической памяти, в отличие от Reynolds, Berdine, Bornat и Parkinson/Hurlin. Ограничения связаны, например, с использованием как императивной переменной вместо логического символа, а также в связи

с логическим представлением правил и его выводом как доказательством теоремы над динамической памятью. Кроме того, сняты ограничения вывода встроенных предикатов, таким образом, могут произвольно анализироваться определённые абстрактные предикаты во время запуска программы.

2. Повышена выразимость утверждений в отношении структуры куч за счёт ужесточения операций над кучами, использовавшиеся ранее и имеющие неоднозначные трактовки в процессе доказательства, в отличие например от Reynolds, Jones/Hosking, Cormen/Leiserson, Atallah, Khedker, Muchnik и Pavlu.

Впервые установлены свойства моноида и группы, позволяющие производить вычисления над кучами одним проходом и без анализа потенциально всех под-выражений динамической памяти, в отличие от Suzuki, Leino и Berdine, а также в отличие от Abadi и Cardelli, которые не вводят указателей и чьи вычисления строятся на мало интуитивных алгебраических кольцах с недостатком неполноты и проблемой локальности. Предложено расширить «*UML/OCL*» с указателями с ужесточённой моделью.

3. Достигнуто повышение выразимости куч и полноты правил, за счёт введения частичного оператора «*\_*», которое заменяет переменную или любую часть терма кучи, в отличие от интерпретируемых предикатов «*false*» и «*true*» от Reynolds, Berdine/O'Hearn, а также в отличие от предложенных изменений входного языка Clarke, Apt и Tofte/Talpin.

## Теоретическая значимость работы

1. Разработана обобщенная архитектура верификатора динамической памяти на основе конвейера, которая может быть использована и подключена в существующие анализы на основе Пролог-термов как центральное промежуточное представление.
2. Верификация динамической памяти на основе логического языка программирования является более адекватным представлением.
3. Абстрактные предикаты могут распознаваться автоматически, т.е. с помощью обобщенного подхода и без применения тактик.
4. Предложено определение единичной кучи для сокращения многозначности. Предложено вычисление для лучшего сравнения куч, а также для улучшения качества программы на более ранней стадии для возможного включения в моделирования с UML.
5. Сводимость языков верификации и спецификации динамической памяти.

## Практическая значимость работы

При автоматизированной поддержке вывода с повышенной выразимостью теоретически обоснованный и разработанный прототип на основе диалекта Пролога позволяет проводить верификацию проще и короче.

С практической точки зрения, добавление новых языковых возможностей языка программирования приводит к новым проблемам между спецификацией и верификацией куч. таким образом: (1) необходимо проводить спецификации исключительно на логическом/декларативном языке без побочных эффектов, например близком к Прологу; (2) любое представление на языке спецификации должно обрабатываться элементами языка верификации, какова бы ни была система логического вывода; (3) языки спецификации и верификации имеют сильные пересечения, и поэтому унификация обоих языков упрощает оба процесса.

Разработанные в рамках диссертации решения (платформа для верификации, далее – Платформа) разрешают добавлять новые фазы по обработке динамической памяти, менять существующие и переводить данные Си-программы в термовое представление. В качестве входного языка может быть использован любой другой язык, в том числе формально пустой язык. К термовому представлению, которое также может меняться, могут добавляться новые элементы. Платформа, предложенная на основе термового представления является открытой.

Теории о кучах можно добавлять произвольно, в том случае если они позволяют синтаксический перебор. Теории не о кучах также могут быть включены в SAT-решатели.

Предложенное ужесточение куч позволяет решить проблему полноты на основе «неполного определения синтаксиса», сравнивать кучи и сокращать их описания. Правила состоят из троек Хора, которые определяют кучи. Если упростить сравнение соседних куч таким образом, что анализируются только переходные кучи, то правила в общем случае упрощаются. Проверка спецификации также даёт возможность сравнивать входную и минимальную программы за счёт выявленного графа кучи. Если автоматизированное доказательство данной теоремы о куче осуществить не удаётся, то унификация термов Пролога автоматически выявит максимально обобщенный контр-пример без дополнительных затрат. Ужесточение позволяет последовательно анализировать подкучи и соответствующие подформулы без возврата и нового поиска. Отпадает анализ всех конъюнкций. Если данный набор абстрактных предикатов перебирается распознавателям  $LL(k)$  или  $LR(k)$  и т. д., то имеется (положительное или отрицательное) доказательство. Практическим компромиссом считается переоформление правил, которое допускается ради универсальности подхода.

Разработанный диалект Пролога позволяет повысить выразимость утверждений куч, автоматизировать абстрактные предикаты и упростить спецификацию куч, что в конечном итоге повысит эффективность использования разработанного программного средства при решении практических задач.



## Степень достоверности результатов

Достоверности результатов обеспечивается логическими, формальными выводами и доказательствами представленных в диссертационной работе теорем, а также выполненной реализацией прототипа. Качество предложенных решений, в том числе универсальных, подтверждается проведённой тщательной экспертизой сравнимости термов на более чем 80 специально подобранных примеров из большого числа типичных заданий.

Кроме использованных формальных методов, корректность логического вывода основывается на семантическом анализе, который исключает возможность недопустимых синтаксических и семантических ошибок при предложенной в работе архитектуре конвейера.

Достоверность основных результатов, полученных в диссертационной работе, подтверждается также их апробацией на различных международных и российских конференциях.

## Реализация результатов работы

На основе полученных в диссертационной работе теоретических результатов реализованы прототипы на Прологе с мультипарадигмальным расширением, позволяющие проводить верификацию на основе синтаксического анализа. Для анализа и сравнения существующих подходов реализованы ПО «*shrinker*» для локализации ошибок и ПО «*builder*» вместо ПО «*make*».

Результаты диссертационной работы использованы при разработке учебно-методических материалов по  $\lambda$ -вычислениям и введению в систему верификации Соq «*Верификация систем программного обеспечения*» на кафедре МО ЭВМ СПбГЭТУ.

## Положения выносимые на защиту

1. Диалект языка Пролог, как инструмент для спецификации и верификации динамической памяти.
2. Метод верификации абстрактных предикатов куч на основе распознавания атрибутивной транслирующей грамматики.
3. Подход к устранению многозначности описания куч для упрощения их анализа и верификации.
4. Комплекс программных средств для верификации, динамической памяти (Builder, Shrinker, ProLogika).

## Апробация работы

Основные результаты работы докладывались научных конференциях докладывались и обсуждались на следующих конференциях:

1. International Conference on Advanced Engineering Computing and Applications in Sciences (ADV-COMP), (Venice, Italy, 2016)
2. 18th Conference of Open Innovations (FRUCT), (Saint-Petersburg, Russia, 2016)
3.  $EMC^2$  «Технологии Майкрософт в Теории и на Практике Программирования: Новые подходы к разработке программного обеспечения», (Saint-Petersburg, Russia, 2014)
4. Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog (SYRCoSE), (Saint-Petersburg, Russia, 2014)
5. Advances in Methods of Information and Communication Technology, (Helsinki, Finland/Petrozavodsk, Russia, 2008)
6. A Stricter Heap Separating Points-To Logic, (Moscow, Russia, 2016)
7. International Conference on Control Processes and Stability, (Saint-Petersburg, Russia, 2008)
8. Санкт-Петербургский Электротехнический Университет «ЛЭТИ» (ППС ЛЭТИ), (Saint-Petersburg, 2015)

## Структура и объем диссертации

Диссертационная работа состоит из введения, пяти глав, заключения, библиографического списка и приложений. Основная часть работы изложена на 267 страницах и содержит 97 рисунков.

Для введения в предметную область диссертации *«Логический язык программирования как инструмент спецификации и верификации динамической памяти»* приводится этот раздел. Первый раздел посвящается вычислению Хора, основным определениям и постановлению решения вычислением, альтернативным подходам и свойствам систем вычислений Хора.

Далее для сравнения различных динамических и статических подходов, можно использовать *«качественную лестницу»* из рисунка 1.1 в любой момент. Эта лестница является моим личным предложением для классификации качества, которое должно соблюдать любое программное обеспечение. Чем больше программа соблюдает критерии качества, тем искреннее можно считать данную программу качественной. То есть, если программа некорректно вычисляет результат, то на самом деле можно считать неважным, насколько данная функция выполняет критерий входного домена, и тем более безразлично, насколько быстро это сделано. Самое главное - это корректное вычисление программы. Если это соблюдается, то только тогда можно считать уровень *«базисно надёжным»*.

1.	Корректность	базисно надёжно
2.	Полнота	полностью надёжно
3.	Оптимальность ресурсов	оптимально надёжно

Рисунок 1.1: Качественная лестница по критериям важности

С растущим спросом в надёжности растёт спрос на универсальность данной функции, т.е. тогда имеет смысл распространить качество на любые входные функции. Если базисные операции работают правильно, то тогда можно требовать, что для всех входных данных функций всё работает правильно — это второе требование, которое мы обозначим «*полностью надёжным*», более жёстким, чем первое. Если первые два критерия соблюдаются, то с вычислительной точки зрения, данная функция корректна и полна. Тогда имеет смысл, далее вводить оптимизации, которые мы обозначим «*оптимально надёжными*» и которые не меняют вычислительные свойства корректности и полноты. Надёжными оптимизациями могут послужить, например, ускорение часто востребованных программных блоков, визуализация эффектов, работа с файлами и т.д. Визуальные эффекты и работа с файлами не будут рассматриваться.

Каждый из установленных критериев можно чётко обозначить и можно предложить некую метрику для измерения выполнимости. Не имеет смысла рассуждать о качестве программы, когда программа вычисляет корректно и быстро, но не полностью, потому, что универсальность применения подрывает качество существенно, когда не имеется определение для входного вектора, даже, если все остальные случаи высчитываются очень быстро. Тогда можно, либо ограничить диапазон видимости входного вектора для полного покрытия функции, либо сузить покрытие ради неполной надёжной функции. Когда будут рассматриваться различные подходы, качественная лестница будет использована как ориентир для принятия или отклонения предложений в дальнейшем.

В данной работе рассматриваются императивные и искусственно-гипотетические языки программирования. Для практической применимости необходимо рассматривать объектно-ориентированные модели, поэтому в следующем разделе вводятся основные понятия объектно-ориентированных вычислений, в частности, *Теория Объектов* (ТО). Далее рассматриваются теоретические и практические проблемы работы с динамической памятью, а затем вводятся модели представления динамической памяти, как, например, *анализ образов* (АО), *вычисление регионов* (ВР), а также другие модели представляющие косвенно-динамическую память. *Логика распределённой памяти* (ЛРП) является основной теоретической моделью представления динамической памяти этой работы. Затем даётся обзор по автоматизации доказательств и обсуждаются её факторы противостояния. С целью лучшего понимания проводимых далее доказательств с помощью модели Хора и улучшения сходимости деревьев доказательств, вводятся представления «*абстракции*». С кратким обзором по тематике можно также ознакомиться в [303]. Затем, с целью ознакомления, рассматриваются обла-

сти применения и связанные с главным направлением работы в этой области, в частности — *анализ псевдонимов* (см. раздел 1.4.1), *сбор мусора* и верификация кода с *интроспекцией*. В конце этой главы представляются существующие программные системы и среды.

## 1.1 Вычисление Хора

*Вычисление Хора* (назван в честь информатика Чарльса Антони Ричард Хора, в литературе также встречается фамилия «Хоаре» вместо «Хор», однако известно, что сам автор пишет свою фамилию как «Хор» на русском, а также это способствует избегать неверные произношения других авторов) - это формальный метод *верификации*, который позволяет проверить верность данной программы, данной некоторой *спецификацией*, т.е. согласно описанию, характеризующему свойство поведения программы. Спецификации описывают *состояние вычисления* с помощью математических формул и теорем, а вывод происходит согласно аксиомам и правилам рассматриваемой области дискуссии.

Цель верификации программы, с помощью вычисления Хора, - это проверка соблюдения свойств программы, что является повышением качества и надёжности программы. Если программа соблюдает свойства утверждениями, то это позволяет нам характеризовать программу и сравнивать её с другими программами, где одно или иное свойство, возможно, не соблюдается.

Любое *правило Хора* необходимо рассматривать, как *логическое суждение*  $A \Rightarrow B$ , которое читается: «если суждение *антецедент*  $A$  выполняется, *тогда* выводимое суждение *консеквент*  $B$  следует». Рассмотрим рисунок 1.2:

$$\begin{array}{ll}
 (P1) \frac{A}{B} & (P2) \frac{B \vee \neg B}{C} \\
 (P3) \frac{\{P\}C\{Q\}}{\{P'\}C'\{Q'\}} & (P4) \frac{\{P \wedge \Pi\}A\{Q\} \quad (P \wedge \neg \Pi) \Rightarrow Q}{\{P\} \text{ if } (\Pi) A \{Q\}}
 \end{array}$$

Рисунок 1.2: Логические правила вычисления Хора

Правило Хора (P1) является *аксиомой*, пусть  $A$  будет пустым антецедентом. Главная идея аксиомной системы заключается в проверке верности следствия согласно применению конечной последовательности определённых правил. Отныне, мы не различаем между правилами и аксиомами, т.к. первое является обобщением второго. Правило (P2) является аксиомой, если *антецедент*  $B \vee \neg B$  выполняется всегда аналогично *логике утверждений* и сопоставляется «*истинно*». Однако, изначально не имеются искусственные ограничения в вычислении Хора, например, касательно суждений. Утверждения могут быть сопоставлены предикатами, а *логический вывод* становится *суждением* над предикатами. Различные методы могут быть применены для вывода с *предикатами первого порядка*, например: *метод естественного вывода* [262], *метод резолюции* или *метод семантических таблиц* («*Tableaux method*»). Перечисленные методы являются *дедуктивными*. В отличие от дедукции, ещё имеются *абдукция* и *индукция*. Индуктивный метод вывода предлагает ввод ещё не существующие или тяжело выводимые, явным образом, утверждения в набор рассматриваемых правил. Индуктивное правило введённое «кажется» искусственно, нельзя отклонить из-за отсутствия противоречивого примера. Классическим примером индукции служит *теория опровергаемости по*

*Попперу*: данное утверждение «все лебеди белые» на практике, из-за ограничений не может быть проверено целиком за конечный промежуток времени. Поэтому, предлагается исходить из верного утверждения до тех пор, пока не будет замечен противоречивый экземпляр, например, не будет найден чёрный лебедь. Индукция нам предлагает ради преодоления необъяснимых и неотрицаемых феноменов в рамках рассматриваемого ракурса ввод нового правила, согласно которому феномен может быть обоснован. Свойства индуктивно определенных перечисляемых, возможно бесконечных структур, проверяются с помощью конечной формулы. Когда наблюдается противоречивый индукцией экземпляр, мы вынуждены скорректировать наши утверждения о мировоззрении. В отличие от этого, абдукция ищет необходимые и допустимые предпосылки, чтобы следствие было выводимо из набора правил.

Вычисление Хора можно охарактеризовать, как дедуктивное суждение, применив к императивному программному оператору (как это было предложено изначально Хором), а каждое выводимое суждение описывается состоянием вычисления, до и после выполнения оператора и программным оператором (см. раздел 1.1.1). Ради улучшения *сходимости доказательства*, часто можно добавлять индукцию и абдукцию в качестве метода вывода. Вывод дедукции интуиционистен [47] потому, что утверждение только тогда верно, когда даны условия и соответствующий факт предусловия. Когда мир расследуемых выводов производится строго согласно правилам и пересекаемые в антецеденте, правила исключены, то *космос выводимых следствий* является *замкнутым*, к примеру, правила (P3) и (P4) из рисунка 1.2, интерпретации возможных логических утверждений также замкнуты.

### 1.1.1 Тройка Хора

Для *формальной верификации* с помощью спецификации, Хор в [116] предлагает для *императивных языков программирования* определить тройку.

**Определение 1.1** (Тройка Хора). *Тройка Хора состоит из предусловия  $P$  до и постусловия  $Q$  после загрузки программного оператора  $C$ , сокращено обозначено как  $\{P\}C\{Q\}$ .*

*Декларативные языки программирования*, например, *функциональные*, отличаются от императивных тем, что состояние вычисления меняется в зависимости от содержимого переменных. Порядок выполнения программных операторов не строгий. Декларативную парадигму программирования можно отделить от императивного с помощью модели организации памяти касательно символов и переменных. Далее в этой работе рассматриваются исключительно императивные языки программирования, в качестве входного языка программирования. Изначально Хор предлагал, в качестве языка программирования простой императивный язык наподобие *диалекта Паскаль* и нотацию  $P\{C\}Q$ . Скобочная нотация над оператором оказалась не популярной, поэтому скобки стали ставить вокруг  $P$  и  $Q$ .  $P$  и  $Q$  оба описывают *состояния вычисления* в качестве утверждений до и

после  $C$ .  $C$  может содержать любое количество императивных операторов. Из сказанного следует, что  $C$  меняет пошагово состояние вычисления, если  $C$  не делится далее, а следовательно, состояние памяти меняется пошагово. Под памятью мы подразумеваем *список процессорных регистров, стек и динамическую память* (см. рисунок 2.1). Тройка Хора интерпретируется так: если имеется состояние вычисления  $P$  и программный оператор  $C$  выполнен, то вычисление совершено и находится в состоянии  $Q$ . Другими словами,  $P$  и  $Q$  описывают состояния памяти до и после  $C$ . Если тройка Хора соблюдается, то переход состояний памяти верный, в противном случае, имеются следующие причины несоблюдения:

1. Если  $C$  не завершает работу, то данные правила Хора назовём «*неполной определённой*» и состояние  $Q$  не достижимо. Такое поведение  $C$  опишем формально как:  $\vdash \{P\}C\{Q\} \rightarrow (\llbracket C \rrbracket \neq \perp) \wedge Q$ , где  $\llbracket \cdot \rrbracket$  денотационное преобразование программного оператора [9],[6],[276].
2. Аксиоматические правила не полны. Выбираемое правило отсутствует для данного состояния  $P$ .
3. Полученное актуальное состояние вычисления  $Q$  не является ожидаемым состоянием  $Q'$ .

На рисунке 1.3 указаны наиболее важные аксиомы и правила для императивных языков программирования. Программу написанную императивной парадигмой можно преобразовать в программу декларативной парадигмы, а также обратно, благодаря симуляции *машиной Тьюринга* о вычислимости. Программы можно более абстрактно представить блок-схемами. Вызовы подпрограмм завершаются «*обычной*» стековой архитектурой. Изначальное вычисление Хора [116] не накладывает дополнительные ограничения на описания утверждений в математических формулах, также как и предложенные подходы из [16] не накладывают дополнительные ограничения. Апт предлагает вычисление Хора для верификации одно- и многопоточных программ, а также утверждения классифицировать на входные и выходные переменные. Позже мы оценим достоинства и недостатки различных подходов верификации программ.

$$\begin{array}{ll}
 \text{(SEQ)} \frac{\{P\}A_1\{Q\} \quad \{Q\}A_2\{R\}}{\{P\}A_1; A_2\{R\}} & \text{(LOOP)} \frac{\{P \wedge B\}S\{P\}}{\{P\}\text{while } B \text{ do } S \{-B \wedge P\}} \\
 \text{(ASN)} \frac{}{\{P[e/x]\}x := e\{P\}} & \text{(CONSEQ)} \frac{P' \Rightarrow P \quad \{P\}C\{Q\} \quad Q \Rightarrow Q'}{\{P'\}C\{Q'\}} \\
 \text{(STRONGPRE)} \frac{R \Rightarrow P \quad \{P\}A\{Q\}}{\{P\}A\{Q\}} & \text{(WEAKPOST)} \frac{\{P\}A\{Q\} \quad Q \Rightarrow R}{\{P\}A\{Q\}}
 \end{array}$$

Рисунок 1.3: Неполный список правил для верификации

Итак, *правило следования* (SEQ) из рисунка 1.3 служит примером и означает: если имеется оператор разделитель «;», то для доказательства корректности  $A_1; A_2$  с предположением  $P$  и постуло-

вием  $R$  необходимо доказать, что существует промежуточное утверждение  $Q$ , если доказать сначала первый оператор  $A_1$  с предусловием  $P$ , а затем  $Q$  служит в качестве предусловия в доказательстве  $A_2$ . Корректность последования считается доказанным, как только  $A_1$  доказано, а  $A_2$  доказано с постусловием  $R$ .

Вторым примером служит *правило логического следствия* (CONSEQ), которое является обобщением правил (STRONGPRE) и (WEAKPOST). Конкретизированные правила, либо ужесточают предусловие, либо обобщают постусловие. Если предикат верный в общем случае, тогда для некоторого множества  $V$  предикат также верен в частном случае для любого  $v \in V$  или для любого подмножества  $V_1 \subseteq V$ .  $V$  является *обобщением* от  $V_1$ , а  $V_1$  является *конкретизацией*  $V$ . Нетрудно убедиться в том, что импликация  $V \Rightarrow V_1$  в силе, однако,  $V_1 \Rightarrow V$  не для каждого предиката действительно. Исходя из общего случая, можно выводить верность предиката без ограничения общности для конкретного случая, поэтому в силе правило (STRONGPRE), а также в силе (WEAKPOST). Если (STRONGPRE) и (WEAKPOST) объединить, учитывая переименование переменных состояний вычисления, то как раз получается (CONSEQ).

Третьим примером служит *правило цикла* (LOOP). Это правило гласит, что если имеется цикл в качестве программного оператора и мы имеем предусловие  $P$  и постусловие  $Q$ , то достаточно доказать корректность блока цикла  $S$ , как единый программный оператор и добавить к  $P$  условие цикла  $B$ . При выходе из блока цикла необходимо добавить в постусловие отрицательное условие цикла.

Отмечается, что благодаря циклу и *правилу присвоения* возможно преобразовать любой другой вид цикла в данный вид цикла (LOOP) как изложено в рисунке 1.4.

$$\begin{aligned} \text{(REPEAT)} \quad & \frac{\{P\}S\{P'\} \quad \{P'\}\text{while } B \text{ do } S \{Q \wedge \neg B\}}{\{P\}\text{do } S \text{ while } B\{Q \wedge \neg B\}} \\ \text{(FOR)} \quad & \frac{\{P\}i := 1; \text{while } B \text{ do } (S; i := i + 1) \{Q\}}{\{P\}\text{for } i \text{ from } 1 \text{ to } n \text{ do } S \{Q\}} \end{aligned}$$

Рисунок 1.4: Правила циклов заменившие while

Ради простоты, договоримся, что в *правиле* (FOR) *ранее известных количеств итераций*  $n$  переменная  $i$  является свежей, т.е. неиспользуемой в предикате  $P$  или  $Q$ . Если переменная не свежая, то согласно *диапазону годности* переменных, вводится свежая переменная. С проблемами *индексации термовых выражений* и подхода избежания коллизии наименований можно ознакомиться в [204]. То есть, правило (REPEAT) можно преобразовать в (LOOP) и обратно, однако, в общем (FOR) можно только в (LOOP) преобразовать, т.к. для обратного шага всегда необходимо заранее знать количество итераций, что не всегда известно. С вопросами преобразования *примитивной рекурсии*, *μ-рекурсии* и общими видами взаимной формы рекурсии, а также с вопросами свойств терминации



при *взаимной рекурсии* можно ознакомиться в [25].

Правило для оператора условного перехода здесь отдельно не вводится потому, что он без ограничения общности может быть сопоставлен правилом цикла (LOOP). Для полного понимания преобразуемости, рекомендуется ознакомиться с минимальным языком с точки зрения вычислимости «PCF» [209],[65]. Основной идеей вычислимости является симуляция Тьюринг-вычислимых функций с помощью минимального набора программных операторов. Особенность циклов, в отличие от других *линейных программных операторов*, заключается в том, что один и тот же блок повторяется множество раз, при этом, состояние переменных обычно меняется. Чтобы зафиксировать совокупность всех изменений блока цикла, необходимо проанализировать *зависимость данных* и все изменения переменных до выхода из цикла. Повторение цикла необходимо абстрагировать, обычно вручную, подставляя новые искусственные переменные, для определения наиболее обобщённого уравнения целевых переменных (ср. рисунок 1.11), которые формируют *инвариант цикла*. Нахождение инварианта может часто оказаться трудным, по крайней мере не тривиально и чисто автоматически не может быть выявлен, поэтому требует построение спецификации вручную. Постусловие блока цикла представляет собой формулу, которая содержит инвариант. Аналогично к фиксированному отображению в проективной геометрии, когда одна точка при трансформации остается фиксированной, тогда инвариант блока цикла это утверждение, которое остается верным при любом множестве раз итераций. Итак, данная инвариантная формула  $\Phi$  должна соблюдать равенство  $\Phi \circ Y = Y \circ \Phi \circ Y$ , где  $Y$  является некоторым *комбинатором плавающей точки*, а  $\circ$  является бинарным оператором применения функций.  $Y$  является некоторым синтаксическим методом симуляции повтора, либо его цель может определиться как «поисковиком минимума», который широко применяется в  $\lambda$ -вычислениях [22]. Можно использовать альтернативную нотацию вычислимости  $\mu$ -оператора Клини, указав в качестве минимизирующих параметров меняющиеся переменные.

Рассмотрим пример из рисунка 1.5.

```
a:=0; b:=x;
while b>=y do  b:=b-y; a:=a+1; do
```

Рисунок 1.5: Пример кода остатка при делении целых чисел

Инвариантом здесь может быть  $a \cdot y + b = x^b \geq 0$ , т.к. равенство остатка при делении на  $y$  не меняется циклом.  $y$  является делителем целого числа  $x \geq 0$ ,  $a$  целое частное число, а  $b$  это остаток при делении  $x$  на  $y$ . Правило оператора присвоения (ASN) означает: если переменной  $x$  присваивается годное значение  $e$ , то, до и после присвоения состояния  $P$  остается без изменений. Состояние до присвоения среды присвоенных символов необходимо расширять, включив  $x$ . В случае возникновения коллизии с именем, необходимо сначала в спецификации провести переименование конфликтующих переменных.

*Полнота правил* зависит от *полноты троек Хора* в виде  $\{P\}C\{Q\}$ , которая зависит от покрытия всех программных операторов  $C$  вместе со всеми допускаемыми предусловиями  $P$  (см. опр.1.7). Постусловия  $Q$  являются лишь логическими последствиями троек, которые выводимы из  $P$  и  $C$ . Так как правила для любой годной программы могут применяться потенциально в любом порядке, возникает вопрос, а может ли одно правило нечаянно или преднамеренно исключить любое другое данное правило? Найти ответ наивным подходом может оказаться сложным делом. Очевидно, что если имеется постусловие  $Q$  и данная программа  $C$  выводит различные  $P_1$  и  $P_2$ , то это явно показывает на *некорректность* правил Хора. Кроме полноты и корректности, согласно лестнице качества из рисунка 1.1, оптимальность ресурсов также важна. Вопрос, насколько эффективны или «удобны» правила Хора, затрагивает также вопрос о компактном, но понятном для пользователя представлении. Если идет вопрос об автоматизации, то это также затрагивает вопрос о вычислительной технике. В этом разделе мы увидели, что задача постановления инварианта может оказаться сложной попыткой, т.к. для разумного и обобщённого вывода, необходимо включить все переменные блока цикла, включая все переменные памяти. Обратим внимание на то, что автоматически выделенные переменные имеют диапазон видимости, а у динамически выделенных переменных диапазон отличается. Кроме того, необходимо заметить, что сравнение равенства между имеющейся и ожидаемой спецификацией может потребовать некоторый консенсус по представлению состояний вычисления.

### 1.1.2 Логический вывод

Правило ( $P1$ ) из рисунка 1.2 представляет собой самую обобщённую форму логического правила. Верификация, это проверка данной программы  $C$  с условием, что при начальном предусловии  $P$ , следует постусловие  $Q$ . Верификация, это формализованный процесс (см. рисунок 4.8), который начинает проверку последовательности программных неделимых операторов с предусловием  $P$  и с постусловием  $Q$ . *Результат верификации* либо верный, либо неопределённый, когда постусловие отсутствует или оператор не терминирует, либо отрицательный. Применяя правила, может возникать необходимость разветвления доказательства на под-доказательства, в итоге, *структура доказательства является деревом*.

**Определение 1.2** (Входной язык программирования). *Язык программирования является формальным языком, чьи слова соответствуют программам, которые являются последовательностью программных операторов. После запуска каждого из программных операторов, меняется состояние памяти. В качестве входного доказуемого языка программирования, рассматривается по умолчанию императивный язык, близкий к подмножеству Си с объектным расширением.*

Императивный язык программирования выбирается по целому ряду причин. Во-первых, императивные диалекты, как Си или Ява, довольно популярны, и необходимость введения всё новых

формализмов к большой части отпадает. Во-вторых, в этой работе выбирается прототипно Си диалект, который имеет возможность удобно и просто обсуждать синтаксис и семантику операций над динамической памятью. Естественно, Си имеет моменты, которые зависят от одной или иной платформы, однако это обсуждается и важно понять, например, для дальнейшего применения предложенных подходов.

**Определение 1.3** (Язык спецификации). *Язык спецификации является формальным языком, который ссылается на переменные и единицы данной входной программы, символьные выражения, кванторы и вспомогательные единицы для проведения доказательств. Язык спецификации подлжит некоторой согласованной формальной логике. Спецификация, в отличие от входного языка программирования, несет декларативный характер, а не императивный.*

$$\frac{\frac{A_1 \quad \frac{\text{false}}{A_2}}{A_3} \quad \frac{\frac{\text{true}}{B_1}}{B_2}}{B}$$

Рисунок 1.6: Пример отрицательного логического вывода

Язык спецификации в каждом блоке графа потока управлений (см. рисунок 1.11) описывает состояние вычисления, опираясь на состояние памяти (см. рисунок 2.1). Рассмотрим свободно выбранное *дерево вывода* из рисунка 1.6 со следующими утверждениями  $\{A_1, A_2, A_3, B_1, B_2, B\}$ . Изначально требуется доказать тройку  $B$ , согласно опр.1.1. Для этого применяется данное правило, в антецеденте которого должно иметься  $A_3$  и  $B_2$ , оба из которых следует отдельно доказать с соответствующими сопоставлениями так, чтобы имелось соответствие строго по правилу (например, преобразование локальных символов) с *консеквентом*  $B$ . Далее, применив некоторые имеющиеся правила, мы показываем, что  $B_1$  является предусловием для  $B_2$ , а  $B_1$ , согласно правилу, всегда верное утверждение, например,  $\{n = 0 \wedge n \geq 0\}a = 5; \{n = 0\}$  если очевидно, что  $a$  не связанная, т.е. *свободная* переменная с  $n$ . Далее доказывается  $A_3$ , в результате чего, получается, что  $A_2$  противоречит самому себе. Это условие достаточное, чтобы  $A_3$  вычислялось как «ложь», а следовательно и  $B$ . То есть, мы только что доказали, что тройка утверждения Хора  $B$  неверна и причина тому  $A_2$ . Поэтому в данном примере нет необходимости дальше доказывать  $A_1$ , независимо, верно оно или нет.

**Определение 1.4** (Логическое следствие). *Логическое следствие  $A \vdash B$  обозначается как утверждение  $A$ , к которому применяется некоторое данное правило один раз. Оно приводит к утверждению  $B$  (по Фреге [263]). Если  $B$  получаем после применения правил несколько раз подряд, включая ни разу, то следствие получается  $A \vdash^* B$ . Если мы хотим выразить, что некоторая тройка  $A$ , согласно данному набору правил Хора  $\Gamma$  всегда истина, то мы это обозначим как  $\models A$ , либо как  $\models_{\Gamma} A$  если ударение поставить на выбранный набор правил из набора  $\Gamma$ .*

**Определение 1.5** (Доказательство как поиск). *Доказательство в вычислении Хора, при данном наборе правил  $\Gamma$  и данного следствия  $B$ , является поиском аксиом, т.е.  $\models_{\Gamma} B$ .*

В данном определении мы сознательно допускаем неточность в связи с набором правил и вычислением Хора. Например, ссылаясь на  $\Gamma$ , мы ссылаемся на *формальную логику*, которая состоит из замкнутого по зависимости подмножества данных правил Хора, как *множество носителя* и базисных логических констант. По умолчанию мы согласуем, что для любого данного набора правил, для логического вывода, мы подразумеваем присутствие корректно определённого вычисления Хора, согласно тройке Хора из опр.1.2, опираясь на опр.1.4.

**Определение 1.6** (Корректность вычисления Хора). *Вычисление Хора является корректным, если исключен случай, когда синтаксически подлинная программа  $C$  и данный набор правил  $\Gamma$  выводят различные противоречивые результаты.*

Если один логический вывод приводит к одному результату  $B_1$ , а второй также допущен согласно  $\Gamma$  и вывод приводит к другому результату  $B_2$ , который не выводим из  $B_1$  или наоборот. т.е.  $\{P\}C\{Q\} \vdash^* \{P_1\}C_1\{Q_1\}$  и  $\{P\}C\{Q\} \vdash^* \{P_1\}C_2\{Q_2\}$  но, при этом,  $\{P_1\}C_1\{Q_1\} \not\vdash^* \{P_2\}C_2\{Q_2\}$  и  $\{P_2\}C_2\{Q_2\} \not\vdash^* \{P_1\}C_1\{Q_1\}$  (см. рисунок 1.7), то правила являются *не корректными*. Это означает, что свойство корректности данного вычисления Хора соблюдает *свойство диаманта/ромба*, т.е. *теорема Чёрча-Россера* применяется к тройкам Хора (см. [203]). Если хотя бы одно утверждение при выводе противоречит результату другого вывода, то правила Хора являются *не корректными* и *не полными* (см. опр.1.7). *Сходимость* — более жесткое требование, чем корректность и не всегда соблюдается например, из-за *незавершения вычисления* (более подробнее можно ознакомиться в [247]). Однако, можно заметить, что корректность обязательное условие для сходимости. То есть, если вычисление не корректно, то имеется хотя бы один случай, когда выводятся два или более различных результата и это обязательно не корректно. Согласно Штайнбаху [247] и его рассматриваемой *системе переписки термов* (с англ. «*term rewriting system*») [19], *терминация программ* сильно определяет сходимость. Он представляет правила вывода в качестве правил переписки термов. С помощью *ограниченности упорядоченных цепочек*, он может в частных случаях доказать терминацию системы переписки. Аппроксимация верхнего порога выводов системы в общем из-за теоретической нерешимости не распространяется на самосодержащие термы или на неограниченные символы [207]. Идея *нисходящей цепочки* тесно переплетается с *теорией доменов* [236]. Штайнбах использует цепочки для решения *лимита* выводов, т.е. для решения вопроса терминации, которая является условием корректности.

Кук [67] рассматривает корректность и полноту различных вычислений Хора. Отмечается, что оба свойства существенно могут меняться, например, применив лишь несколько модификаций к переменным в процедурах. Общее понятие полноты по Куку определяется, как *тотальная функция* покрывая все входные программы, учитывая ранее упомянутые свойства в [15]. Корректность опре-

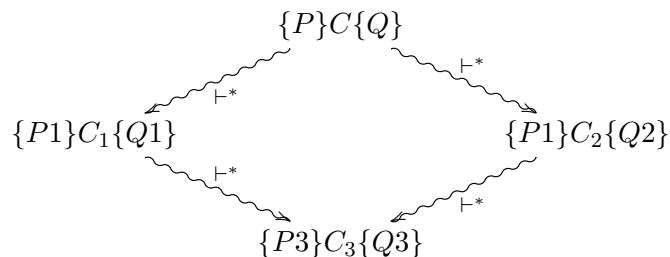


Рисунок 1.7: Теорема Чёрча-Россера применена к тройкам Хора

деляется, как эквивалентность между *наблюдаемым* и *должным поведением*, как это предлагается в [79],[189] с помощью *операционной семантики* [208] над тройками Хора. Вычисления интерпретируются с помощью *абстрактного автомата*. Для достижения «удобного» — здесь подразумеваются, либо полные, либо корректные вычисления, вводятся дополнительные ограничения в императивном языке программирования, такие как:

- **Огран. №1** разрешается использовать глобальные переменные в процедурах, однако, запрещается их передавать в качестве актуальных параметров.
- **Огран. №2** запрещаются параметры по вызову (или по ссылкам).
- **Огран. №3** рекурсивные процедуры и *функционалы* (*функции высшего порядка*)

Почему такие ограничения, как только что были показаны, могут привести к некорректности или к неполноте? Классический стек при передаче *параметров по ссылкам* нуждается в обратной связи сквозь стек-окон вызовов. Как только, прекращается вызов, то так прекращают существовать и значения. Передача по ссылке разрешает манипуляции единого содержимого в других *стековых окнах*, но адрес *указателя* на объект может меняться в каждом стековом окне. Если допускать *параметры по вызову* или рекурсивные функции, то это может привести к изменениям вне соответствия *вызову стека*, как например, *глобальные переменные*, они практически могут меняться везде. В таких случаях «*наивные*» спецификации могут быть просто неправильными в общих случаях. Этому можно противостоять, ограничив модус входных и выходных параметров процедур. Иерархические спецификации [235],[37] могут сильно раздуть спецификации, и из-за широкой периферии возможных мест в программе, где может поменаться один указатель, польза при этом, явно ограничивается. В итоге, они также могут оказаться мало эффективными, т.к. все встроенные процедуры специфицируются — при этом, исходя из наиболее общего сценария. В частности, объектные экземпляры могут потерять свойства идентичности и целостности, т.к. по обобщённой схеме передача процедур в качестве параметра, может привести к не непредвиденным действиям, а следовательно, может кардинально поменять состояние вычисления.

Кларк [62] выделяет исключительно актуальные ограничения вычисления Хора, которые остаются до сегодняшнего дня. Острые ограничения касаются:

- **Огран. №4** выразимость и неполнота *языка утверждений*
- **Огран. №5** ограничения в связи с инвариантами циклов (преобразование, и т.д.)
- **Огран. №6** рекурсивные процедуры (не) использующие глобальные и *статические переменные*
- **Огран. №7** со-процедуры в качестве входного и выходного параметра
- **Огран. №8** динамическое выделение и освобождение ячеек памяти (см. [215])

Также как и Кук, Кларк подразумевает под корректностью гарантии о том, что все синтаксически корректные теоремы выводятся верно, а все не корректные теоремы, как ложные. Как любая формальная система, вычисления Хора тоже подлежат теоретическим ограничениям. Например, если рассмотреть *теорию целых чисел*, то можно всегда придумать всё новые теоремы над целыми числами, которые явно корректны, но которые нельзя доказать данным замкнутым вычислением. Этот феномен лучше известен, как *теорема Гёделя о неполноте*. Кларк также замечает, как это ранее до него заметил Кук, что если вычисление Хора содержит не завершающийся цикл, то причиной служит рекурсия в правилах, которая применяется сама к себе, поэтому верификация может не завершаться. Он предлагает запретить *псевдонимы ссылок* (см. далее), а ради корректности исключить неограниченную рекурсию. Проблему Кларка можно частично разрядить, если правила сопоставления применяются сначала к крайне внешнему терму, вместо к крайне внутреннему терму. Это необходимо учесть при реализации правил Хора в программных средах, где сначала вычисляются параметры процедур, а только затем, передаётся содержимое параметров. Языки программирования без *ленивого вычисления*, как например, «*OCaML*», в отличие от *ленивого вычисления*, как например, «*Хаскель*», должны, если этого требует ленивый алгоритм, добавить ленивое вычисление искусственным добавлением дополнительных проверок на местах использования параметров. По Кларку выразимость всегда касается языка утверждений, который может быть представлен в виде термов. Ограничение №6 касательно рекурсивных процедур можно исключить, если: (i) рекурсия завершается, т.е. некоторая нисходящая цепочка вывода всегда существует (см. [247]) и/или (ii) последовательность параметров и типизация параметров [55] при вызове процедур должна точно совпадать и исключать переменные, выделенные в актуальном стековом окне, т.е. не глобальные, не статические переменные, и т.д.

Кофмэн [134] выделяет проблемы выразимости и возможность применимости на практике, как наиболее важные, которые сильно противодействуют популяризации *формальному методу* верификации — языки спецификации и языки программирования. Наиболее важными практическими

проблемами выразимости он считает индукцию и абстракцию. Для того, чтобы простым образом различать ошибки некорректного вычисления от недостоверной спецификации, он также замечает необходимость, простого но обобщённого подхода для генерации *контр-примеров* — пример, который доказывает неверность данной формулы с помощью конкретных входных символов.

Герхарт [104] анализирует существующие к тому времени подходы для решения проблем полноты, которые к сегодняшнему дню решены не в полном объёме и решены не удовлетворительно: (i) наиболее обобщённый подход в решении терминации программ, (ii) решение открытых вопросов верификации переменных в различных областях памяти, как например, статические переменные (см. [62]), (iii) вопросы удобства наиболее обобщённого представления и использования систем верификации. Она отмечает, что *индуктивные определения* являются одним ключевым методом в решении проблем выразимости, и поэтому их рассматривает как многообещающий технический аппарат. Герхарт выдвигает требование: простые доказательства должны ссылаться на универсальные и глобальные требования. Из [62] также следует, что проблемы из (ii) можно считать трудными и глобальными, что вряд-ли минимальная модификация вычисления Хора позволит их решить.

**Определение 1.7** (Полнота вычисления Хора). *Вычисление Хора является полным, когда для любой синтаксически корректной программы, верность согласно правилам, может быть доказана и верность некорректных программ может быть отвержена.*

В случае нехватки хотя бы одного правила до завершения верификации, вычисление считается *неполным*, а сама верификация *неопределённой*. На практике, уже маленькая модификация программных операторов или свойств единиц спецификации, может привести к существенному изменению системы вычисления [62, 67, 68]. Это свидетельствует о большой сложности системы верификации. Пример 1: согласно ограничению № 6 отсутствие статических переменных или рекурсивных процедур может всё равно привести к полной программе, в зависимости от того, какая реальная часть программы рассматривается, и какие комбинации допускаются. Пример 2: полнота верификации программ с внутренними процедурами в общем недействительна, когда хотя бы одно из ограничений 6 или 7 не соблюдается. Если два правила приводят к различным результатам, т.е. из данного состояния  $A$  выводится  $A \vdash B_1$  и  $A \vdash B_2$ , при этом  $B_1$  и  $B_2$  синтаксически различны, но свойство алмаза соблюдается, то  $B_1$  и  $B_2$  являются лишь промежуточными состояниями и оба состояния сходимы. Проблема проверки сходимости методом конечного отслеживания может оказаться неэффективным, т.к. во время верификации необходимо проводить экспоненциальное количество под-доказательств. Эту проблему можно избежать, если детерминировать все правила. Также Кларк приводит следующее ограничение для устранения неполноты:

- **Огран. №9** частично-вычисляемые структуры данных.

У частично-вычисляемых структур данных все поля вычисляются в момент доступа. Типичный пример взят из [255] на языке *Хаскель* о потенциально бесконечных линейных списках.

```
take 10 [ (i,j) | i <- [1..], let k = i*i, j <- [1..k] ]
```

вычисляет  $[(1,1), (2,1), (2,2), (2,3), (2,4), (3,1), (3,2), (3,3), (3,4), (3,5)]$ ,

однако, определение линейного списка как второй аргумент от функции **take** не имеет лимита. Данная структура определяет множество, которое имеет только нижний порог (это целое число 1), но не имеет верхнего порога. С помощью частично-вычисляемых структур, можно проверить, насколько *строго вычисляет* данная процедура. «*Строго*» подразумевает, что данная процедура сначала вычисляет все входные параметры, а затем их записывает в память, а *не строгие* процедуры означают, что входные параметры вычисляются частично и только тогда, когда они требуются на данном этапе алгоритма (см. [256], [255]). Таким образом, из этого можно сделать вывод, например, проверить терминацию программы можно, используя бесконечно определённую структуру данных в качестве быстрого теста.

Уэнд [265] подразумевает под *полнотой функции* определения Кука, уточняя, что каждый верный параметр на входе соответствует верному параметру на выходе, а каждый не верный параметр соответствует ошибке. То есть, переходная функция должна быть тотальной, и все не терминирующие функции не определены по умолчанию (см. опр.1.6). Уэнд показывает, что спецификация программы, представленная графом потока управления тогда не определена и не полная в общем случае, когда для описания графа используется *логика высшего порядка*. По Уэнду квантифицируемые предикаты должны давать «*понятное определение*» самым лучшим образом так, чтобы человек прочитав лишь спецификацию, мог бы сразу и интуитивно охватить полностью замысел предиката.

Кук [68] сравнивает решение проблемы *выполнимости булевых формул* с теоретическими оценками решения проблем верификации. Статья представляет теоретические пороги сложности. С практической точки зрения, статья не пригодна по двум причинам. Первая причина - пороги слишком грубые и поэтому для применения недостаточны. Вторая причина - основной характер статьи — это философский дискурс познавательного характера.

Лэндин [152] предлагает элементы формального вычисления преобразовать в выражения  $\lambda$ -термов, как это было изначально предложено Чёрчом. Под формальным вычислением можно также подразумевать вычисление Хора. Лэндин на примерах условных переходов и рекурсии показывает, что термы полностью покрываются. Более того, он показывает, что программы на основе функциональной парадигмы [255],[35] могут быть представлены также в программе на императивном языке программирования с помощью *замыканий* (с англ. «*closure*») и на основе операционной семантики. То есть, представляются обобщённые *модели вычислимости*, о которых важно знать при моделировании систем верификации.

Исходя из опыта последних декад, Апт [15] определяет некоторый обобщённый диалект Си с редуцированным множеством программных операторов для анализа полноты и корректности. В дополнении ранее обсуждаемых работах, Апт видит общую рекурсию, как наиболее важную проблему,



которую трудно прогнозировать в отношении следующего состояния программы. Поэтому, он предлагает ограничиться примитивной рекурсией. Также Апт предлагает допускать только те вызовы процедур, у которых актуальные параметры совпадают с *формальными параметрами*, например, «*инкорректные параметры*» исключать, т.к. они могут послужить целому ряду аномалий. Например, неверное разделение, неинициализированные поля, а также полученный функционал в связи с отсечением параметров при вызове, могут полностью поменять семантику процедуры, но принципиально не решить ни одну проблему. С помощью урезания параметров можно сильно варьировать семантику и заметно изменять синтаксис.

Кук [67] и другие упомянутые авторы рассматривают две основные проблемы полноты:

- **Полнота №1** Незавершение вычисления процедуры.
- **Полнота №2** Ограничение выразимости языка утверждений (например, предикаты и инварианты).

В виде примера корректных и полных правил Хора, Кук выделяет рисунок 1.8 (Кук использует обратную запись скобок, как и Хор). На рисунке  $A_j$  представляют программные операторы,  $D$  является блоком декларации переменных,  $\sigma$  является переменной средой, а  $\star$  обозначает *звезду Клини*.  $\sigma$  имеет тип:

имя переменной  $\rightarrow$  значение

В дополнении к ограничению Хора 6, надо отметить принципиальное ограничение:

- **Огран. №10** не автоматически выделенные переменные

Ранее уже упоминалось, что подключение глобальных, статических и динамически выделенных переменных, может аннулировать правила Хора. Далее, локальные переменные используемые в некоторых нитях одновременно, могут также аннулировать правила, т.к. аспекты параллельного вычисления в общем не могут быть покрыты, например, правилами Кука. Тема этой работы посвящается однопроцессорному вычислению, а не параллельному запуску программы.

Для полной индустриальной применимости, необходимо включить обработку *исключений*, которые просто так не вписываются в существующие правила и довольно трудно формально корректно и полностью охарактеризовать не только, но и в целом из-за возможного изменения стека при каждом программном операторе по-разному (см. [84], [107]). До сегодняшнего дня не было найдено предложение о вычислении Хора, которое позволило бы (корректно и/или полностью) *отматывать динамическую память*, аналогично *стеку* [107].

Хор [116] считает абстракцию спецификации самой главной преградой для верификации мало тренированным инженерам, которые желали бы быстро научиться верифицировать простые примеры. По Хору тяжело формализовать утверждения для программных меток, безусловных переходов

$$\begin{array}{ll}
(\text{VAR}) \frac{P \ y/x\{\text{begin } D^*; A^* \text{ end}\} Q \ y/x}{P \ \{\text{begin new } x; D^*; A^* \text{ end}\} Q} & (\text{SEQ}) \frac{P > R, R\{A\}S, S > Q}{P\{A\}Q} \\
(\text{ITE-1}) \frac{P\{A\}Q, Q\{\text{begin } A^* \text{ end}\}R}{P\{\text{begin } A; A^* \text{ end}\}R} & (\text{ITE-2}) \frac{}{P\{\text{begin end}\}P} \\
(\text{CON}) \frac{P \ \& R\{A_1\}Q, P \ \& \neg R\{A_2\}Q}{P\{\text{if } R \text{ then } A_1 \text{ else } A_2\} Q} & (\text{ASN}) \frac{}{P \ e/x\{x := e\} P} \\
(\text{CALL-2}) \frac{p(x : v) \text{ proc } K, P\{K\}Q}{P\{\text{call } p(x : v)\}Q} & (\text{LOOP}) \frac{P \ \& Q \ \{A\} P}{P\{\text{while } Q \text{ do } A\} P \ \& \neg Q} \\
(\text{PAR}) \frac{P\{\text{call } p(x : v')\}Q}{P \ u, e/x', v' \ \{\text{call } p(u : e)\} Q \ u, e/x', v'} & \\
(\text{CALL}) \frac{P\{\text{call } p(u : e)\}Q \quad \text{where } \sigma = z'/z}{P\sigma \ \{\text{call } p(u : e)\} Q\sigma} & \\
(\text{PROC}) \frac{D, P\{\text{begin } D^*; A^* \text{ end}\}Q}{P\{\text{begin } D; D^*; A^* \text{ end}\}Q} & 
\end{array}$$

Рисунок 1.8: Пример полного и корректного набора правил из [67]

и передач параметров по имени. Он не настаивает и не опровергает использование любой логики, любого порядка, однако, он считает декларативную спецификацию утверждения решительным фактором успеха.

### 1.1.3 Автоматизация логического вывода

Цель этого раздела заключается в введении и демонстрации практических и теоретических проблем автоматизированной верификации. В этом разделе мы рассмотрим примеры в системе верификации «*Coq*». В *Coq* [32] можно доказывать заранее специфицированные утверждения с помощью теорем, различных индуктивно определённых структур и различных команд на основе типизированного  $\lambda$ -выражений второго порядка. Утверждения задаются на функциональном языке «*Gallina*», а последовательность доказательств задается языком последовательных команд «*Vernacular*». «*Coq*» является ассистентом доказательств потому, что часто он сам не в состоянии полностью и самостоятельно, даже для простых примеров, находить доказательства. Вместо этого, в «*Coq*» имеется множество узкого круга поддерживаемых теорий, которые можно подключать в работающее ядро ассистента. Ассистент позволяет записывать и отслеживать доказательство и выявить промежуточные состояния доказательства.

В последовательность команд включается набор так называемых «*тактических команд*». Они

пробуют текущее представление программного состояния упростить полу-автоматически, используя рассматриваемую теорию. Теория вещественных чисел, например, используется для ускорения сходимости, результатом чего является, либо завершение доказательства, либо упрощение состояния вычисления. «*Coq*» — довольно мощная и широко используемая платформа для верификаций, что в области автоматизации доказательств теорем, можно встретить не так часто. Применение также включает в себя верификацию корректности фреймворков компиляции [225],[157],[41],[42],[158],[159],[215].

Формулы из логики предикатов задаются языком «*Vernacular*».

**Определение 1.8** (Логическая формула предикатов первого порядка). *Логическая формула предикатов первого порядка  $\Phi$  определяется так:*

$$\Phi ::= \text{true} \mid \text{false} \mid x \mid \text{REL}(f(\vec{x})) \mid P(\vec{x}) \mid \neg\Phi \mid \Phi \circ \Phi \mid \forall x.\Phi[x] \mid \exists x.\Phi[x]$$

где,  $x$  булева переменная,  $f$  функтор,  $P$  предикат утверждения,  $\text{REL}$  предикат связи между аргументами (реляция), а « $\circ$ » логическая конъюнкция, либо оператор, либо дизъюнкция. Вектор  $\vec{x}$  определяет вектор термов, который по умолчанию содержит компоненты в соответствии использования. Формулы использующие кванторы предполагают, что логическая переменная  $x$  имеет свободной в  $\Phi$ .

«*Coq*» использует при редукции нормализованные формулы, которые могут быть не определены или не полностью определены. «*Coq*»-схемы основаны на *ленивой редукции* на вычислительной модели  $\lambda$ -вычисления второго порядка (см. [55], [203], [179]). Разумеется, что из-за произвольного вида правил, естественно не может быть гарантии касательно полноты. Типизация  $\lambda$ -вычислений позволяет избегать целый ряд *парадоксов типизации*, в связи с *рекурсивными определениями Канторовских множеств* (см. [22],[34]), как термы содержавшие сами себя. Отсутствие типизации может приводить, к *парадоксу Расселя о брадобрея*.

**Определение 1.9** (Термы  $T_{\lambda_2}$ ). *Набор типов  $T_{\lambda_2}$  в типизированном  $\lambda$ -вычислении второго порядка определяется так:*

$$\begin{aligned} t &\in T_{\lambda_2}, & t &\in V \\ (t_1 \rightarrow t_2) &\in T_{\lambda_2}, & t_1, t_2 &\in T_{\lambda_2} \\ \forall a.t &\in T_{\lambda_2}, & a &\in V, t \in T_{\lambda_2} \end{aligned}$$

Примерами корректно определённых  $T_{\lambda_2}$ -типов, являются например,  $\forall a.a$  или  $(\forall a_1.a_1 \rightarrow a_1) \rightarrow (\forall a_2.a_2 \rightarrow a_2)$ .

**Определение 1.10** (Множество термов  $\Lambda_{T_{\lambda_2}}$ ).  *$T_{\lambda_2}$ -термы  $\Lambda_{T_{\lambda_2}}$  определяются как:*

$$\Lambda_{T_{\lambda_2}} ::= V \mid \Lambda_{T_{\lambda_2}}\Lambda_{T_{\lambda_2}} \mid \lambda x : t \in T_{\lambda_2}.\Lambda_{T_{\lambda_2}} \mid \Lambda x.\Lambda_{T_{\lambda_2}} \mid \Lambda_{T_{\lambda_2}}t \in T_{\lambda_2}$$

Корректно определёнными  $\Lambda_{T_{\lambda_2}}$ -типами являются, например,  $\Lambda a.\lambda x : a.x$  или

$$\lambda x : (\forall a.a \rightarrow a).x(\forall a.a \rightarrow a)x.$$

Редукция ( $\beta$ -редукция, см. [22])  $\lambda$ -термов проводится как применение возможно неопределённого терма к данной  $\lambda$ -абстракции, например:

$$(\Lambda a.\lambda x : a.x) \text{ Int } 3$$

можно редуцировать к  $(\lambda x : \text{Int}.x) 3$ , далее редуцируется к «3», при этом  $\text{Int}$  тип целых чисел, т.е. множество  $V$ .

Задача редукции  $T_{\lambda_2}$ -термов заключается в: (1) вычислении результата и (2) проверке типов вычисления. Проверка типа отличается от задачи верификации, отсутствием состояния.

**Определение 1.11** (Проверка типа). Проверка данного терма  $e$  имеющий тип  $t$  для данного набора правил и аксиом  $\Gamma$  задается, как  $\Gamma \vdash e : t$ . Тип терма проверяется следующими структурными правилами типизации и далее правилами редукции рассматриваемой теории (здесь пропущены):

$$\begin{array}{c} (\forall\text{-Intro}) \frac{\Gamma \vdash e : t}{\Gamma \vdash \Lambda a.e : \forall a.t} \quad (\lambda\text{-Intro}) \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x : t_1.e : t_1 \rightarrow t_2} \\[10pt] (\forall\text{-Elem}) \frac{\Gamma \vdash e : \forall a.t}{\Gamma \vdash e t' : t[a := t']} \quad (\lambda\text{-Elem}) \frac{\Gamma \vdash e_1 : t_1 \rightarrow t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \end{array}$$

Ради простоты, в определении базовые правила были упущены, т.к. универсальность разрешает их определить более обобщёнными не типизированными  $\lambda$ -вычислениями [22], например, аксиома  $\overline{\Gamma \vdash x : t}$  или правила ( $\exists$ -Intro) и ( $\exists$ -Elim), которые определяются аналогично ( $\forall$ -Intro) и ( $\forall$ -Elem).

Так как  $\text{Coq}$  основан на  $T_{\lambda_2}$  и редукция *редексов* производится снаружи во внутрь, то проблема проверки типов решаема. Сложность проверки линейная. Для стратегии редукции с внутренней стороны к внешней, в общем случае, проверка не решаема (см. [203] и [32]).

Рассмотрим относительно простой пример «исключённого третьего» на рисунке 1.9. Пример тавтологии  $p \vee \neg p$  можно считать интуитивно понятным, но при отсутствии логических таблиц и при использовании только правил импликации — так оно положено в *интуиционистском суждении логических утверждений*, задача может оказаться гораздо труднее. В зависимости от набора правил, *тавтология Пирса* может быть в принципе не выводима.

Рассмотрим пример из рисунка 1.9. Нам необходимо доказать, что первое определение `peirce`, которое обходится без дизъюнкции и отрицания, но содержит импликацию « $\rightarrow$ » может быть преобразовано в определение `lem`. Обратим внимание, что оба определения содержат квантифицируемые утверждения. Доказательство является последовательностью команд, начиная после ключевой команды `Proof` и заканчивая перед ключевым словом `Qed` (перевод с латинского «*quod erat*

```
Definition peirce := forall (p q: Prop), ((p->q)->p)->p.
```

```
Definition lem := forall p, p \ / ~p.
```

```
Theorem peirce_equiv_lem: peirce <-> lem.
```

```
Proof.
```

```
  unfold peirce, lem.
```

```
  firstorder.
```

```
  apply H with (q:=~(p \ / ~p)).
```

```
  firstorder.
```

```
  destruct (H p).
```

```
  assumption.
```

```
  tauto.
```

```
Qed.
```

Рисунок 1.9: Теорема Пирса об исключённого третьего в системе «Coq»

*demonstrandum*» означает, «что и требовалось доказать»). Сначала имеется лишь теорема о равенстве обоих теорем, затем обе стороны равенства развёртываются. Затем `firstorder` пробует сопоставить  $\forall$ -квантифицированные утверждения преобразованные в *нормальную форму Сколема*. Теперь необходимо доказать на правой стороне определения `lem`, что для любого предположительного верного утверждения  $p$ ,  $p \vee \neg p$  также верно. В этом случае, левая сторона является доказуемой гипотезой  $H = \forall p, q. ((p \rightarrow q) \rightarrow p) \rightarrow q$ , которую необходимо доказать для любых утверждений  $p$  и  $q$ . Сейчас  $p \vee \neg p$  заменяется  $p$ ,  $q$  сопоставляется  $p \vee \neg p$ . Таким образом, мы получаем  $(p \vee \neg p \rightarrow \neg(p \vee \neg p)) \rightarrow p \vee \neg p$  — в качестве доказуемой текущей гипотезы, по-прежнему условию, что  $p$  является верным утверждением. Теперь, чтобы доказать верность гипотезы, необходимо следствие `peirce` отделить от предусловия. Необходимо текущую гипотезу абстрагировать и затем опять преобразовать в нормальную форму Сколема, чтобы  $q$  более не являлась квантифицируемой переменной. Затем мы получаем новую более простую гипотезу  $H_0 = (p \rightarrow q) \rightarrow p$ , которую нам удастся доказать, если предположить, что из  $H = \forall p. p \vee \neg p$  правая часть дизъюнкции верная. То есть, мы вводим новую гипотезу  $H_1 = \neg p$ . Такой выбор произвольный и применив  $\neg p$  к  $H_0$ , нас сразу приведет к тому, что  $p$  верное, потому, что импликация всегда верна, как только левая сторона импликации ложная. Так как мы предположили изначально, что  $p$  является верным утверждением, мы доказали правоту теоремы. Тактика `tauto` обязательна для успешного завершения доказательства теоремы, которая из данных существующих гипотез и верных утверждений пытается простейшим

образом, механически найти завершение доказательства без каких-нибудь дополнительных знаний о теореме или используемых лемм.

Обратим внимание на то, что, хотя пример простой и интуитивен, успешное доказательство все-таки требует довольно не малых ресурсов для преобразования в нужную форму. Для применения одной или иной тактики требуется также применение, не очевидных формул абстракций. Не трудно заметить, что автоматически такого рода *нестандартные преобразования* будет очень тяжело выявить и распознать.

**Определение 1.12** (Формальное доказательство). *Доказательством является последовательность применения равенств рассматриваемой теории из аксиом до доказуемой теоремы.*

Если правила Хора полны и удастся провести доказательство из аксиом до доказуемой теоремы, то доказательство может быть *автоматизировано*. Если правила Хора полны, то любая теорема может автоматически доказываться – ответ, либо положительный, либо отрицательный. Чтобы выразить *формальную теорию*, необходимо определить: *семиотику*, *синтаксис* (см. например опр.1.8), *семантику*. Когда речь идёт о языках, а формальная теория обозначается именно выражениями некоторого языка, целесообразно определить *прагматику*. Семантики, например, *аксиоматическая семантика*, обозначают в формальной системе, например в вычислении Хора, какое выражение выводимо или нет. Знаки и взаимосвязи формул являются абстракцией некоторых реальных предметных объектов или физики (классический термин был введён в аналогии реальных объектов, которые принадлежат естественным правилам природы). Поэтому, описание элементов «*физики*» исторически часто называется *метафизикой*, т.е. абстрагированной физикой, либо логикой. Необходимо отметить, что любая *формальная логика* также является по определению специфической *формальной алгеброй*.

Аналогично модульному программированию, доказательства могут строиться с целью упрощения и выявления основных мыслей доказательств. Более подробное обозначение находится в следующих главах. Для аналогии послужат *единицы доказательств леммы* (вспомогательные теоремы), *теоремы* и *индуктивные определения*. Самым простым примером индуктивного определения можно считать естественные числа (см. рисунок 1.10).

```
Inductive nat : Set := 0 : nat | S : nat -> nat
```

Рисунок 1.10: Индуктивное определение чисел с помощью термов Чёрча

Мы не будем отдельно описывать тактики, т.к. тактики лишь вспомогательные образцы последовательности команд при доказательствах. Они лишь имеют некоторый абстрактный характер для упрощения написания доказательств. Абстракция предикатов, для написания и восприятия человеком, имеет большое значение. Можно в общем считать: чем проще доказательство, тем оно лучше. Символы и предикаты могут быть использованы и их определение необходимо развёртывать, лишь

при необходимости. Исходя из текущего состояния вывода и сочетаемых правил, желательно было бы автоматизировать принятие решения системой верификации — когда развёртывать определение и когда свёртывать часть данной формулы обратно к определению. Количество и порядок развёртываний и свёртываний заранее не предсказуемо. По ранее упомянутым причинам, лучше редуцировать термы лениво и снаружи во внутрь, иначе текущая редукция может приостановиться, хотя имеются редексы.

**Наблюдение 1.13** (Модель вычисления верификации). *Единицы доказательств напоминают единицы модулярного программирования. Леммы соответствуют процедурам, главная доказуемая теорема соответствует главной входной процедуре.*

Из рисунка 1.9 можно выявить: следующие проблемы касательно автоматизации доказательств.

1. Описание проблемы соперничает с выразимостью. Это может привести к серьезным ограничениям выразимости и к раздутым описаниям.
2. Упрощение равенств косвенной теории раздувает объём и количество правил Хора, хотя, например, арифметическая теория не связана на прямую с состоянием памяти или программными операторами. Это также препятствует автоматизации.
3. Объяснения при отрицательном выводе, либо отсутствуют полностью, либо желают лучшего, например, интуитивный и обобщённый метод генерации контрпримера.

Подробнее ознакомиться с проблемами автоматизации доказательства теорем можно в статьях Воса [280],[279], несмотря на возраст статей, до сих пор проблемы в основном остаются актуальными. Проблемы Воса можно разбить на три класса: (i) представление модели утверждений, (ii) представление правил вывода, (iii) выбор оптимальной стратегии и тактик логического вывода. Для повышения эффективности верификации, Вос предлагает вводить параллелизацию, индексацию баз данных и знаний для более быстрого доступа и обработки данных, а также анализировать модифицированные технологии вывода. В статьях можно обратить внимание на следующие, часто неявные, замечания Воса:

- Проверка типов может быть полезной с целью избежания статических ошибок, но на самом деле бесполезной для верификации.
- Нотация формул, возможно, не столь важна, но охватить семантику (императивную) программы важно.
- Хотя метод резолюции широко обсуждается в статье, *естественный вывод* все-таки рассматривается как более эффективный, особенно на практике, чем, например *метод резолюции*. Причину этому, наверное можно искать только в близости к классическим математическим доказательствам.

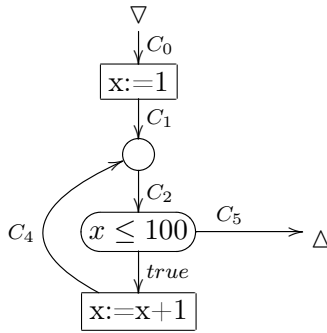
- Существует соперничество между локальным и глобальным поиском оптимума в доказательствах.

Из всех проблем, которые Вос выявляет [280], наиболее важной можно считать избавление от избыточных формул при спецификации, а при верификации от избыточных частей повторного вывода. Главный лозунг Воса: «*Лучше упрощать утверждения, чем перебирать различные варианты*». Эвристика Воса считает в худшем случае лучше с полиномиальной частотой проводить упрощения насколько возможно, чем рисковать экспоненциальный взлет поискового пространства проверки. Вос исходит из 90% сбережений. Престон [212], ссылаясь на Воса, ожидает, что удовлетворительное исследование каждой из первоначальных проблем Воса [279] займёт объём работы, как одна диссертационная рукопись. Овербейк [191], ссылаясь на [279], даёт каждому из перечисленных проблем обзор практических примеров. Мекок [165] является старой, но не устаревшей в целом, обзорной статьёй. Галлье [102] является более актуальной и подробной монографией по теме автоматического доказательства. Его проблемы совпадают большей частью с обсужденными. В [160] Леруа задаёт вопросы к общему процессу верификации и даёт критические оценки с практической точки зрения, его оценки совпадают в основном с упомянутыми. Он, Аппель и Докингс [14] предлагают обществу разработчиков и исследователей соблюдать общие принципы при разработке верификаторов для простого сравнения между собой.

**Абстрактная интерпретация.** Кусо(-вые) [74] [73] предлагают первыми формальный метод (см. опр.1.12) «*абстрактной интерпретации*», как универсальный статический метод, основанный на аппроксимации пошагово вычисляемых интерпретаций данной программы. Метод анализирует граф потока управлений [189] данной программы. Если граф не нормализован, т.е. либо нет ровно одного состояния входа, либо одного состояния выхода, то граф преобразуется именно в такой, объединив входные и выходные состояния. Программные операторы упорядочиваются не строго согласно алгебраической решётке по порядку операторов. С помощью абстракции, интерпретация пытается приблизить лимит [6] ради введения новых зависимых параметров с помощью ограниченного случая (*сужения диапазона видимости*) и с помощью общего случая (*расширения диапазона*). Таким образом, неинициализированные переменные имеют порог  $[-\infty .. \infty]$ . Процесс пошаговой аппроксимации интерпретации считается завершённым, как только, две последовательные интерпретации равны. Интерпретации обновляются после каждого программного оператора. Из-за *проблемы приостановки* интерпретации могут иметь не предсказуемые, т.е. арифметически очень большие, либо очень маленькие лимиты, например, в циклах с неопределёнными значениями переменных при входе в цикл. *Статическое вычисление допустимых порогов значений* [189] является одним классическим применением. Методы Кусо могут быть модифицированы и применены в самых различных областях, например, в [252] для улучшения быстродействия в связи с предсказыванием более оп-



тимального ветвления запуска кода. Чтобы сравнить равенство между двумя абстрактными интерпретациями, вводится условие для изоморфизма изображений. Две интерпретации  $I_1, I_2$  считаются эквивалентными, когда существуют два отображения  $\alpha, \alpha^{-1}$ , которые *инъективны* и *сюръективны*, с которыми интерпретация  $I_1$  преобразуется с помощью  $\alpha$  в  $I_2$ , и  $I_2$  обратно в  $I_1$  с помощью  $\alpha^{-1}$ . Хотя метод Кусо(-вых) универсален, его можно было бы автоматически с трудом преобразовать в правила Хора и наоборот из-за целого ряда причин. Во-первых, метод Кусо предполагает аппроксимацию, а правила Хора знают только точные результаты, либо символьные значения. То есть, можно было бы расценивать систему правил Хора, как частный случай метода Кусо(-вых), хотя цель метода направлен все-таки на арифметическую аппроксимацию интервалов. Во-вторых, для верификации вычисления Хора, используются термовые утверждения некоторого языка спецификации и т.д. Комментированный пример абстрактной интерпретации находится на рисунке 1.11.



Для данного графа потока управлений слева мы высчитываем допустимые диапазоны значения переменных, согласно [74], [73]. Метод присвоения утверждений к блочной схеме был впервые замечен у Флойда [97]. Эбриел [7] представляет собой введение в язык программирования «Би», использующий парадигму присваивания программных операторов к соответствующим значениям.

$$C_0 = [, ]$$

$$C_1 = [1, 1]$$

$$C_2 = C_1 \cup C_4$$

$$C_3 = C_2 \cap [-\infty, 100]$$

$$C_4 = C_3 + [1, 1]$$

$$C_5 = C_2 \cap [101, +\infty]$$

Рисунок 1.11: Блочная схема с присвоенными значениями

Штайнбах [247] демонстрирует наглядно, что терминация является жестким условием, предположением сходимости (например, применив *алгоритм Кнута-Бендикса*) и полноты, а сам вопрос терминации в частных случаях является решимым.

Кроме упомянутых ограничений, отдельно от этого, имеется теоретическое ограничение в связи с *арифметикой Пресбургера*. Арифметика Пресбургера является настоящим подмножеством более известной *арифметики Пиано* над натуральными числами, которая знает только об операции «+» (или инверсная ей операция минус) [244], и где любые другие операции отсутствуют. Это приводит к тому, что «*простые входные программы*» содержавшие плюс, можно в выражениях за конечное время решить в предикатной логике первого порядка (например, в решении офсета для ссылочных указателей, см. позже), в отличие от выражений по арифметике Пиано или сложнее. В частности,

это имеет значение для решимости автоматического доказательства теорем [69], [61] (см. набл.1.13). Решимость выражений из арифметики Пресбургера ограничена  $\Theta(m, n) = 2^{2^{n \cdot m}}$ , где  $n$  минимальное количество предлагаемых разветвлений поиска, а  $m \geq 1$  некоторый константный фактор [95]. Сложность разрастает очень быстро для  $n$ . Важно заметить, что несоблюдение арифметики Пресбургера, например, включение дальнейших операторов, на практике не обязательно приводит к нерешимости частных случаев, однако, соблюдение арифметикой может сильно снизить верхний порог решимости доказуемых теорем.

Поммерель [210] оценивает теоретические критерии сложности и приходит к выводу, что неэффективные реализации прикладных теорий являются самым большим барьером эффективной верификации. К примеру, он приводит состояния моделирование троек Хора в качестве огромных линейных систем уравнений, строки чьи, почти все пустые. Ситуацию можно переломить, если прикладные теории изолировать от правил Хора. Рекомендуется использовать *решатель*, который упростит анализ полноты основных (*логические*) и прикладных правил (*структурные*). Из практического опыта, решатель резко повысит эффективность решения прикладных теорий [188]. Решатели запускаются во время верификации при необходимости. Примерами решателей являются функциональные языки «*Why*» [272] и «*Y-not*» [187].

#### 1.1.4 Альтернативные подходы

Как было упомянуто в начале главы, верификация является формальным методом с помощью которого, можно проверить, соблюдает ли данная программа спецификацию или нет. Верификация троек Хора является статическим методом без запуска программы. В зависимости от точности спецификации для достижения максимально качественного уровня ПО при разрастающейся сложности и функциональности данной системы, на практике может потребоваться добавление, уточнение и сокращение специфицируемых свойств. Необходимо заметить, что сокращение правил происходит всегда тогда, когда правила обобщаются, т.е. отпадает необходимость специфицировать детально. Кроме представленного подхода в прошлых разделах, упоминались ряд иных статических подходов, как, например, *проверка моделей* и локализации проблемы с помощью удаления синтаксического и окружающего кода. Также имеются динамические подходы, как, например: тестирование, безошибочная разработка программного обеспечения и расширенная проверка типов.

**Подход №1 – Тестирование.** Тестирование является динамическим подходом проверки ранее подготовленных сценарий, в которых данная программная система или модуль (не) вычисляет ожидаемый результат. Сценарии это тесты, которые задаются разработчиком вручную и описывают свойства по отдельности. Когда тест успешен, следующий тест проводится до тех пор, пока все тесты выполнены. Тест, это критерий качества, который (не) соблюдается при запуске программы. Если хотя бы один тест не успешен, то тестирование провалилось. Когда сценарий теста известен, то, все, входные данные и выходные результаты, сравниваются согласно данному модулю. Если при про-

ведении теста требуется ручное вмешательство, то тест не автоматизирован. Автоматизация теста (АТ) [24] дает существенные преимущества: высокая эффективность, т.к. за короткий промежуток времени можно проверить большое количество тестов. Далее, исключаются ошибки из-за неточных входных данных. Все внешние зависимости должны описываться тестом. Если тесты простые, то модули можно легко понять и вероятность выше, что ошибки отсутствуют.

Для данной программы  $p$  и набора тестов  $\forall i. t_i(p)$ , каждый из тестов запускается для  $p$  и наблюдается, либо успех, либо провал. Каждый из тестов можно рассматривать как утверждение одного аспекта функциональности ПО. Недостатками АТ можно считать ручную постановку, в зависимости от сложности графа потока управления  $p$ . Число  $n$  необходимых тестов  $p_1, p_2, \dots, p_n$  для покрытия всех ветвей может сильно возрасти. Чем сложнее получается граф в частности, когда граф содержит обратные связи и циклы, тем сложнее ПО в общем.

**Подход №2 – Автоматическая проверка моделей.** Основной проблемой подхода №1 является экспоненциально возрастающий объём тестов, который трудно автоматизировать для представительного множества тестов. Идея проверки модели заключается в описании данной программы с помощью утверждений и временных выражений, которые затем проверяются пошагово, изучив выполнимые входные. Проблема заключается в том, что решения дискретных проблем часто не тривиальные и могут быть выявлены, либо с большими затратами, либо вовсе не могут быть выявлены. Если уравнения сильно ограничиваются, то вероятность высокая, что результаты пропускаются. Если уравнения слишком обобщённые, то результаты тоже могут теряться или быть не найденными из-за отсутствия быстрого нахождения, либо отсутствия решения в принципе. К уравнениям *модели*, которые описывают булеву формулу выполнимости и при правильных результатах всегда верна, применяется стратегия поиска для расширения найденных результатов [63], согласно данной формуле. Отсутствие эффективной стратегии поиска равномерно к незавершению или к очень медленному завершению проверки [143]. Предлагается целый ряд оптимизаций, как например *символьное использование* [63] при работе с *моделями* или статистические методы над контролем расширения [148]. Программные инструменты включают в себя, например «VDM++» [271], [270].

Альтернативно к аксиоматическим правилам можно назвать (автоматическое) тестирование [167] и проверку моделей, как это было впервые введено Пеледом и Кларком [63] ([147] [270] представляют собой обзоры современных инструментов). Преимущество тестирования, в отличие от проверки моделей, является простота проверки поведения программы без дополнительных формул, но, проблема полного перебора всех возможных тестов, остается без изменения в обоих случаях.

**Подход №3 – Безошибочная разработка программного обеспечения.** Альтернативой ранее представленных подходов можно считать «*программирование вообще без ошибок*», при котором не требуется статическая или динамическая фаза проверки, точнее фаза проверки проводится одновременно с моделированием ПО, упираясь на знакомые и «*хорошие*» паттерны [136]. Если ошибок мало или ошибки незначимые с точки зрения быстрогодействия в не критическом месте, то естествен-

но количество ошибок будет стремиться к нулю. Если количество программных строк приближается к нулю, то и количество возможно совершенных ошибок будет приближаться к нулю – эти утверждения всегда в силе, независимо от конкретной программы и не требуют дополнительного объяснения: пустая программа содержит минимальное количество ошибок. Ошибки в программе *создаются* человеком. — Почему? При создании программы полностью исключить «человеческий фактор» по определению не возможно: ПО и спецификация задаются человеком. Ошибки в программах тоже создаются человеком. Отладка и обнаружение возможных ошибок проводится человеком. Абстрактные типы данных, алгоритмы и интуиция — всё это свойственно для человека. Разработка и описание модулей и интерфейсов ПО проводится человеком. Следовательно, не удастся исключить человека при написании ПО, но могут иметься подходы, которые пытаются исключить ошибки на более ранней стадии при создании ПО.

**Разработка ПО.** Вопрос об отсутствии ошибок в ПО на раннем этапе нужно искать при построении и моделировании, например с помощью языка моделирования «*UML*» и «*OCCL*». На практике много раз доказано, что с помощью тщательного моделирования можно избежать множество типичных ошибок, но увы, не всегда и не все виды ошибок это покрывает потому, что:

- индивидуальности предполагаемого разработчика, создание ПО является творческим трудом человека.
- трудности в предсказывании ошибок со ссылками и в том числе, из-за трудности локализации проблем.

**Преобразования.** Задача проверки корректности *трансформаций структур данных* заключается в соблюдении корректного перехода от одного графа к другому [87], соблюдая программные операторы. Подход похож на позже представленную модель динамической памяти в отрезках. Однако, внутри описывается и работает трансформация как «система переписывания», поэтому в дальнейшем она не будет рассматриваться. Системой переписывания является, например «*Stratego XT*» [125] или [292],[87]. [132] предлагает использовать эвристики систем переписывания для лучшей сходимости верификации программ.

**Подход №4 – Проверка типов.** Проверка типов исключает большинство ошибок, которые могут возникать из-за некорректного применения типов переменных и выражений. Например, присвоение 32-разрядного целого числа, 8-разрядному числу плавающей точки, в лучшем случае, может «проглатить» или быть безобидным, а в худшем случае, может привести к совершенно иному числу, если интерпретируются и копируются только все передние 8 от 32 битов.

Задача проверки типов заключается в выявлении проверки совместимости данных и выявленных типов в выражениях программных операторов. Однако, если проверка типов соблюдается, то можно сказать, что семантическая сходимость типов соблюдается. Из совместимости типов не следует утверждать, что данный цикл «правильный». Для того, чтобы можно было это утверждать,

необходимо сначала определить свойство, а затем проверять. Увы, проверка типов является лишь предыдущим шагом перед верификацией. Проверка типов не содержит информацию о зависимостях данных и тем более не содержит никакой информации о том — какие, на каком этапе и сколько объектов будет выделено в динамической памяти. Эта информация проверяется на этапе «*статического анализа*» и является независимой от этапов построения программы:

№	Наименование этапа	входное представление
0		строка программного кода
1	Синтаксический анализ	токены, дерево вывода
2	Семантический анализ	да/нет
3	Генерация кода	дерево вывода
4	Статический Анализ	да/нет, любые структуры данных
4*	Связывание кода (linking)	целевой файл
5	Запуск программы	вывод текста и ошибок на терминале

Рисунок 1.12: Фазы генерации кода

Проверку типов можно задать как « $\Gamma \vdash e : t?$ », где  $\Gamma$  переменная среда содержащая термы вместе с типами, является проверяемым термом и  $t$  является проверяемым типом (см. [22]). Для проверки, имеет ли данная последовательность программных операторов  $e$  тип  $t$ , используя аксиомы и правила типизации  $\Gamma$ , необходимо проверить каждый из операторов в  $e$ . В отличие от троек Хора, термам представляющие программные операторы, присваивается тип, т.е. множество допустимых значений. В отличие от вычисления Хора, проверка типов только устанавливает принадлежности к множеству возможных значений, т.е. к некоторому типу и проверяет сходимость типов. Частное значение некоторого объекта не специфицируется. Имеются исключительно программные операторы и типизированные выражения. При проверке типов отсутствует также понятие как «*состояние вычисления*», при корректной типизации состояния, либо выводимы из *набора правил*  $\Gamma$  типизации, либо нет.

Аналогично вычислению Хора, проверка типов проводится снизу-вверх, т.е. проверка начинается с данного  $e$  и успешно завершается в случае вывода. Проверка типов является проверкой узкого круга свойств программ, т.е. является сильно ограниченной формой верификации троек Хора. Проверка типов программы является обязательной семантической проверкой программы, которая должна проводиться до любых других семантических проверок, как например, верификации свойств динамической памяти. Как мы увидим позже, представленные модели динамической памяти, также как и другие семантические фазы исходят из того, что выражения при присвоении годные, иначе нельзя считать входную программу корректной, как тот язык программирования интерпретируется

строго согласно спецификации, например [243].

**Наблюдение 1.14** (Фаза проверки типов). *Фаза проверки типов должна проводиться перед другими семантическими фазами (как, например, верификации), которые нуждаются в корректной типизации, согласно спецификации языка программирования.*

Если далее сравнивать проверку типов по Хиндлей и Миллнеру [115],[203], [48] с тройками Хора, то необходимо заметить, что при проверке типов, предусловие всегда самое универсальное, а значит верное. Далее, при проверке типов, имеются фундаментальные проблемы, которые являются следующими эквивалентными проблемами при проверке упрощённых троек Хора (см. глава 4 на рисунке 1.13).

Проверка типов по Хиндлей-Миллнеру [178]: $\vdash_{\Gamma} e : t$	вычисление Хора: $\vdash_{\Gamma} \{P\}C\{Q\} \equiv B$
Проверка типов (type checking), дано: $e, t$ : проверить: $\vdash_{\Gamma} e : t$ ?	Проверка доказательства (proof checking), где $\overline{A_{i,j}}$ аксиома тройка, $A_{i,j}$ правило: $B \vdash_{\Gamma} A_{n,0..} \vdash_{\Gamma} A_{n-1,..} \vdash_{\Gamma} \overline{A_{0,..}}$ , дано: всё.
Вывод типов (type inference), дано: $e$ : найти: $t$ ?	Логический Вывод (inference), дано: $B$ , $\Gamma$ , найти: $B \vdash_{\Gamma} A_{n,0..} \vdash_{\Gamma} A_{n-1,..} \vdash_{\Gamma} \overline{A_{0,..}}$ .
Проблема содержимого (inhabitant problem), дано: $t$ , найти: $e$ ?	Проблема содержимого (inhabitant problem), дано: $\{P\}, \{Q\}$ , найти: $C$

Рисунок 1.13: Сравнение проблем между проверкой типов и вычислением Хора

Были предприняты расширения, которые позволяют: проверять совместимость записей, ужесточение типов, свойства неприсвоенности и статические лимиты объёмов базисных структур данных [96]. Но для проверки свойств динамической памяти, увы, даже те упомянутые расширения, совершенно здесь не достаточны. Например, приведем *правило последовательности*:

$$(\rightarrow\text{-elim}) \frac{\Gamma \vdash e_1 : (s \rightarrow t) \quad \Gamma \vdash e_2 : s}{\Gamma \vdash (e_1 e_2) : t}$$

Правило ( $\rightarrow$ -elim) наглядно демонстрирует, что указание типа (например,  $s \rightarrow t$ ) может служить как элементарное утверждение процесса верификации (см. [58], [186]). В [22] широко представлены доказательства всех важных теорем и лемм  $\lambda$ -вычисления в связи с определением его синтаксиса и

семантики, в том числе, кратко о проблемах и ограничениях не типизированного и типизированного  $\lambda$ -вычислений. Статья [207] посвящается вопросам теоретической нерешимости редукции  $\lambda$ -термов в общности без явной типизации (например, в вычислениях Карри, в отличие от вычислений Чёрча). Например, терм  $\lambda x.xx$  не может быть решён в общем, но при типизации становится очевидно, что данный терм не принадлежит регулярной типизации. Если бы  $\lambda x.xx$  был типизирован, то допустимый второй терм  $x$  имел бы некоторый тип  $t_2$ , а первый  $x$  обязательно должен иметь некоторый тип функционала  $t_0 \rightarrow t_1$ , что невозможно приравнять к  $t_2$  в типизированном  $\lambda$ -вычислении первого порядка, в котором типы далее не специфицируются. Следовательно,  $\lambda x.xx$  не типизируемый терм. Однако, можно сконструировать в качестве бесконечно приближенного типа лимит [166]. Этот тип соответствует рекурсивно определённому абстрактному типу данных, который имеет в общем случае зависимость (возможно, рекурсивную) к другим типам данных и представляет собой тип данных записи.

**Наблюдение 1.15** (Нередуцируемость верификации к типизации). *Проверка типов не может быть в общем редуцирована к проблеме верификации в вычислении Хора, однако, обратное действительно.*

**Подход №5 – Автоматическая редукция проблемы:** Автоматическая редукция проблемы пытается редуцировать строки программы, чтобы данная ошибка, либо иное поведение программы, например «результат функции возвращает 5», сохранялось. То есть, если некоторая программа вычисляет некоторую таблицу, используя различные формулы, а нас интересует только одна ячейка в ней (например, единственная с ошибкой), то можно весь код удалить, который не нужен для вычисления той самой критической ячейки. Таким образом, получается редуцированная и более простая программа, которую можно легче анализировать. Очень часто бывает, что ошибка в программе связана лишь с маленькой частью изначальной программы. Увы, особенно в больших или незнакомых программах, локализация программ занимает очень много лишнего времени.

Допустим, мы строим проект одной командной строкой. Вместо «*GNU make*» [100] ради простоты, используется сильно упрощённая программа для построения любого ПО «*builder*» [304]. Для автоматизации и редукции входной программы используется программа «*shrinker*» [305]. Программа работает циклически: строит ПО, анализирует или запускает ПО, сравнивает наблюдаемое поведение программы с ожидаемым результатом. Если программа после сокращения по-прежнему строится, запускается и сравнение успешное, то редукция продолжается, иначе редукция проваливается. Если редукция проваливается, то, либо берётся другая редукция, либо прежняя (успешная), как окончательная. Программа при запуске выдаёт всё, что важно на «*stdout*» и «*stderr*». Уговаривается по умолчанию, что любое другое поведение можно записывать в «*stdout*» или «*stderr*», в том числе, интересующее нас поведение программы (так называемый «симптом»), без ограничения общности. Если симптом всё ещё наблюдается при запуске программы, то поведение инвариант и мы «про-

бует» сокращать, далее пока не останется возможности. Предложенный подход описывается на рисунке алгоритм №1. Представленный алгоритм наивный и не оптимальный, т.к. выбор *lineseq* не детерминированный и может содержать любое количество строк, равно хотя бы одной или более строк. Нужно заметить, что объём выбранных строк может расти, но также уменьшаться со временем, когда варианты редукции отклоняются несколько раз подряд, в зависимости от стратегии редукции.

---

**Алгоритм 1** Наивный подход редукции программы по строкам по [305]

---

```

1: procedure NAIVESHINKER(prog, symptom)
2:   newProg  $\leftarrow \emptyset$ 
3:   oldProg  $\leftarrow prog$ 
4:   for all loc(newProg)  $\leq loc(oldProg)$  do
5:     (newProg, oldProg)  $\leftarrow$  (REDUCE(oldProg, symptom), newProg)
6:   end for
7: end procedure

8: procedure REDUCE(prog, symptom)
9:    $\exists lines \leftarrow lineseq(prog)$ 
10:  if RUN(prog  $\setminus$  lines) == symptom then
11:    return prog  $\setminus$  lines
12:  else
13:    return prog
14:  end if
15: end procedure

```

---



## 1.2 Объектные вычисления

Объектные вычисления представляют собой формализм для различных вычислений с объектами. Вычисления могут в себя включать, например, совместимость типов, проверку верности условий. Объектные вычисления можно разделить на два вида, по Абади-Карделли и по Абади-Лейно (см. рисунок 1.14).

Вид	Наименование	Пример
(1)	Абади-Карделли [3] классный экземпляр объекта	<pre>class MyClass{ int A; int B; MyClass C; }; MyClass o1 = new MyClass(); o1.A=1; o1.B=2; o1.C=NULL; MyClass o2= new MyClass(); o2.A=5; o2.B=3; o1.C=o2;</pre>
(2)	Абади-Лейно [155], [4], [2] обыкновенный объект (не-классовый)	<pre>object o1 = [ A=1; B=2; C=[A=5; B=3] ];</pre>

Рисунок 1.14: Виды объектных вычислений

Итак, зачем на самом деле нужна формализация объектов? Почему нельзя сопоставить объекты простыми переменными? Отвечая на второй вопрос первым: да, можно сопоставить на самом деле. Однако, в связи с введением объектов поля, больше абстрагируются. Вследствие этого, алгоритмы упрощаются и обобщаются. Класс объектного экземпляра, это прежде всего, «*абстрактный тип данных*» (АТД), а не только хранитель данных. АТД также является носителем множества объектов, предусмотренные для решения комплексной проблемы (см. [156]). Операции над объектом замкнуты, т.е. сам объект не может распадаться на некоторые другие объекты при запуске операций. Поэтому программист, при написании обязуется, определять в объектах лишь те операции, которые к нему подходят и которые действуют только на внутреннем состоянии всех полей самого объекта – поля других объектов не доступны. Это подразумевает разделение данных аналогично записям. Операции объектов на практике, это методы классов, а объект, это объектный экземпляр.

*Объектно-ориентированная парадигма* популярная, и успешно использовалась в индустрии на

протяжении последних десятилетий, опираясь на АД. Одним из преимуществ АД является — возможность строить надёжное, гибкое, интуитивное и быстрое ПО за счёт технологии моделирования, как например «*объектно-ориентированное моделирование*» (ООМ) [77], [229] или «*паттерны*» [136]. Перечисленные методы и идея широко применяются в индустрии до ныне. Нельзя недооценивать важность использования паттернов на практике, которые способствуют к существенному снижению ошибок при написании кода, благодаря четкому распределению ролей вовлечённых объектов. Паттерны программирования являются *стереотипами*, т.е. эпистемологическим механизмом - идиомой, которая способствует человеку «*быстрее*» воспринимать во времени вовлечённые объекты и связывать их, для решения поставленной задачи, не читая весь программный код полностью. Многие классические, философские идеи и течения вовлечены в представление объектов, как, например атомизм, коннективизм и т.д. Паттерны можно интерпретировать, как сравнение объектов ООМ с ролями *актёров* сценария, для решения определённого сценария: каждый актёр в определённой ситуации ведёт себя так, как этого требует «манускрипт», точнее спецификация. Прагматизм паттерна повышается, если сценарий можно объяснить лучше и понятнее.

Объектное вычисление может быть использовано для семантических анализов, например для проверки типов или для верификации, что будет представлено чуть позже. Кроме того, оно может послужить анализу зависимости полей данного объекта, например для эффективного *распределения процессорных регистров*. В частных случаях может иметь смысл, все объектные поля преобразовать в массив или отдельные локальные переменные, если граница массива известна и не велика [44] и размер объекта не велик. При распределении регистров, зависимость данных на прямую определит какие регистры будут использованы (например, регистры общего пользования) или стековые поля. Быстрые ресурсы, т.е. процессорные регистры, в разы быстрее, чем доступ к операционной памяти, однако, они сильно ограничены по объёму и ёмкости. С другой стороны, процессорные регистры, в зависимости от архитектуры ЭВМ, могут, согласно *двоичному интерфейсу приложений «ABI»* (см. [252], [253], [160]), временно *хранить* только несколько *слов*. Слова ограничены и тем не менее, имеющийся *ABI* обычно не делит общие блоки памяти представляющие объектные экземпляры потому, что при компиляции трудно предугадать точно, когда и где, какие поля будут или не будут использованы. Однако, если возможно было бы это уточнить, стоило бы модуль предсказания «*GCC*» в этом плане улучшить, т.к. на данный момент всегда принимается отрицательный прогноз. Реальность такая, что объекты остаются неделимыми и если памяти не остаётся, то всё выкладывается на стек, что тормозит и желает иметь лучшее.

Вычисление записей [90] отличается в основном от объектов тем, что они являются чистыми хранителями данных. Запись можно рассматривать, как кортеж различных типов, что может быть полезно для описания вычислений одновременных процессов. Скотт [236] характеризует любой тип данных (т.е. в обобщённом виде, в качестве кортежа) как *алгебраическую решётку*, которая имеет *инфимум* как изначальное (возможно не инициализированное) значение и окончательный резуль-

тат (возможно параметризованный), как *супремум* и все состояния между обоими *экстремальными точками* объясняются, как состояния кортежей, которые соблюдают *порядок вычисления*. Для более подробного ознакомления применения и обоснования верификации с объектами, особенно в связи с нединамическими переменными, можно ознакомиться в [58] и [182]. Для дальнейшей дискуссии в качестве примера мы приведем работу [23]: условия верификации задаются формулами логики предикатов первого порядка. Атрибутные поля моделируются, либо отдельно, т.е. атрибуты выделяются в отдельные регионы памяти, либо встроено в содержащий объект, т.е. все поля и типы известны. Наиболее комфортабельным вкладом работы можно считать предикаты **pack** и **unpack**, которые свёртывают, либо развёртывают предикат вручную. Аналогично к [23] и [58], рекомендуется моделировать зависимые поля, как отдельные зависимые объекты. Обе работы предлагают ввод новых утверждений для инвариантов объектов, которые всегда верны до тех пор, пока некоторый объект жив. Никакой из упомянутых подходов не имеет отношения к *указателям* (см. глава 2).

Вычисление объектов классных экземпляров упоминаются только коротко. Этот вид вычисления доминирует современные языки программирования, как, например, Ява или Си(++) [252], [243]. Класс используется для генерации объектных экземпляров при запуске программы. В этом разделе нас пока не интересует область выделенной памяти. Используются командные операторы **new**, **malloc** и **calloc**. После короткой характеристики, мы переходим к обыкновенному виду вычисления объектов.

Оба вида вычислений представляют возможность проверки типов и спецификации/верификации программных операторов. Абади и Карделли мотивируют предложенное вычисление необходимостью формализовать вычисление с объектами из-за различных эффектов, как *подклассы*, *полиморфизм* и *замкнутость/рекурсия объектных типов*. Оба автора ссылаются на систему типизации лямбда-вычислений как одним примером второго порядка. Нужно заметить, что  $\lambda$ -термы, представляющие объекты, могут быть использованы, как квантифицируемые типы (т.е. типовое множество классов, которое выражается знаком  $\forall$ , см. [203], [48]) с помощью неопределённых символов. Далее, рекурсивные и нерекурсивные классы требуют уточнения структуры. Другими словами, «*зависимые от новых параметров типы*», которые представляются абстракцией  $\lambda$ -типов высшего порядка дополнительную абстракцию над объектными типами по Абади-Карделли.

Без ограничения модели памяти, например, в каком регионе операционной памяти содержится объект, или, где объект расположен в соответствии материнского объекта: внутри и снаружи, любой объект по модели Абади-Карделли имеет указатель. На рисунке 1.15  $x$  указывает на объект, который имеет один внутренний объект, а он опять же имеет два содержания  $abc$  и  $def$  и некоторое содержимое  $ghi$ . Кроме того, имеется указатель  $y$ , который ссылается на объект содержащий  $jkl$ , при этом, объект  $x$  содержит некоторое поле (или любое из подполей одного из полей), которое ссылается на содержимое от  $y$ , т.е.  $jkl$ .

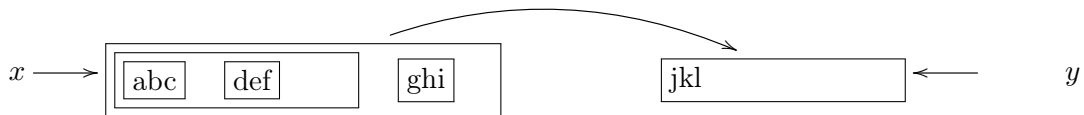


Рисунок 1.15: Указатель  $x$  ссылается на объект, чьё содержимое ссылается на  $y$

Классная иерархия наследования представляет собой *упорядоченное множество*, инфимум чей по конвенции пустой класс, например `Object` или `[]`. То есть, любые два класса из иерархии можно сравнить с самим собой, два класса, которые ничего не связывает и не связаны между собой, т.е. имеются следующие сравнения порядка наследованности:  $\sqsubseteq$ ,  $\not\sqsubseteq$ .

Абади и Карделли лишь предоставляют возможность наследования в своей модели вычисления, также как и *делегация* или *внутреннее* представление полей и объектов. Использование ключевых слов `self` и `super` в языке Си++ [243] допускается, однако, использование подлежит к заранее проводимому семантическому анализу контекста ссылаемого класса. Эта предосторожность сильно увеличивается, вместе с целым рядом других способов вычисления, при использовании обыкновенного объектного вычисления. Карделли замечает, что дополнительные накладные расходы в связи с проверками актуального и ссылочного класса, например, при использовании `self`, приводят к принудительным проверкам всей иерархии классов, которые можно было бы избежать, если контекст достаточно однозначен. Нужно отметить, что на практике это замечание не столь важное потому, что определение класса по иерархии не является критической проблемой при компиляции программы. Поиск наследованности по иерархии слияния фактически является поиском по дереву, что можно сделать за счёт  $\Theta(n) = \log(n)$  в худшем случае. Это вполне достаточно. Серьёзным ограничением наследованности, является угадывание, какой именно метод будет запускаться. С вопросами практической и теоретической реализации наследованности множества классов на языке Си++ можно ознакомиться в работе Рамана [216].

Методы объектов содержат код, который доступен только объекту. Ради простоты, мы смоделируем автоматически выделенные методы, которые в отличие от статических методов, означают, что методы доступны исключительно тогда, когда соответствующий объект был ранее выделен оператором `new`. В случае статичных классов, нет необходимости доступа к заранее выделенной памяти объекта, т.к. переменные в стековом окне выделяются глобально независимо от объекта. Далее, обозначим по умолчанию, что любой объект, если не уговорено по другому, всегда имеет один метод «конструктор» для построения объекта и инициализации нейтральным значением всех полей (что записывается в отдел `.ctor` на ассемблере, в случае языка Си++ [243] при переводе) и один аналогичный метод утилизации объекта, который записывается в «`.dctor`». Таким образом, можем определить следующие фазы объектов (см. рисунок 1.16), которые нам послужат в качестве специ-

фикации:

1. Конструктор
2. Деструктор
3. Постоянное присутствие
4. Предусловие метода
5. Постусловие метода
6. Любое дополнительное утверждение в методах

Рисунок 1.16: Специфицируемые этапы объекта

Только что были обсуждены пункты 1 и 2. Пункт 3 касается любого состояния *«немёртвого объекта»*. Объект немёртвый, когда он *«жив»*, это когда программный оператор вызывается после момента конструкции, но до утилизации объекта. Пункты 4 и 5 подразумевают ту самую спецификацию, как у процедур. Ради применимости пытаемся избежать принудительную спецификацию всей программы. Обозначим это *«лёгким»* подходом, но пока что не будем рассматривать такой вариант. В целях увеличения гибкости и локализации проблем, уговаривается по умолчанию, что любые *«дополнительные»* утверждения разрешаются, которые также могут быть упущены (пункт 6).

Методы в обоих вычислениях изначально не ограничиваются. Однако, не ограниченные методы также могут включать в себя изменение методов в любой момент запуска программы, удаление существующих методов и добавление новых методов. Очевидно, что при неограниченном вычислении спецификация становится всё сложнее, и возможные изменения трудно, или даже теоретически и практически не решимы. Следовательно, максимальная возможность приводит к жесткому ограничению верификации.

Как было упомянуто, неограниченные методы не дают и не расширяют с теоретической точки зрения *«Тьюринг-вычислимость»*, изменения методов при запуске (*«динамические методы»*) и умолчанию далее не рассматриваются. Когда речь идёт о динамических методах, только тогда, они могут рассматриваться.

Системы объектных вычислений имеют следующее логическое суждение:

$$E \vdash obj : A :: B$$

где,  $E$  объектная среда, содержащая существующие объекты в момент актуального состояния вычисления программы,  $obj$  является объектным представлением (т.е. переменной в вычислении №1, а целым объектом в случае вычисления №2).  $A$  является типом  $obj$  (т.е. наименование класса объектного экземпляра  $obj$ ).  $B$  является спецификацией объекта  $obj$ , которая содержит, например, состояние объекта. Если вместо  $obj$  проводится вызов метода  $obj.m()$ , то  $B$  содержит состояние  $obj$

до и после вызова  $m$ , но  $B$  также может далее содержать утверждения из рисунка 1.16, которые не противоречат данной фазе программного запуска. На результаты до и после запуска программного оператора или метода, можно ссылаться с помощью встроенных вспомогательных операторов и регистра специального назначения, который содержит, либо весь объект (в случае №2), либо объектное поле. В центре объектного вычисления стоит объектное суждение вместе с типом и утверждением. Отметим, что  $B$  может меняться и при этом  $A$ , т.е. типизация, остаётся без изменения. Это становится актуальной проблемой, когда сигнатура/тип дальнего метода знакома, но само состояние нет. Это тот случай, когда используется архитектура веб-сервера или драйвера, когда известно как метод должен выглядеть, но детали функциональности не (полностью) известны.

Отметим, что обе модели, Абади-Карделли и Абади-Лейно, не поддерживают изначально ссылочные указатели на любые объекты, которые были бы выделены во время запуска программы. Указатели будут рассматриваться позже, однако, уже сейчас обсудим, что избегая лишние операции копирования, можно существенно улучшить быстродействие, если гарантированно, что некоторые объекты разрушаются и эти объекты не будут использоваться далее, или все копии объекта остаются без пользы.

Классовые типы  $T$  определяются рекурсивно. Тип — либо целое число, либо состоит из классов. Классовый объект содержит поля  $f_i$  и методы  $m_j$  и все наименования отличаются друг от друга согласно конвенции. Все поля имеют некоторый тип  $T$ . Методы имеют  $T_j \rightarrow T_{j+1} \rightarrow \dots \rightarrow T_k$  в качестве типа сигнатуры.

Далее рассматривается вычисление №2, объектное вычисление по Абади-Лейно.

Язык программирования «*Baby Modula 3*» [2] был предложен Абади и компанией «*Digital Equipment Corporation*» в качестве экспериментального модулярного языка. Он содержит минимальный набор операторов языка *Modula 2*, который очень близок к синтаксису и семантике языка Паскаль, с дополнениями для прототипизации языка объектного вычисления обыкновенного вида. Язык похож на предложенные и далее расследованные языки Абади, Карделли, Лейно и содержит все необходимые единицы для  $\mu$ -рекурсивных схем: присвоение и рекурсию. С более подробными дискуссиями на тему модели объектов и выразимость объектов можно ознакомиться в [111], где имеются чуть устарелые, но значимые статьи по данному вопросу. Целью ввода иного вычисления является альтернативное представление формализации. Однако, имеется эквивалентность обоих видов, которую можно доказать методом «*полной абстракции*» [179],[209],[65],[117], при которой необходимо доказать эквивалентность денотационной и операционной семантик [9],[290],[6],[276],[250],[34], касательно наблюдаемого поведения объектов. Доказательство изложено в [220],[222] и следует отметить, что в целях простоты модель Абади-Карделли описывается в разы проще. Кроме того, попытка доказать полную абстракцию увенчается успехом, но за счёт очень сложной операционной семантики в объектном вычислении. Это можно обосновать тем, что возможные рекурсивные типы гораздо проще

вывести, если рекурсия содержится лишь в наименованиях типов, т.е. классов, чем определяется рекурсией объектных экземпляров.

Синтаксис языка «*Baby Modula 3*» определяется с помощью термовых выражений  $a$  как определено на рисунке 1.17, где  $A$  является типовым выражением.

$a ::= x$	.. переменная
$\text{fun}(x : A)b$	.. декларация процедуры с телом $b$
$b(a)$	.. вызов процедуры $b$ с параметром $a$
$\text{nil}$	.. пустое значение типа
$a[f=b, m=c]$	.. объектное расширение
$a.f$	.. доступ к полю
$a.m$	.. доступ к методу
$a.f:=b$	.. присвоение поля
$a.m:=c$	.. присвоение метода
$\text{wrong}$	.. ошибочный тип

Рисунок 1.17: Определение объектно-термовых выражений по Абади-Лейно

Термовое выражение эквивалентно к выражениям языков «*Modula*» или «*Паскаль*» и не нуждается в дополнительных объяснениях, кроме:  $\text{fun}(x : A)b$  определяет анонимную процедуру (эквивалентно к  $\lambda$ -абстракциям) с одним входным параметром  $x$  с типом  $A$ . В общем разрешаются анонимные процедуры с нуля или более параметрами. Переменная  $x$  является в теле процедуры  $b$  связанной переменной. Вызов процедур, например  $b(a)$ , подразумевает, что типы последующих аргументов совпадают с типами декларации процедуры. Пустое значение  $\text{nil}$  совместимо со всеми другими типами указателей. Например, не инициализированные объектные поля равны  $\text{nil}$ . Объектное расширение добавляет поля с указанным значением к предлагаемому объекту, таким образом, что объекту добавляется новое поле. Методы также расширяемы. Метод может обновляться, т.е. код метода может также как и переменная меняться во время запуска программы.  $\text{wrong}$  является резервированным ключевым значением, который показывает на неверное вычисление. Данным определением объекта всегда является вектор, который состоит из лексикографически упорядоченного множества пар из наименований полей/методов вместе с содержанием. Содержание опять же может быть объектом, которое определяется аналогично определению из рисунка 1.17.

Семантика «*Baby Modula 3*» определяется аксиоматически и проводится двумя шагами: сначала проводится проверка типов (один набор правил), затем верификация с другим набором правил. Проверка типов объектов также требует определение на подкласс, которое может проводиться по-компонентно:  $A <: B$  тогда и только тогда, когда  $B$  содержит все поля и методы с наименованиями как в  $A$ . Из-за отсутствия классовых идентификаторов в объектном вычислении, рекур-

сивные объекты необходимо симулировать подобно случаю «комбинаторам плавающей точки» в  $\lambda$ -вычислениях, которые имеют похожее ограничение. Итак, смешано-рекурсивные объектные определения выявляются с помощью искусственного комбинатора  $\mu(X)$  для любого объекта  $X$  к данному объектному выражению согласно рисунку 1.17, который приписывается после  $\mu(X)$   $\tau$ , таким же образом, как аппликация  $\lambda$ -термов, например,  $\lambda x.x\tau$ .

Следовательно, для проверок объектов имеются: переменные среды, подтипы (в том числе подклассы) и общие структурные правила. Ими производятся проверка типов и верификация объектов (см. позже, ср. минимальную логику вычислимости «*LCF*» [209]). Ради простоты (см. [155]) часто ссылаются на структурные операционные семантики [208], хотя, как было упомянуто ранее, преобразование в/из денотационной семантики(-е) в общем случае остается тяжелой проблемой.

Однако,  $\mu$ -определения объектов — существенная проблема, т.к. для одних и тех же начальных данных вывод может быть различным, т.е. некорректность очевидна. Приведем в качестве примера правила из рисунка 1.18.

$$\frac{\begin{array}{l} \vdash \text{nil} : \mu(X)\text{Root} \\ \text{Self} = \mu(X)\text{Root} \end{array}}{\vdash \text{nil} : \mu(X)\text{Root}} \quad \frac{\vdash \text{nil} : \text{Root}}{\vdash \text{nil} : \mu(X)\text{Root}}$$

Рисунок 1.18: Пример набора правил, которые некорректны

То есть, правила с рекурсивными определениями могут быть подорваны и недоказуемы из-за возможных расходящихся доказательств, при этом, лишь тривиальное решение, т.е. неподвижная точка, может действовать единственно «надёжным» путём. Такая обстановка, увы, не удовлетворительна. Это как раз и есть причина, почему объекты в вычислениях Абади-Лейно в общем не могут быть типизированы в вычислении типов первого и второго порядка. Разрешить эту проблему возможно, если правила верификации и проверки типов обогатить дополнительными правилами, которые сильно зависят от контекста. Это резко ухудшит простоту правил. По Абади-Лейно классы как таковы, не существуют. Объекты могут иметь произвольное число полей других объектов, следовательно, классы могут зависеть от других классов, включая от собственного класса. Это порождает тип третьего порядка, а вычисление эквивалентно  $\lambda_3^{\rightarrow}$  (см. опр.1.11). Поэтому, сходимости и замкнутости по типу имеет большое значение для решения при статическом анализе классового типа [2]. Если сходимости конечная и процесс верификации перечисляем, то таким образом, можно сложить все перечисленные типы вместе, в одну запись. Таким образом, новой записью является тип объединяющий все зависимые типы. Авторы [2] обращают внимание на то, что любые операции полученные объединением типа замкнуты и все полученные (под)-типы также могут использоваться как базисный объектный тип. То есть, полученный объектный тип является идеалом, опираясь на сравнение объектов «<:», см. позже.



Язык программирования представленный в [4] (см. рисунок 1.19) является упрощением языкового представления рисунка 1.17, однако, присвоение отсутствует, а вместо этого имеется символьное присвоение и некоторый дополнительный *синтаксический сахар* условного перехода. Далее, динамические методы запрещаются. Синтаксис определяется следующим образом:

$a, b ::=$	$x$	.. переменная
	$  \text{ false }   \text{ true }$	.. ложное/верное утверждение
	$  \text{ if } x \text{ then } a_0 \text{ else } a_1$	.. условный переход
	$  \text{ let } x = a \text{ in } b$	.. символьное присвоение
	$  [f_i = x_i^{i=1..n}, m_j = \psi(y_j)b_j^{j=1..m}]$	.. объектный конструктор
	$  x.f$	.. доступ к полю объекта
	$  x.m$	.. доступ к методу объекта
	$  x.f := y$	.. присвоение поля объекта

где  $x, y, z, w$  являются переменными,  $f, g$  поля объекта  $x$ ,  $m$  является методом.

Рисунок 1.19: Упрощённые объектные по Абади-Лейно

Переименование во время запуска разрешается, если оно соблюдается вызывающей и вызванной сторонами. Локальные переменные в процедурах запрещаются. Также запрещаются параметры методов, которые меняются внутри методов. Для выразимости это не является ограничением, т.к. это всегда можно симулировать с вводом дополнительных полей. Особенность присвоений заключается в том, что присвоения полей возвращают в качестве результата объект, в котором некоторому существующему полю было присвоено значение. Важным ограничением в отличие от [2] является по Абади-Лейно в частности, исключение рекурсивно-определённых объектов.

Отношение подтипов определяется в рисунке 1.20.

$$\begin{aligned} \vdash A <: A' \quad \Leftrightarrow \quad & [f_i : A_i^{i=1..n+p}, m_j : B_j^{j=1..m+q}] = A \wedge \\ & \wedge [f_i : A_i^{i=1..n}, m_j : B_j^{j=1..m}] = A', \quad p, q \in \mathbb{N}_0, \end{aligned}$$

Рисунок 1.20: Подтип класса

где  $A'$  это некоторый класс, а  $A$  соответствующий подкласс.

Соотношение  $<:$  позволяет вместе с другими правилами проводить проверку типов, например, самое главное — это *правило объектного построения*:

$$\text{(CONS)} \frac{\begin{array}{c} A \equiv [f_i : A_i^{i=1..n}, m_j : B_j^{j=1..m}] \\ E \vdash \diamond \quad E \vdash x_i : A_i^{i=1..n} \quad E, y_j : A \vdash b_j : B_j^{j=1..m} \end{array}}{E \vdash [f_i = x_i^{i=1..n}, m_j = \psi(y_j)b_j^{j=1..m}] : A}$$

где  $A$  некоторый объектный тип (не класс),  $E$  это объектная среда,  $x_i$  некоторые значения полей, а  $b_j$  некоторые значения методов, т.е. тела процедур. Проверка верности программы по упрощенному Абади-Лейно проводится согласно следующему суждению:

$$\sigma, S \vdash b \rightsquigarrow r, \sigma'$$

где  $\sigma$  является начальным состоянием стека,  $S$  описывает состояние стека,  $b$  данная последовательность программных операторов («программа»),  $r$  является результатом, который сохраняется в виртуальном регистре неограниченной ёмкости, и  $\sigma'$  описывает финальное состояние памяти. Спецификации объектов определяются рекурсивно по компонентам:

$$[f_i : A_i^{i=1..n}, m_j : \psi(y_j)B_j :: T_j^{j=1..m}],$$

где  $A_i, B_j$  это спецификации,  $y_j$  параметры анонимных процедур  $\psi$  используемые в  $B_j, T_j$ , а  $T_j$  определяет спецификации перехода памяти из одного состояния в следующее.

Суждение спецификации определяется по  $E \Vdash a : A :: T$ , где  $E$  это объектная среда, программа,  $A$  тип,  $T$  переходное описание. То есть, проверка типа по  $A$  и переходные состояния записаны в  $B$  и применяются одновременно, если даже технически по разным этапам, всё равно не трудно увидеть, что  $B$  окажется сильно раздутым и незначительное изменение памяти может привести к большим последствиям. Предложенное соотношение  $<:$  [4] аналогично применяется к спецификациям. Аналогичные проблемы распространяются на спецификации. Например, требуется доказать очевидную программу:  $\emptyset \Vdash ([f = \text{false}].f := \text{true}).f : \text{Bool} :: (r = \text{true})$ . Дерево доказательства с аннотациями использованных правил находится в рисунке 1.21.

$$\begin{array}{c} \text{(FUpd)} \frac{\text{(CONS)} \frac{\text{(env1)} \frac{}{1} \quad \text{(const1)} \frac{}{2}}{3} \quad \text{(const2)} \frac{}{4}}{\text{(FSel)} \frac{5}{6}} \end{array}$$

- 1  $\emptyset \Vdash \diamond$
- 2  $\emptyset \Vdash \text{false} : \text{Bool} :: \text{Res}(\text{false})$
- 3  $\emptyset \Vdash [f = \text{false}] : [f : \text{Bool}] :: \text{Res}([f = \text{false}])$
- 4  $\emptyset \Vdash \text{true} : \text{Bool} :: \text{Res}(\text{true})$
- 5  $\emptyset \Vdash ([f = \text{false}].f := \text{true}) : [f : \text{Bool}] :: \text{Res}([f = \text{false}].f := \text{true})$
- 6  $\emptyset \Vdash ([f = \text{false}].f := \text{true}.f : \text{Bool} :: (\underbrace{r = \text{true}}_{= \delta([f = \text{false}].f := \text{true}, f)}))$

Рисунок 1.21: Деревя вывода для примера объектного вида программы

Определения использованных правил находятся в рисунке 1.22.

$$\begin{array}{c}
\text{(FSel)} \frac{E \vdash x : [f : A]}{E \vdash x.f : A} \quad \text{(FUpd)} \frac{E \vdash x : A \quad E \vdash y : A_k^{k=1..n} \quad A \equiv [f_i : A_i^{i=1..n}, m_j : B_j^{j=1..m}]}{E \vdash x.f_k := y : A} \quad \text{(env1)} \frac{}{\emptyset \vdash \diamond} \\
\\
\text{(const1)} \frac{E \vdash \diamond}{E \vdash \text{false} : \text{Bool}} \quad \text{(const2)} \frac{E \vdash \diamond}{E \vdash \text{true} : \text{Bool}}
\end{array}$$

Рисунок 1.22: Пример набора правил для объектного вида объектного вычисления

Утверждение  $\diamond$  обозначает истину. Корректность представленного набора правил можно прочитать подробнее в статье [2] и в её сопровождающей технической статье. В подходе Абади-Лейно необходимо отметить, что ситуация с абстракцией желает иметь лучшее. Параметры в методах видны только снаружи во внутрь, но не наоборот, например в  $b_1 \equiv \text{let } x = (\text{let } y = \text{true in } [m = \psi(z)y]) \text{ in } x.m$  внутренний  $y$  не может быть проверен снаружи, согласно правилам из [2] и разница между предикатами (какого именно порядка и ограничения) и утверждениями всё-таки не достаточно ясна. Абади и Лейно предлагают в качестве дальнейшей работы следующее: улучшение абстракции спецификации, исследование меняющихся параметров, а в связи с этим, вопросы о полноте, использовании указателей на объекты и на поля объектов, а также более детально разобраться с рекурсивно-определёнными объектами. На данный момент не вычисление по Абади-Карделли и по Абади-Лейно не обращают внимание на указатели или *псевдонимы* (см. позже).

Оба вида вычислений страдают от того, что отсутствует поддержка указателей. В [3] перечислены важные и нужные расширения: поддержка параллельности, возможность ссылаться на адресные поля и использования абстракции в спецификациях, так как впервые это уже заметил Хор [116].

Банэрий [21] представляет язык, который поддерживает объекты, разместившиеся в стеке и в том же регионе памяти (см. [258]). Подход в [21] перемещает все локальные переменные в стек, но висячие указатели по определению языка не допускаются. Рекурсивные предикаты над объектами запрещены. Особенность глобальных инвариантов заключается в не меняющихся зависимостях между объектами. Он предупреждает о сильно возрастающей проблеме абстракции и поддерживает инициативу по-объектного взгляда индивидуальных предикатов.

### Программные нити.

В [4] объекты квалифицируются как абстрактные типы данных, но только не как обыкновенные вычисления записей [90]. Отличие определяется в первую очередь не по имеющимся полям, а по структуре данных. Записи имеют любые зависимости к иным записям.

Переходы из одного состояния записи в другое, можно получить с помощью применения правила редукции. Таким образом, *трансформации графа* можно рассматривать как *редукции с записями*. Эриг и Роузен предлагают подкласс процедур, который безопасен тем, что он не смешивает зависимые поля объектов вместе с анализом с помощью теории категории. Их подход можно считать «безопасным» при запуске нескольких нитей одновременно, если доступ к подмножеству полей объ-

екта не зависит друг от друга. Таким образом, авторы дают возможность к более эффективному доступу полей, без необходимости полной блокировки при много-поточных программах. Поэтому, процедуры порождения и утилизации, при соблюдении условия, могут запускаться одновременно без изменения поведения и корректно. Похожие подходы к доступу объектов представляются более подробнее в [127].

Хойсман [120] задаётся также вопросом параллельного доступа к объектам, также как и Хурлин [121] и [90]. Хойсман пытается с помощью «*контрактов между объектным взаимодействием*» дать возможность не только определить полный или никакой доступ к методу классу, но также подключить временные условия доступа. Хойсман предлагает подключать в спецификацию историю объектов и вызовы методов. Также предлагается объединить описание различных объектов при отдельных стереотипах и совместных действиях объектов, т.к. предложенный вариант видимо страдает от необходимого уровня абстракции, в первую очередь, для параллельного запуска методов объектов.

### **Фреймворки.**

Для достижения цели проверки спецификаций целых программных комплексов, представляются иные формальные подходы (см. опр.1.12). Один из наиболее широко известных и популярных подходов, является подход Мейера «*о делении ответственности*» [175]. Мейер [172] предлагает, как и многие до него, методологию «*модулярный подход*», аналогично принципу Лискова и принципу скрытия по Парнасу. В алгоритмах связанных между собой, актёрам ответственности приписываются стереотипы, согласно которым, выполняются ранее определённые роли. Определение ролей в соответствующей *онтологии* участников и взаимосвязей позволит лучше охватить и понимать объектную структуру. Принцип Мейера можно вкратце охарактеризовать как:

Чем сильнее деление ответственности [каждого из единиц онтологии], тем более таким объектам, можно доверять.

Замысел заключается в том, что, чем меньше становится программа, тем сильнее редуцируется её сложность, если изначальную программу не менять (кроме сокращения). Например, метрика, в том числе по Мак-Кейбу, вычисляет показательное число, которое оценивает сложность графа управления данной программы. Чем сильнее разбита программа на маленькие легко-контролируемые и логически связанные между собой единицы, тем проще получается граф потока управлений. Метрики можно сравнивать, т.е. и сложности двух программ. Заметим, что объект, который кажется на первый взгляд простым, может в реальности иметь зависимости с любыми другими объектами, которые заранее неизвестны. Такие «*неожиданные зависимости*» часто на первый взгляд скрыты и их трудно обнаружить. Неожиданные связи уже являются признаком тому, что в отличие от данной спецификации, данный объект вероятно содержит ошибку или серьёзную не выявленную проблему. Неожиданные связи соединяют часто отдаленные объекты между собой, что для простых моделей

и алгоритмов маловероятно и приводят к ситуации, когда два объекта связаны между собой, хотя на самом деле не должны быть связаны.

Рилэ [227] предлагает вводить тесты, покрывающие спецификации для обнаружения отклонений объектов и взаимосвязей между объектами, так называемые «*коллаборации*». Проверяются взаимосвязи между объектными модулями с помощью спецификаций, например, «*модульной алгеброй*» [30] или «*компонентной алгеброй*» [93],[239],[259]. Эти подходы формализованы и описывают интерфейсы между классовым объектным и иными экземплярами. Описание интерфейсов, увы, ограничивается лишь входными и выходными параметрами методов. В [94] предлагается подход навигации по объектам, согласно данной спецификации. В [30], [93] и [17] предлагается добавлять последовательность вызовов методов в качестве утверждения, для избежания ошибок в связи с не правильным порядком вызовов методов. Это предложение похоже на подход Хойсманой. Далее, программы, часто страдают от того, что их необходимо специфицировать целиком. Это очень неудобно, когда речь идет о средних и больших программах. Фактически, все представленные подходы исключают указателей из языка программирования, т.к. любые утверждения о программе могут просто оказаться ложными. Порядок вызовов методов может оказаться практически значимым не только внутри одного объекта, но а также для целой сети объектов, например: исключение возможности вызова метода  $m_2$  первым объектом, до тех пор, пока не будет вызван метод  $m_1$  третьего объекта. Программисту важно помнить в какой момент *безопасно* вызывать метод, а когда требуются инициализации или иные вызовы и в каком порядке. Это нужно рассматривать как постоянную «*классическую*» проблему при изучении любого нового фреймворка. Вместо детальной и длинной спецификации с практической точки зрения, хорошо бы иметь спецификацию, которая статически обнаруживает соответствующие зависимости между объектами и порядком загрузки.

Язык объектных ограничений «*OCL*» [1] является расширением языка «*UML*». Он графический и текстовой де-факто стандарт по статическому и динамическому моделированию ПО. «*OCL*» позволяет добавлять формулы взаимосвязей, например, ограничить связь типов связанных между собой объектов. Формулы описывают в основном лямбда-термы второго порядка предикатной логики (см. опр.1.9), т.е. с атомными типами. Кроме ранее упомянутых ограничений в разделах, остаётся отсутствие возможности специфицировать указатели.

Сафонов [231] представляет и анализирует целый ряд примеров из области компиляции и связанные с ней проблемы. Цель анализа заключается в постановлении критерий *доверительных компиляторов*. Необходимо заметить, что автор выбирает компиляторы преднамеренно, т.к. они являются по сложности одними из наиболее сложными ПО. В этом можно убедиться, например, с помощью тщательного анализа кода, особенно переходы и ответвления, например используя метрики. Сложность проверки для данного компилятора заключается в том, как определить, что любая преобразованная программа верна и оптимальна? Для получения этого, Сафонов ссылается на метод Мейера о *делении ответственности*. В качестве контр-аргумента можно привести, например подход Леруа

из проекта «*CompCert*» [31], который также исследует корректность, но также частично быстродействие. Увы, подход не имеет, большого практического значения из-за слишком резких практических ограничений. — Поэтому, по Сафонову, можно выявить следующие критерии значимые на практике и которые решают уровень приемлемости:

- Верный и понятный диагноз ошибки, включая генерацию контр-примеров.
- Трансформация моделей, включая объекты, предпочитается переход между моделями из-за относительно малых затрат спецификации.
- Возможность расширять и модифицировать объектные модели вычисления.
- Использовать формальные описания, если применимость от этого не страдает. В частности, Сафонов считает *полноту покрытия* формальной модели и *локальность* спецификации важными критериями.
- Абстрактные типы данных описывают собственные инварианты объектов.
- Промежуточные представления должны быть достаточно гибкими и абстрагированными так, чтобы они могли быть использованы на различных этапах верификации.

## 1.3 Модели динамических куч

### 1.3.1 Преобразование в стек

В разделе 1.2 в связи с объектными вычислениями уже велась небольшая дискуссия о преобразовании в стек.

В [258],[257][274] расследуется функциональный подход, в котором все переменные перемещаются в стековое окно (в [52] приводится формальное доказательство корректности подхода Тофти и Тальпина [258]). Реализован подход Гросманов диалектом Си «*Cyclone*» при поддержке языка «*ML*» (также см. [109]). Регоин определяется как совокупность связанных элементов динамической памяти, которые, например, при удалении полностью могут утилизироваться в один раз. Стековое преобразование, в частности, должно следить за диапазоном видимости, чтобы исключить нечаянного уничтожения, согласно автоматическому выделению и утилизации стека. В [98] обсуждаются диапазоны видимости за пределами стекового окна и демонстрируется расширение в системе «*Cyclone*». При перемещении переменных, необходимо следить за интервалами годности, иначе полученные «псевдонимы» и «возможно-псевдоним» могут стать ложными. Функциональный подход исключает динамические списки как параметры, рекурсивные структуры данных и процедуры, которые возвращают функционал. Все типы функций должны быть определены при компиляции. Мейер [173],[174] считает, что кроме корректности программы, важным является быстрый сбор мусора (см. [153]). Поэтому он предлагает, по-возможности, целиком избавляться от динамической памяти и переоформить алгоритмы для работы только над стеком. Как уже было предложено в [258], вталкивание в стековое окно не занимает большие затраты при минимальных объёмах. Он требует, что к абстракции объектов необходимо уделять больше внимания и предлагает использование вспомогательных переменных для уменьшения спецификаций. Корректность присвоений в статье полностью не доказана, а только для полного класса данных, как утверждается в статье. На самом деле, присвоение может оказаться ложным, когда, например, допускаются изменения программного кода во время запуска программы. Доступ к содержимому в стеке может быть быстрее, чем в динамической памяти, но всегда требует дополнительные затраты на вталкивание и выталкивание в стек и из стека, которые, в зависимости обсуждаемого алгоритма, могут резко снизить общее быстроедействие так, что использование динамической памяти может быть быстрее [12].

### 1.3.2 Анализ образов

Главной целью анализа образов [232, 189, 199, 85, 275] является выявление инвариантов в кучах памяти при запуске программы, например, для обнаружения псевдонимов. Выявляются фигуры описывающие инвариантную и гибкую часть динамической памяти [275], например, для линейного списка [85] или двусвязного списка [60]. Граф зависимостей между образами всегда описывается полностью с помощью *трансферных функций*, таких как: пустой образ «пуст», присвоение полю

или указателю и выделение новой динамической памяти. [232] вводит основные соотношения между двумя указателями: «псевдонимы», «не псевдонимы» и «возможно псевдонимы».

[199] замечает, что [232] и [189] могут привести в зависимости от данной программы, к не точному, а следовательно, к не правильному выводу, если для «если-тогда»-команды в одном случае вычисляется «возможно псевдонимы», а в альтернативном случае «псевдонимы», тогда результатом вычисления послужит «псевдонимы», хотя правильный ответ «возможно псевдонимы».

Более того, [199] содержит подробное сравнение подходов [232] и [189]. [199] оценивает [189], как более точный метод и предлагает оптимизацию путей по графам образов с одинаковыми началами и псевдонимами, в качестве более эффективного описания, которое одновременно и короткое. Предложенная оптимизация имеет верхнюю сложность  $\Theta(n^2)$  и сокращает вычисление в среднем, примерно на 90%. Павлу предлагает симулировать более сложный анализ псевдонимов за пределами процедур, преобразуя, насколько это возможно, вызовы процедур и глобальные переменные в локально интра-процедуральные элементы через переименование. Подход в его работе был, на самом деле, уже ранее успешно применен в системе «GCC» для решения комплекса иных проблем. Павлу, так же как и я, считает, что контекст-независимые подходы в точном анализе псевдонимов, малоперспективны (ср. [113]), сравнив характеристики. Далее он предлагает оптимизацию отделения объединяющих вершин, которые образуют подграфы, от вершин, которые твердо не содержат псевдонимы.

[192] представляет собой среду для визуализации образов куч для быстрого анализа и обнаружения инвариантов, редко используемых связей между образами. Для навигации по графу используются операции «абстракция графа» и «конкретизация графа» с помощью чего, удаётся подграфы свёртывать и развёртывать. Визуализация трансферных функций, которая приводит к развёртыванию/-свёртыванию более одного подграфа, желает быть лучше, однако, ограничение принципиальное и следовательно улучшение не ожидается.

[54] представляет собой подход, основанный на распределении куч по образцам. Подход сравнивает предположительно подходящие начала правил по длине и выбирает наиболее длинное правило первым. Особенность подхода заключается в аппроксимации обеих сторон рассматриваемых правил для выбора принимаемых правил с помощью ограниченной абдукции.

### 1.3.3 Ротация указателей

[249] предлагает «ротацию указателей», которую можно считать безопасной, если: (1) содержание куч не меняется, (2) все элементы ротации годны до и после ротации (3) количество переменных не меняется. Преимуществом безопасной ротации можно считать отсутствие нужды сбора мусора, который определяется просто, но также эффективные операции над списками, как, например, копирование. Хотя ротация указателей в явной спецификации не нуждается, всё-таки минимальное изменение параметров может привести к трудно-прогнозируемому поведению программы, особенно



часто, если указатели неожиданно оказываются псевдонимами. [249] предлагает базисный объём ротаций, однако, на практике этого далеко не достаточно, в следствии чего, необходимо компоновать индивидуальные ротации из базисных. Далее, ротации очень чувствительны к минимальным изучениям вызовов. Например, поменять местами аргументы, на первый взгляд — безопасная операция, но она может привести к полной утилизации всех входных списков. Неожиданность также возникает часто с безобидными данными, например, когда один входной список пуст или ротации комбинируются.

Ротация может быть представлена как пермутация, где все её компоненты  $x_j$  являются указателями:

$$\begin{pmatrix} x_1 & x_2 & x_3 & \cdots & x_{n-1} & x_n \\ x_2 & x_3 & x_4 & \cdots & x_n & x_1 \end{pmatrix}$$

К примеру рассмотрим из рисунка 1.23 модифицированный пример ротации из [249].

```
var temp: T;
y:=NIL;
while x!=NIL do begin
    temp:=x^.next;
    x^.next:=y;
    y:=x;
    x:=temp;
end
y:=NIL;
⇔ while x!=NIL do
    rotate(x,x^.next,y);
```

Рисунок 1.23: Пример кода ротации указателей по Сузуки

Это соответствует линейному списку с тремя элементами следующей сложной трансформации, которая состоит из шагов (№1–№4), см. рисунок 1.24.

В зависимости от входных данных, можно выявить отдельные свойства, которые иногда тяжело обобщать, но иногда просто, как на примере `rotate(x,x,y)` является тождественным отображением, т.к. `tmp:=x; x:=y; y:=x; x:=tmp;`. Очевидно, даже маленькая модификация может привести к негодности свойств, а следовательно свойства симметрии могут быть нарушены, а следовательно доказуемые свойства больше неверны, например, `rotate(x,y,x^.next)`, т.к.  $x$  и  $x^.next$  связаны и пересечение между  $y$  и  $x^.next$  не полностью исключено априори.

### 1.3.4 Файловая система

Граф кучи отображается в файловую систему так, что вершинами являются файлы, а зависимости между ними — ярлыки (например с помощью команды `ln -s` под Линуксом). Папки могут быть использованы как хранитель объектного экземпляра. Содержимое вершины находится внутри файла.

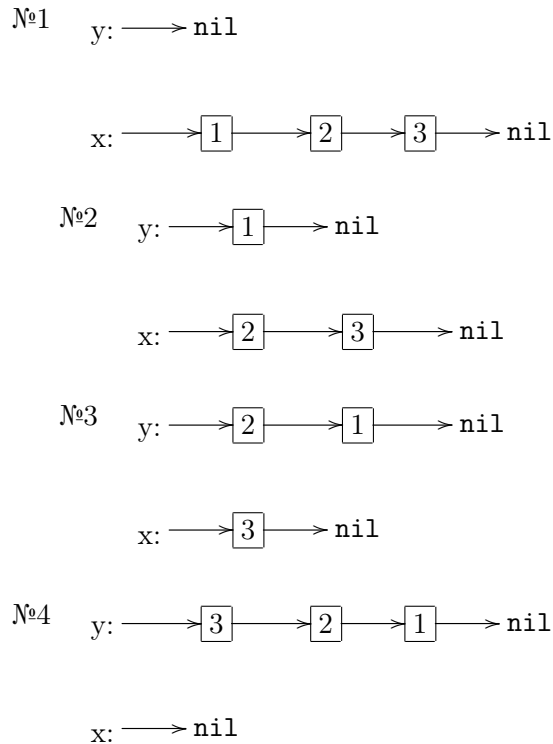


Рисунок 1.24: Динамическая память при запуске программы ротации указателей

Таким образом, файловая структура имитирует графовую кучу. Теперь можно описать схему файловой системы, не отслеживая символьные ссылки. Полученный перечень имеет структуру дерева и может быть записан в качестве слабоструктурированной единицы, например, в XML-документ. Под Линуксом все операционные средства преобразованы в виде файла. Операции над файлами теперь соответствуют операциям над кучами. Задача верификации теперь может быть задана как проверка данного состояния файловой системы. С другой стороны, операции над файловой системой можно проверить на корректность с помощью (например, XML-) схемы, что явно может упростить ошибки или дефекты с большими и многими файлами. Во избежание траты времени на сравнение многих и больших файлов и папок, можно использовать команду `sha1sum` или `md5sum`, т.к. писание и чтение на носителе часто ограничивается физическим барьером.

Для проведения операций [292] и описания [298] может быть использован декларативный язык, основанный на Прологе. В выделенном примере из [292] высчитываются из верхней папки ровно два файла **a** (например, файлы с указанным префиксом) и высчитывается содержимое, которое должно совпадать в обоих случаях.

```
template(element(top,_,[A,A]),[text(T)]):-
    A=element(a,_,_),transform(A//p#1,T).
```

Приближенный аналог на языке XSL-T является:

```

<xsl:template match="top[count( child::*)=2] and a[1] and a[2]">
  <xsl:text>
    <xsl:value-of select="//a//p"/>
  </xsl:text>
</xsl:template>

```

## 1.4 Побочные области

Основные определения псевдонимов, мусора и обсуждения находятся в разделе 1.3. Далее представлены побочные области, которые в принципе могут быть добавлены к предложенному фреймворку из раздела 4.5 и характеризуются как анализ указателей.

### 1.4.1 Анализ псевдонимов

Вайль [269] представляет наглядный обзор по тематике, несмотря на возраст статьи. По Вайлю анализ псевдонимов является побуждающим процессом приближения указателей, ссылающихся на одну и ту же ячейку памяти, которая может быть изменена из-за того, что создаёт множество псевдонимов с экспоненциальным ростом вариантов. Анализ псевдонимов за пределами процедуры гораздо тяжелее, потому, что необходимо анализировать все возможные вызовы и продолжения, в котором далее будут существовать переменные. Естественно, что в продолжении переменные так же могут меняться, и это является причиной трудного анализа. Чтобы упростить анализ, Вайль вводит ярусы псевдонимов, по которым оценивается сложность каждой программной команды.

[181] содержит полный обзор всех нынешних и прошлых анализов псевдонимов, в том числе свои собственные. Статья Мучника вместе со статьёй Вайля представляются самыми важными в области. Он делит анализы на зависимые от графа потока управлений и независимые, на «*псевдонимы*» и «*возможно псевдонимы*», а также на замкнутые подпрограммы и вызовы подпрограмм. Анализы псевдонимов по Мучнику можно уточнять, однако, классификация по сложности Лэнди [151] тоже ориентируется по диапазону видимости указателей.

В отличие от Мучника, Купер [70] предлагает, одним из первых, подход анализа псевдонимов независимый от графа потока управления. Главной мотивацией тому служит резкое упрощение алгоритмов, т.к. необходимо следить за указателями и операциями использования. Недостатком является неточность.

Стоит отметить, что система «GCC» [252] разрешает программисту языков Си устанавливать директиву для компиляции вручную, по которой возможно твёрдо исключать псевдонимы для определённой функции. Таким образом, генерируемый код становится более эффективным.

[118] является расширением подхода Мучника. С помощью битовых векторов, подход становится более эффективным. Общий подход улучшения анализа следует также искать в [137]. Более того, представленный подход анализа присвоений и использований может быть обобщён в подходе к

«SSA» [76],[245],[289],[238]. Наим [183] утверждает, что улучшение доступа можно достичь через хеширование подмножеств псевдонимов. Остаётся отметить, что подход Наима содержит предпосылки и намерения переписать проблему анализа псевдонимов как проблему «SSA». В общем следует отметить, что указатели не единственные переменные, которые трудно выявить при статическом анализе [242] за пределами процедур. Они являются одними из самых трудных проблем. Подход Сривастава полезен, независимо от всех остальных обсужденных работ, по причине, что глобальная оптимизация на уровне связывания кода, например, передвижение мало-эффективного кода или устранение ненужного кода часто не может полностью быть исключено во время предыдущих фаз компиляции.

[199] делит анализ псевдонимов на две методологии: подход с помощью «*унификации*» [246], который находит больше случаев «*псевдоним*» и на подход с помощью «*плавления*», который ради скорости слабее.

[150] и [214] задаются вопросом, почему сегодня анализ псевдонимов остаётся в общем не решённым. Ответ лежит в сложности выявления псевдонимов за пределами процедур и в ограниченности решаемости для случая «*псевдонимы*». В сжатых подграфах зависимостей наименование вершин остаётся общей проблемой в [232] и [189]. Далее, эти подходы имеют целый ряд ограничений, как например: доступ к указателям с индексом, разрешает лишь не переменные выражения; исключаются объектные указатели; запрещаются массивы с гибкой шириной; исключаются пересекаемые в памяти структуры данных (например, «*union*»-структуры на языке Си [243]).

Клинт [64] занимается верификацией программ, и его центральная проблема заключается в доказательстве сложности корректности псевдонимов в связи с со-процедурами. Работу Клинта можно считать одной из первых в этой области. Обобщённой областью Клинта можно считать верификацию с функционалами, в которой необходимо упомянуть работу [169] для указателей с логикой высшего порядка, и работу Поулсона [198], который пробует применить абстракцию процедур для принятия более эффективного правила при верификации с указателями.

Готсман [108] представляет подход приближения псевдонимов между процедурами с помощью модели ЛРП. Хотя статья представляет характеристики запуска и сравнения базисных оценок, для сравнения было бы интересно узнать о соотношении подходов Мучника и Купера.

#### 1.4.2 Сбор мусора

Джоунс и соавторы [127] представляют, более чем полный, обзор на тему *сбора мусора* и детально обсуждают актуальные подходы с большим вниманием на параллельные алгоритмы, однако, это не цель данной статьи. Помимо первого выпуска [127], второй выпуск одноимённой монографии не только сильно отредактирован и представляет целый ряд новых параллельных подходов, но практически одновременно является совершенно другой книгой. Стоит лишь упомянуть Уитингтона, [277] в качестве обзора на тему, который сфокусирован в основном на многопоточные реализации сбора мусора, также как и Долигез [88]. Блэкбёрн [40] приводит сравнения наиболее важных сборщиков

мусора, что также упомянуто в [127].

На мой взгляд, Аппель [12] прежде всего опасается на то, что на практике из-за архитектуры фон-Неймана и *программного интерфейса «ABI»* на кодовом уровне имеются ограничения с обращением к стеку, связи с копированием данных в и из стека. Аппель предполагает, что, в таком случае, выгоднее использовать динамическую память, но при условии, что либо вообще ничего не потребуется, либо меньше ресурсов, чем для стека (см. [258], [174]). Подход Аппеля заключается в многопоточной реализации с *«копирующим сборщиком мусора»* [127], который действует только, если было совершено изменение в данных, а зарезервированный превышает в семь раз объём свободных куч.

Несмотря на возраст [153], деление памяти на быстрый кэш и медленную, большую память принципиально остаётся в силе. Ларсон рассматривает *«уплотняющий сборщик мусора»* в зависимости от объёма освобождающей области ( $R$ ), объёма активных данных ( $A$ ) и объёма быстрой памяти ( $H$ ). Он постулирует две оптимальные стратегии для быстрогодействия и выделения/очистки динамической памяти: стратегия №1) Максимизировать  $R$ , если  $A \ll H$  не соблюдается. Стратегия №2) Приравнять  $R = H$ , если  $A \ll H$ .

Кроме упомянутых в [12] ограничений, сбор мусора ещё имеет ограничение в связи с адресацией. Если речь идет о данных «XOR»-связанных структурах, то мусор не может быть локализован классическим методом потому, что адреса принадлежащих объектов (например, поля объектов или ссылки на последующий элемент в списке) получаются не абсолютными, а относительными. То есть, для дважды связанного списка с помощью одного «*офсета*», можно выявить следующий и предыдущий элемент, если таков имеется, вместо двух отдельных указателей, которые содержали бы два абсолютных адреса.

[128] предлагает сбор мусора по возрасту выделяемых ячеек, т.е. в зависимости от того, как часто выделенный объект употребляется и как долго он остаётся. Если объект используется редко, то через несколько итераций он уже может оказаться в менее быстром сегменте памяти. Если объект используется часто, то он, наоборот, перемещается в более быстрый регион памяти — это касается не только сбора мусора, а также запуска программ в целом [261],[181],[260],[217]. Быстродействие в среднем приближено к оптимуму, исходя из большого практического опыта и множества экспериментов.

[119] даёт оценки к нынешним технологиям «SSD»-дисководов. Операции доступа к «HDD»-дискам похожи на динамические кучи. Писание превышает длительность чтения в 10 раз. Сбор мусора на «SSD»-дисках при «жадной» стратегии, становится наихудшей при задержке на 45%. Самая худшая стратегия по производительности — писание всё новых блоков. А лучшая стратегия — писать последовательные данные без сбора мусора, по отдельности.

Кальканё [53] замечает, что программы высокого абстрактного уровня не всегда имеют преимущества по сравнению с программами, которые манипулируют динамической памятью. Например, за-

мечается, что сбор мусора во многом неэффективен в функциональных языках программирования. Выходит, что маленькие программы «опасны» потому, что неверное использование с указателями может иметь побочные эффекты, но с другой стороны эффективны.

Подход Уэйт-Шора [234] классический и первозванный в данной области. Он характеризуется тем, что все элементы динамической памяти имеют счётчик активных указателей. Если, по каким бы то ни было причинам, ячейка вдруг более не жива, т.е. количество ссылающихся указателей становится ноль, тогда система управления (обычно ОС) активируется и принимает соответствующие шаги по утилизации ненужного объекта.

Ситуацию вокруг сбора мусора на данный момент лучше всего можно охарактеризовать: подходов имеется больше, чем достаточно, в общей сложности, как минимум 50 различных. Выигрыш через новооткрытие на сегодняшний день кажется мало вероятным и бесперспективным, т.к. на данный момент сборщики мусора в системах эксплуатации могут быть запрограммированы так, чтобы они не мешали запуску программы с помощью нитей.

### 1.4.3 Интроспекция кода

Интроспекция означает, что при запуске программы единицы модуля определяются, например, код вставляется, меняется или добавляется во время запуска. Очевидно, что интроспекция фундаментально усложняет статический анализ, ради проблемы приостановки, а значит, и верификацию программы. Принцип интроспекции основан на решимости, которая приводит к загрузке, либо известного, либо экстерного неизвестного кода. В любом случае должен существовать механизм в заимосвязи, который определяет коммуникацию между вызывающей и вызванной сторонами. Например, в Си++ интроспекция основана на «RTTI» [243], в Яве [96] каждый объект содержит ещё, кроме полей и адресов, идентификатор в качестве смежной записи, которая читается при запуске.

Формэн [99] определяет интроспекцию программы на Яве, как возможность вычитания и модификации данных и самой программы во время запуска. Это включает в себя загрузку кода, который может определиться даже только во время запуска. Следовательно, это потребует, кроме запуска, ещё компиляцию кода во время запуска. Запуск совершается объектами из определённого загрузочного контейнера. Манипуляция и доступ к классам, объектам и их компонентам осуществляется с помощью зарезервированных операторов.

Чеон [59] даёт краткий обзор текущих и предложенных вопросов интроспекции в области Ява. Вопросы частично ещё актуальны на данный момент. Так как запуск неизвестного кода представляет собой не только возможную опасность, но и сильно могут пострадать корректность и полнота. Печально было бы, если вдруг метод, именно для одного входного значения, вообще не работает, либо вычисляет неправильный результат. Для этого Чеон формулирует список предложений и важных обстоятельств, как например: (1) загрузка и избегание конфликтующих имен в контейнере, (2) при запуске доступ к объектам должен соблюдать типизацию, (3) спецификацию наследованных

объектов очень тяжело понять и отследить, если иерархия наследованности велика, (4) спецификации обычно ссылаются на объекты в стеке и куче, как в таком случае быть при интроспекции с окнами стека, которые просто разваливаются, т.к. объектная замкнутость может нарушаться при интроспекции?

Главными областями применения интроспекции являются загрузка приложений или методы веб-серверов, загрузка и доступ к динамическим библиотекам, например [282].

## 1.5 Существующие среды

[224], [27], [223] представляют собой хорошее введение в ЛРП, в том числе при поддержке программных средств. ЛРП является подструктурной логикой [218],[219],[89], которая избавляется от констант, например, булевых значений и использует символы для структурных замещений, в качестве констант. В ЛРП структурными правилами служат, согласно [218], правила *сужения*, *контракции* и *сопоставления*, а константами служат ячейки динамической памяти (см. главу 3). В правилах « $\rightarrow$ » заменяется на « $\star$ », которая разделяет две непересекаемые и различные друг от друга кучи, кроме того случая, когда уговаривается что-нибудь иное. Кучи в динамической памяти определяются индуктивно. Оператор « $\rightarrow$ », например в выражении « $a \rightarrow b$ », определяет соотношение между переменным символом на левой стороне и выражением (например объект) на правой. Когда мы рассматриваем примеры ЛРП, то подразумеваем, что эти конвенции вместе с правилом фрейма соблюдаются. *Правило фреймов* означает, что если вызов подпроцедуры не меняет части куч, а именно раму обозначенной  $F$ , то в антецеденте достаточно доказать, что тройка Хора без  $F$  верна. Рассмотрим к примеру правило фрейма над кучами, которые определяются позже:

$$(\text{FRAME}) \frac{\{P\}C\{Q\}}{\{P \star R\}C\{Q \star R\}}$$

Здесь  $P$  и  $Q$  являются пред- и постусловием, а  $R$  является кучевым фреймом, т.е. сложное утверждение, которое не зависит от выполнения  $C$ , а также, на чью верность  $C$  не имеет эффекта. Этот принцип является основой модуляризации и мы будем всегда исходить из того, что принцип в силе, кроме отдельно уговариваемых случаев. Если допускаются рекурсивные спецификации вместе с функционалами, то основные свойства фреймового правила могут нарушаться в связи с контекстом, который может меняться любой вызывающей инстанцией [36],[211]. Следовательно, рекурсивные спецификации, в таком случае, должны обобщаться.

[27] вводит вычисление на основе ЛРП с безграничной арифметикой в офсетах над указателями, с возрастающими массивами и рекурсивными процедурами. Например,  $p+x$ , где  $p$  указатель, а  $x$  переменное целое число, не решимо в общем. Оно представляет собой попытку определить области динамической памяти рекурсивно, с помощью замкнутого объёма встроенных правил. Бёрдайн и соавторы, лишь задают вопрос, не является ли типизация достаточной верификацией (см. набл.1.15)?

Из статьи например следует, что нерешаемость безграничного использования указателей, приводит к не точному определению момента сбора мусора, даже для самых безобидных выражений в качестве офсетов памяти и к не точному выбору правил для верификации, которые управляются жадной эвристикой.

Борна [44] предлагает модель очень похожую на ЛРП, которую он называет «*дальнее разделение*» и которая преобразует объекты данных в массив. Таким образом, любое объектное поле получается индивидуальным указателем с конвенцией для наименования и идентичности. Для общего определения куч он постулирует, что необходимо использовать предикаты первого порядка.

Главной заслугой Хурлина [121] в ЛРП, является нововведенный паттерн для верификации доступа к кучам с многими потоками. Он предлагает разблокировать функции, которые улучшают производительность за счёт частичной блокировки сложных ячеек. Все кучи не нуждаются в полной спецификации, которую можно сократить, используя анонимный оператор «*\_*».

Паркинсон [195],[194],[193] представляет объектно-ориентированное расширение подхода ЛРП [224] до Хурлина, используя Ява в качестве входного языка. Модульность и наследование моделируются с помощью «*передачи собственности над вызовами*» и «*семейством абстрактных предикатов*». Модель доступа Борна [44] применяется, т.к. свойство непрерывности наблюдается у правила фреймов для объектов. Он замечает, что зависимость между предикатами создает порядок вызовов предикатов. Из статьи можно выявить, что его предикаты разрешают определить всю динамическую память и стек, но они не разрешают, например, определить любые предикаты первого порядка. Предикаты утверждения, которые сильно отличаются синтаксисом и семантикой от входного языка, не ограничены в типах, однако, использование символов имеет целый ряд ограничений, которые связаны с несимвольной реализацией сопоставления переменных символов. Фактически, предикаты используются как локальные переменные в императивных языках программирования — это существенное ограничение. Для дальнейшего расследования, он предлагает: вызовы методов из родительских классов, статические поля, интроспекцию объектов, внутренние классы и кванторы над предикатами.

Система верификации «*Smallfoot*» [27] является первым верификатором на основе ЛРП. Она работает чисто экспериментально и имеет маленький объём встроенных предикатов для определения куч. Система сильно ограничена в представленных и предложенных к применению структур данных. Она очень часто выходит из строя из-за быстрой прототипизации и связанных с ней множеством ошибок. Часто наблюдается нетерминация всяки с неограниченной тактикой применения подходящих правил. Система «*SpaceInvader*» [54] является наследником «*Smallfoot*» и включает в себя простые элементы абдукторного вывода. «*jStar*» [86],[194] можно рассматривать, как объектно-ориентированное расширение «*Smallfoot*», которое уже поддерживает классные инварианты и ограниченные (*абстрактные*) предикаты. «*jStar*» преобразует все программные команды в форму «*JIMPLE*», которая очень близка к IR от «*GCC*», язык который называется «*GIMPLE*» [171] и с



помощью которого, проводится верификация поблочно-управляющим командам. Для индустриального применения, это решение является наиболее приемлемым. «*Verifast*» [124] работает с входным диалектом Си и разрешает произвольные определения абстрактных предикатов. При невозможности вывода, пользователю приходится вручную добавлять команды и подсказки к правилам для завершения доказательства. Программная среда Хурлина [121] очень похожа на [86].

«*Cyclone*» [109] является верификатором динамической памяти на основе ВР (см. раздел 1.3.1). «*SATlrE*» [199] верификатор, который работает на основе *анализа образов* [232] и реализован в Прологе. «*SATlrE*» отличается тем, что образцы и зависимости между ними вычисляются не каждый раз заново, а только меняющиеся пути. «*Ynot*» [187] является «*SMT*»-решателем на основе «*OCaML*».

Далее, имеются следующие подходы и программные среды: (1) «*KeY/VDM++*» [23], [271] способствуют применению в индустрии объектно-ориентированные спецификации и интеграции с языком моделирования «*UML*», которое не допускает доказательства динамической памяти, (2) отслежка ячеек памяти при запуске программы (динамический подход), например «*Valgrind*» [254] или «*ElectricFence*» [128], [91] [141], (3) программные среды интегрированные (статический анализ) в пакете компиляции «*LLVM*», как например, «*SAFECode*» [75], (4) преобразование программы и утверждений в вид слабоструктурированных данных [20] и возможной трансформацией ею [125], [87] (см. раздел 1.3). Раздел 1.1.3 содержит подробнее обзор по теме «*Абстрактной интерпретации*». Пункт 2 это динамический подход, который можно коротко описать на примере «*Valgrind*». Оно загружает данную программу с аргументами запуска в память и сопоставляет все доступы динамической памяти своими командами. Таким образом, каждый запуск записывает все неверные доступы к памяти, а также утечки памяти.

«*GCC*» [252] и «*LLVM*» [253] являются фреймворками компиляции. Обе содержат модули по статическому анализу [154],[75]. Хедкер [137] предлагает статический фреймворк, основанный на вычислении транзитивного замыкания достижимости в графах для анализа потока данных. Проект «*CompCert*» [43],[161] предлагает чисто академический фреймворк для анализа корректности трансформаций кода для процессорной архитектуры «*PowerPC*» при поддержке верификатора «*Coq*».

В качестве верификаторов общего назначения можно образцово привести уже упомянутую систему «*Coq*», но также имеется ряд иных верификаторов, например, «*PVS*», «*Proof General*», «*Isabelle*» и многие другие. Сравнение выбранных верификаторов можно найти в [92], а для верификаторов динамической памяти в [308]. Аппель [13] представляет перечень, какие проблемы за последние годы были хорошо решены и какое имеется сравнение. Статьи из немецкого популярного технического журнала «*iX*» [140], [139] посвящаются наиболее удобными динамическими программами, ранее представленными, для анализа проблем динамической памяти (см. главу 2).

Статический анализатор «*ESC Java*» (с англ. «*Extended Static Checking for Java*») [96] вводит модульную спецификацию в Ява (см. [105],[28]), но увы, указатели исключаются и модель памяти

исходит из того, что все элементы существуют в динамической памяти, а интроспекция не учитывается. Спецификация Ява-программ предлагается подходом «*Java-ML*» в [20].

Система «*Jahob*» представленная в [283] предлагает прототипную верификацию линейных списков для Ява-программ на основе функционалов на функциональном языке программирования близок к языку «*LISP*». Используется эвристика, которая выбирает правила с более жёсткими требованиями. Цель проекта заключается в поиске автоматизации или ускорения за счёт абстракции спецификации функциями высшего порядка (см. подход Поулсона [198]).

Система верификации объектно-ориентированных систем, как в этой главе уже было изложено, которые более близки к индустриально применимой верификации являются, например, также «*KeY*» [182]. Система «*Baby Modula 3*» [2],[56] может быть применима, но страдает от множества ограничений (см. раздел 1.2).

## 2 Проблемы динамической памяти

В этом разделе имеется короткое введение в типичные и актуальные проблемы в работе с динамической памятью. Затем дается короткая мотивация, почему изучение и исследование этих проблем актуальны с теоретической и практической точки зрения.

Под *динамической памятью* подразумевается (см. рисунок 2.1) та часть операционной памяти процесса, которая выделяется во время загрузки по запросу [163], [164]. Куча является неорганизованной частью памяти (см. рисунок 2.3), в отличие от организованной части памяти, т.е. стека. Организованность подразумевает автоматическое выделение и утилизацию локальных переменных по диапазону видимости (см. рисунок 2.2), с чёткой пропиской связи между элементами. В случае стека, при компиляции «ABI» вынуждает к строгому порядку помещения в текущее стековое окно локальных переменных. Например, задекларированные локальные переменные, при условном переходе видны в нём и могут перекрываться другими локальными переменными с таким же наименованием. При выполнении программы, локальные переменные не доступны за пределами блока видимости. Такое же наблюдается с процедурами и подпрограммами: локальные переменные, а также параметры по значению вталкиваются при запуске процедуры в видимое окно на стеке, т.е. актуальное. При выходе из процедуры, стековые переменные выталкиваются из стека, окно утилизируется (подробные реализации можно найти в [252]). На рисунке 2.2 имеется стековое окно с локальной переменной *o*, которая в данном случае, — целое число 1. Также имеется на рисунке массив, (локально, не обозначен) с указателями, которые ссылаются на *o* и на высший элемент. Все элементы *стекового окна* компактно помещены в стек, т.е. между ячейками обычно не имеются дыры. В отличие от этого, запись не обязательно должна быть компактной, точнее, если не упоминается ключевое слово «packed», то часто это и не происходит, благодаря генерации наиболее оптимального кода по быстродействию или размеру кода.

В отличие от этого, ячейки *кучи процесса* могут быть разбросаны как угодно. Видимость ячеек кучи не зависит от синтаксических блоков, а зависит исключительно от момента выделения динамической памяти до явной утилизации памяти. Если куча не утилизируется программным оператором, то по умолчанию все кучи утилизируются глобальным деструктором программы перед завершением процесса операционной системы. С этой целью кодовый сегмент `.dctor` в программах Си++ содержит все адреса деструкторов переменных объектных экземпляров, которые перед передачей контроля ОС вызываются. Стек и куча определяются операционной системой, они находятся

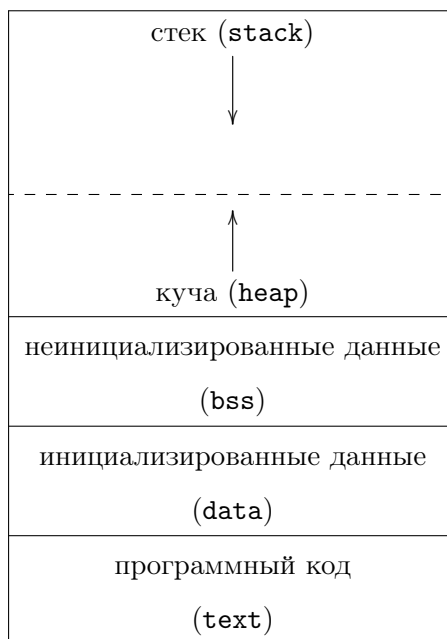


Рисунок 2.1: Типичное распределение процесса в оперативной памяти

в *виртуальной памяти*. Подробные механизмы памяти выделения и утилизации в данный момент нас пока не интересуют. Для стека и кучи нет фиксированного порога при запуске программы, а имеется плавающий порог (штрихованный, см. рисунок 2.1), который может, либо расти, т.е. стек увеличивается за счёт уменьшения кучи, либо наоборот, в зависимости от объёма употребления памяти. «bss» обозначает сегмент памяти, который содержит все глобальные (часто могут включаться иные, не локальные, в зависимости от конкретных реализаций при компиляции [243]) переменные, которым не было присвоено изначальное значение. Все остальные, не локальные и не динамические переменные выделяются в сегмент «data». Загружается программный код. В нём содержится объектный код с адресом точки запуска. Все относительные адреса заменяются абсолютными адресами. На практике программный код может и должен содержать различные фазы запуска программы, например, деструкторы.

Любая ячейка любого сегмента из рисунка 2.1 может быть адресована линейно. Это означает, что доступ к содержимому памяти можно получить по последовательному увеличивающемуся порядку. Любая ячейка памяти имеет последовательную по адресу  $+1$ , где  $1$  является символическим числом, которое представляет собой размер одного элемента соответствующего типа. Однако, согласно рисунку 2.1, имеется плавающая граница, которая практически никогда не достижима. Например, если целое число имеет тип `uint16_t`, то размер  $1$  станет  $2$ . Обратим внимание, что с помощью *эксклюзивного бинарного оператора «ИЛИ»* (XOR,  $\oplus$ ) можно моделировать два указателя в двусвязанном списке с помощью лишь одного «поля прыжка», если в качестве содержимого поля записывается  $a \text{ xor } b$ , где  $a$  является адресом начального поля, а  $b$  следующее поле, т.к. действует  $a \oplus (a \oplus b) \equiv b$ , а также  $(a \oplus (a \oplus b)) \oplus (a \oplus b) \equiv b \oplus (a \oplus b) \equiv a$  (см. [196], [240], [309]).

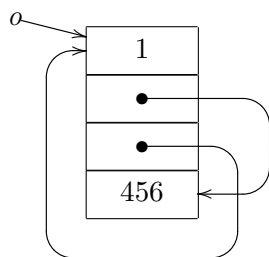


Рисунок 2.2: Пример стека

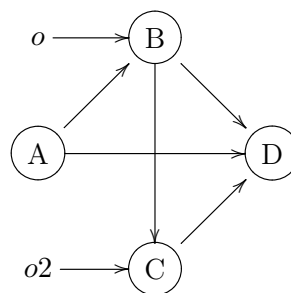


Рисунок 2.3: Пример кучи

Таким образом, можно сэкономить один указатель, хотя, утилизация ненужных ячеек памяти не может быть решена стандартно. Допустим, производится манипуляция двусвязного списка, тогда необходимо проверять достижимость с помощью сумм всех адресов списка.

Памятные модели между языками Си(++) [252],[253] и Ява [128] сильно отличаются, даже между Си++ и Си, если даже на первый взгляд это не очевидно. Так например, у языков программирования, которые совместимы с ISO Си(++) сбор мусора ненужных элементов часто не проводится. С одной стороны, это экономит дополнительные расходы, с другой стороны, это перекладывает большую ответственность на программиста. В языке Ява, сбор мусора очень часто приводит к дополнительным затратам, к счастью, сбор мусора работает часто в параллельной нити внутри Явы в виртуальной машине, что способствует к избеганию больших дополнительных расходов. Ява, как компромиссное решение, использует подход «сбора мусора по поколениям», который можно считать, приближенным к субоптимальным решениям со стороны практических порогов и объёмов мусора. В Яве все переменные и данные хранятся в кучах. Диалекты Си допускают *слабую типизацию* [243]. Не совместимые типы при различной интерпретации битов могут быть преобразованы, например, байт в вещественное число с помощью Си оператора `union`.

## 2.1 Мотивация

В техническом докладе [176] на протяжении десятилетий анализируются совершённые и отслеженные ошибки при разработке открытых и коммерческих программных систем обеспечения. Результаты [176] практически из года в год без изменения подтверждаются многочисленными новыми исследованиями, как например [177],[113],[5]. В [176] демонстрируется, что в приложениях для коммерческих дистрибутивов операционной системы «UNIX», содержатся около 23% ошибочного кода, а код открытых проектов содержит лишь около 7% на платформах «GNU». Авторы осторожны и оптимистичны. По Миллеру наиболее часто встречаются следующие ошибки:

1. Ошибки с указателями и полями. Авторы обращают внимание, что для коммерческих приложений, ошибки либо вообще не обнаруживаются во время разработки программ и начального

тестирования из-за недостаточного покрытия тестов, либо обнаруживается лишь при *портинг* продукта на другую платформу. Авторы более всего обеспокоены этим видом, т.к. их трудно обнаружить и исправить так, чтобы другие модули не пострадали. Чаще всего замечаются следующие трудности: неверный доступ к памяти, неинициализированные ячейки данных или утечки памяти, которые рано или поздно становятся не доступными.

2. Неверный доступ к массивам с преодолением доступных границ, либо в связи с изменением структуры данных, хотя указатель остаётся без изменений.
3. Недостаточная проверка работы с файлами. Часто при чтении, достижение конца файла вообще не проверяется.

Верифицировать динамическую память тяжело потому, что описывается состояние куч, а данная программа с описанием корреспондирует [126] «*неявным*» образом. Если интерпретировать кучу, как граф (см. следующие главы), а программа — это последовательность инструкции построения того графа, то описание одной и той же кучи может быть сделано разными путями. При проверке, обе стороны должны корреспондировать.

Хинд [113] задаётся вопросом, почему анализ псевдонимов до сих пор не решён. Он призывает заняться исследованием корректности и быстродействия (см. рисунок 1.1), т.к. *не выявленные псевдонимы* означают возможную деградацию откомпилированного кода. Если во время компиляции известно, что некоторая пара локальных переменных обязательно ссылается друг на друга, то при *выделении регистров* [181],[135],[237],[238], [226], можно гарантировано использовать один регистр, тогда отпадает необходимость синхронизировать два или более регистров (см. [245], [154], [76], [181]).

Хинд [151] убеждён в том, что для эффективного выявления псевдонимов нужно анализировать входную программу. Лэнди [151] вводит классификацию анализа псевдонимов. Анализы являются NP-твёрдыми. Они делятся на: (1) псевдонимы, которые «*должны ссылаться*» или «*могут ссылаться*» и на (2) псевдонимы внутри процедуры или за её пределами. Анализ внутри процедуры считается эффективным [114]. Анализ за пределами процедуры считается неэффективным и нуждается в улучшении. Хинд [113] предлагает использовать маленькие отрывки области видимости с локальными переменными при использовании эвристического подхода для решения вопроса псевдонимов.

Ху [119] представляет статью, которая посвящена техническим границам циклов писания и чтению актуальных флэш-памятей. Флэш-память как постоянный накопитель, здесь упоминается потому, что виртуальная память ОС при необходимости использует её и это быстрее чем использование винчестерского диска. Писание в память в среднем занимает приблизительно в десять раз больше времени, чем чтение, а блочная запись имеет наибольший эффект. Результаты Ху можно растолковывать так: сбор мусора не нужен вообще, в связи с продолжительностью жизни флэш-устройства,

т.к. сбор мусора сильно задерживает процессы писания и чтения остальных процессов. Кроме *постоянного накопителя*, флэш-память также может быть использована как виртуальная память во встроенных системах. Оно может критически повлиять на быстродействие целого приложения.

Бэссей рекомендует из-за неточности сред верификации для решения различных проблем с динамической памятью: (1) избавляться от программ, которые при запуске меняют программный код (2) соблюдать лозунг: *«обыкновенные ошибки должны быть найдены обыкновенно»* (3) учесть, что проведение анализов программы не обязательно приводит к улучшению качества программы. Отметим, что пункт (1) лишь меняет момент определения программного кода. С одной стороны — имеется увеличение гибкости, с другой стороны — имеется возможность ограничения семантического анализа, например, с проверкой типов, но также дополнительные расходы на динамические проверки и генерации кода во время запроса. Естественно, верификация не в состоянии угадывать поведение статически из-за проблемы приостановки. Из-за упомянутых недостатков (см. [221],[39],[235],[117]) динамически меняющегося программного кода далее динамические изменения не рассматриваются. В связи с ошибочным обращением к динамической памяти подразумеваются:

- доступ к недоступной памяти
- доступ к неинициализированной памяти
- нехватка динамической памяти при востребовании
- снижение быстродействия
- утечка памяти (см. далее).

Далее все эти последствия рассматриваются. Ошибки могут привести к самым непредсказуемым ситуациям, в худшем случае вплоть до неверных дальнейших ответов программы, а к приостановке в лучшем случае. Приостановку можно без преувеличения считать *«наилучшим»* вариантом, т.к. выход в определённой точке программы предпочтительно к неопределённому выходу. Факт приостановки программы говорит о неисправности, а далее коррумпированное состояние не является предпочтительным в различных смыслах корректности и целостности.

Бесси [33] приводит итоги анализа успешных верификаторов в общем, и объясняет актуальные причины и принципы. Наиболее важными пунктами Бесси считает:

- Верификация теорем всегда является точной наукой: эвристики могут применяться, но в конце интересует — следует ли предполагаемый результат из аксиом и правил или нет? Поэтому, давать квантифицированный результат трудно. Малое количество обнаруженных ошибок может означать, что верификатор плохой или хуже других. Бесси обращает внимание, что эвристика *«бери самое крупное правило первым»* на практике даёт хорошие результаты. Рекомендуется выбирать представительные примеры из [282], как например операционные системы, либо программное обеспечение в публичном доступе.

- Нетривиальные доказательства должны быть стандартизированы насколько это возможно. Если имеется возможность, то посторонние подпроцессы должны независимо от правил доказательств приводить к состоянию вместе с программным представлением, в наиболее нормализованный и упрощенный вид, с которым можно продолжать верификацию. Конкретные специфические правила должны быть логично отделены от остальных структурных правил. Общим лейтмотивом должен послужить: *«простые ошибки необходимо обнаруживать простым способом»*. Если лейтмотив не соблюдается, то можно считать верификацию мало полезной.
- По возможности как можно чаще и раньше исключать *«дополнения»* входного языка, которые можно исключать без большой потери выразимости, как например, динамическое обновление кода. Специфицировать всё подряд или то, что не связано со спецификацией (см. опр.1.3) или верифицируется с большими затратами в ущерб читаемости и при необоснованной потере ключевых свойств правил данного вычисления, не доступно. Всё должно быть простое и проверяемое, иначе доказательство может быстро оказаться неприемлемым, ради исключения редких случаев.

Как было ранее упомянуто, программирование с динамической памятью может быть накладным со стороны быстродействия [153]. Бывают случаи, когда использование куч быстрее [12], чем стека, в зависимости, как активно и эффективно работает сбор мусора. Нельзя говорить обобщёно (см. главу 1) нужно разбираться индивидуально, чтобы принять решение, какой алгоритм лучше. Эффективность куч также в значительной степени зависит от свободных/занятых списков куч, которые контролируются операционной системой. Торможение из-за стека происходит в связи с генерацией ненужных инструкций вталкивания и выталкивания. Проблема увеличивается с большими объектами, которые не помещаются в регистры. Каждый процесс копирования структуры данных, будь-то слово процессора, либо комплексный тип, является потенциально лишним, как только речь идет о ссылках. Если значение передаётся, тогда любые дубликаты являются лишними. Увы, устранение по этому принципу не всегда наблюдается в «*GCC*» [252] или «*LLVM*» [253] из-за сильно консервативного подхода. Замечаем, что часто любой алгоритм, который реализован с помощью куч, можно записать при использовании стека последовательно, как было частично предложено в [173],[174]. Адреса любого объекта куч можно свободно передвигать, присваивая зафиксированный адрес при условии, что каждый элемент стека можно однозначно адресовать. Для этого потребуется дополнительная конвенция идентификации последовательных типов (т.е. различия размеров) элементов на стеке. Важно соблюдать максимальный порог вталкиваемых объектов в стек, который увеличивается при уменьшении кучи (см. рисунок 2.1). Элементы массивов, т.е. кучевые объекты одинарного типа, можно вталкивать, просто копируя элементы последовательно. Важно, сохранять на стеке информацию о том, что следует массиву. Таким образом, единственными открытыми вопросами



остаются: (1) Действительно ли так нужно поступать, ради читаемости, эффективности алгоритма и т.д.? (2) Всегда ли применим такой вариант или нет? Первый вопрос довольно спорный, сравним например, элегантность деревьев в кучах (см. [196]). Второй вопрос не всегда решим, в частности, когда лимиты устанавливаются только при запуске программ, прочитав например информацию из файла. Конечно, можно попытаться установить некоторый максимальный технический массив для покрытия большинства случаев, однако, таков алгоритм далеко не эффективен, а тем более неполон. Всё, что можно заранее вычислить и статическим образом установить, это очень полезно для вычисления только со стеком. Увы, часто бывают ситуации, когда это не приемлемо по техническим или другим причинам. Нельзя не заметить, что стек и куча конечны.

Далее рассмотрим некоторые конкретные проблемы динамической памяти на образном диалекте Си.

## 2.2 Проблемы в связи с корректностью

**Пример 1— Утечка динамической памяти.** Утечка памяти является одной из проблем, которая встречается очень часто. При утечке выделяется объект в динамической области памяти, которая, в конце концов, не освобождается. Часто это не приводит к краху загруженной системы, однако, нельзя сказать, что это удовлетворительно. Программа раньше выйдет из строя без предупреждений, либо в любой неопределённый момент. Это может произойти тогда, когда ОС вдруг обнаруживает нехватку доступной памяти, либо ОС не может разрешить доступ к памяти из-за специфических правил ОС по безопасности, либо просто в ОС не осталось достаточно иных ресурсов. Типичный сценарий такой, что программа работает пару минут или даже три недели подряд без событий, а затем, либо из-за внешнего события, либо сообщения, программа неожиданно выходит из строя или просто завершается без указания ошибки. Часто отслеживание, если оно возможно, может не привести к самой ошибке, т.к. выделение памяти (если обнаружить) может являться только *симптомом*, но не настоящей причиной ошибки. Выявить настоящую ошибку крайне тяжело. Ошибкой программы из рисунка 2.4

```
MyClass object1 = new MyClass();
...
object1 = new MyClass();
...
(конец программы)
```

Рисунок 2.4: Пример программы инстанциации объектного экземпляра

является тот факт, что содержимое от `object1` не утилизируется после повторного присвоения. Если предположить согласно ISO Си++ [243], что ОС при завершении программы освободит все

выделенные ячейки, то уже до завершения программа может выйти из строя из-за нехватки памяти. Проследить такой тип ошибок будет крайне тяжело или практически не возможно, в частности потому, что при каждом запуске пороги ответственные за провал могут кардинально отличаться. Проблема заключается в анализе «опасных» мест в программе, которые могут не освобождать ранее зарезервированные ячейки.

**Пример 2 – Неверный доступ к памяти.** Неверный доступ к динамической памяти является ошибкой, которая встречается относительно часто. Причиной служит тот факт, что некоторый объект не инициализирован, либо имеет не правильное значение. Отметим, что ОС по-разному относится к объектам, которые находятся в «bss». Например, ОС «Windows» часто не инициализирует локальные переменные. Это приводит к ячейкам с неопределённым значением, тогда продолжение программы не определено, т.к. вычисляются альтернативные пути программы. Это может уязвить процессы. Причиной является неправильное или неприсвоенное значение переменных в рассматриваемом алгоритме (см. рисунок 2.5).

```
...
// object1.ref равно NULL
// (либо не присваивалось, либо NULL заранее)
...
value = (object1.ref).attribute1;
```

Рисунок 2.5: Пример неверного присвоения объектной ссылки

Анализ ошибки должен сосредоточиться на все присвоения с момента первой декларации переменной `object1`, включая все присвоения, если значение было передано. Из всех присвоений необходимо выявить то присвоение, которое «неправильное» согласно формальной/неформальной спецификации данного алгоритма. Неверный доступ к памяти можно считать как попытку доступа к объектам, которые были нечаянно утилизированы. Неверным доступом можно считать множество ошибок связанных с неправильной разрядностью или шириной процессорного слова в связи с неправильным применением преобразования типизации указателей. Например, в связи с принудительной конверсией типов между `(int*)` или `void*`, либо `(char*)` и `(int8_t*)`, которые в зависимости от компилируемой платформы могут различаться. Также в зависимости от платформы:

```
sizeof(struct(int a, char b)), sizeof(int)+sizeof(char)
```

не должны быть идентичны при типизированном доступе к адресу памяти. Размеры структур и её разрядность могут тоже отличаться. Если предположить, что *a* занимает 2 байта, а *b* занимает 1 байт (что на разных платформах может быть не так), то содержимое, может быть одним из вариантов битовых масок из рисунка 2.6.

$a_0a_1a_2a_3$ $a_4a_5a_6a_7$	$a_8a_9a_{10}a_{11}$ $a_{12}a_{13}a_{14}a_{15}$	0 0 0 0   0 0 0 0	$b_0b_1b_2b_3$ $b_4b_5b_6b_7$
$a_0a_1a_2a_3$ $a_4a_5a_6a_7$	$a_8a_9a_{10}a_{11}$ $a_{12}a_{13}a_{14}a_{15}$	$b_0b_1b_2b_3$ $b_4b_5b_6b_7$	0 0 0 0   0 0 0 0

0 0 0 0   0 0 0 0	$b_0b_1b_2b_3$ $b_4b_5b_6b_7$	$a_0a_1a_2a_3$ $a_4a_5a_6a_7$	$a_8a_9a_{10}a_{11}$ $a_{12}a_{13}a_{14}a_{15}$
$b_0b_1b_2b_3$ $b_4b_5b_6b_7$	0 0 0 0   0 0 0 0	$a_0a_1a_2a_3$ $a_4a_5a_6a_7$	$a_8a_9a_{10}a_{11}$ $a_{12}a_{13}a_{14}a_{15}$

$a_{15}a_{14}a_{13}a_{12}$ $a_{11}a_{10}a_9a_8$	$a_7a_6a_5a_4$ $a_3a_2a_1a_0$	0 0 0 0   0 0 0 0	$b_7b_6b_5b_4$ $b_3b_2b_1b_0$
$a_{15}a_{14}a_{13}a_{12}$ $a_{11}a_{10}a_9a_8$	$a_7a_6a_5a_4$ $a_3a_2a_1a_0$	$b_7b_6b_5b_4$ $b_3b_2b_1b_0$	0 0 0 0   0 0 0 0

Рисунок 2.6: Примеры распределения битовых масок

Поэтому, предпринимать какое-либо утверждение, зависимо от платформы, может быть даже ошибочным, если например речь идёт, об архитектуре «*Intel*», «*PowerPC*» 64-битных, «*ARM*» 32-битных или иных процессорах.

Если, по какой-то причине не инициализируется некоторое поле, то безобидная программа из рисунка 2.7 может не завершиться или завершиться не правильно, а также может выдать неверную информацию – любое из побочных эффектов является недопустимым.

```

object1.next = object;

...

root=object1;

while (root.next!=NULL){
    printf("%d", object.data);
    root=root.next;
}

```

Рисунок 2.7: Пример нетерминации кода

### Пример 3 – Висячие указатели и псевдонимы.

Висячие указатели (см. [8]) получаются, когда несколько указателей ссылаются на один объект и операции над другими указателями, возможные псевдонимы, приводят к тому, что хотя бы один указатель «*нечаянно*» ссылается по ошибке на пустое место или операции над одним указателем меняют связанную кучу. Хотя феномен интуитивно понятен, но на практике определение псевдонимов может оказаться довольно трудным и неточным. Необходимо анализировать не только одну процедуру с указателями параметров, но все возможные вызовы — т.е. одним вызовом множество содержимых указателей может поменяться, а в других случаях может ничего не поменяться. Содержавшие объекты висячих указателей подлежат утилизации, т.к. по определению они являются

мусором. Указатели, ставшие висячими, могут стать невисячими в ходе запуска программы, а также наоборот. Однако, содержимые, ставшие мусором, навсегда утеряны. Феномен висячих указателей можно достичь именно таким способом, но также при использовании различных интерпретаций данных ячеек, например, предусмотренные структурой объединения с помощью «union».

#### **Пример 4 – Побочные эффекты.**

Вместо ненужного копирования структур данных, часто можно при вызовах процедур лучше использовать ссылки. Однако, этот подход содержит опасность, что нечаянно могут пострадать посторонние данные и переменные. Эта проблема является обобщением примеров №2 и №3: указатели не меняются в ходе запуска, но содержимое неожиданно меняется. Далее можно обобщить: модификация неожиданно меняет другие переменные. Модификация одной кучи не должна отражаться на другую.

## **2.3 Проблемы в связи с полнотой**

#### **Пример 5 – Проблемы в связи с выразимостью.**

Несколько проблем с выразимостью уже были представлены в этой главе. Основные проблемы выразимости заключаются в: (1) можно ли все допустимые кучи специфицировать при условии, что все корректные кучи действительно синтаксически верны, исходя из данного набора правил? (2) Выводится ли, что все выбранные неверные кучи верифицируют как неверные, согласно правилам? (3) Какими можно использовать предикаты? (4) Каким условиям и свойствам должны придерживаться предикаты? (5) Каковы взаимосвязи между кучами и насколько адекватно это отражается в формулах? (6) Какие ограничения имеются в связи с использованием символами в описаниях куч? (7) Как лучше описать множество и отдельную кучу? (8) Как можно выразить зависимость между псевдонимами? (9) Имеются ли многозначимые кучи или их описания, если да, то почему и можно ли их эффективно исключать? (10) Какой уровень абстракции нужно вводить для удобного описания куч и для решения задачи верификации? (11) Имеется ли возможность абстрагировать настолько, чтобы, интуитивно стало ясно, о чём идёт речь, и пользователь, без особого труда, имел бы возможность лучше «понять» спецификацию? (12) Можно ли формулы для описания куч преобразовать так, чтобы не было необходимости специфицировать повторно? (13) Можно ли, если потребуется дополнительное преобразование, использовать «что-то более знакомое» программисту для спецификации и верификации чем искусственно определённые и ограниченные подвыражения логики предикатов (например, первого порядка), которые вычисляются и поддерживаются не полностью и часто не интуитивно?

#### **Пример 6 – Проблемы с полными представлениями.**

Согласно рисунку 1.1 в статье Сузуци [249] были предложены операции над указателями, которые можно считать «безопасными». С использованием нужно быть крайне аккуратно потому, что *ротация* одной структуры данных по часовой может очень быстро привести к уничтожению или фальсификации куч. Кроме этой проблемы корректности, часто неявные условия не очевидны. Подход Сузуци, а также другие подходы страдают практически всегда от неполного набора правил и не полностью описанных куч. Часто имеется ситуация: дан набор 25 правил. Вопрос: (1) достаточно ли этих 25 правил или требуется ещё добавлять или даже необходимо удалять правила? (2) Как быть с «*правилами дубликатами*»? Состояния куч связаны с программными операторами. То есть, необходимо описать целые подмножества куч и их сравнивать. (3) Имеет ли смысл ограничить выразимость куч так, чтобы приостановка была решимой с приемлемыми ограничениями? (4) Имеется ли возможность только часть кучи специфицировать и доказывать? (5) Можно ли предложить или использовать простую модель памяти так, чтобы доказательство данной простой структуры данных была простой, как например, реверс линейного списка (см. [223])?

Желательно, чтобы для верификации не было необходимости постоянно все методы полностью специфицировать. Выработка корректной и полной спецификации на практике означает большие затраты рабочего времени инженера, в чем часто нет необходимости. На практике часто специфицировать достаточно лишь некоторые процедуры или объектные классы. Для этого необходимо отдельные фрагменты программы оставлять не специфицируемыми. Практическим требованием для инженера является возможность добавления вспомогательных утверждений в те места программы, где инженер желает подробнее проанализировать некоторую нестыковку со спецификацией для локализации и выявления ошибки (см. алгоритм на рисунке 1). Разработчик также может быть заинтересован в добавлении проверок в произвольных местах программы дополнительно к пред- и постусловиям.

### **Пример 7 – Проблемы в связи со степенью автоматизации.**

Согласно лестнице качества из рисунка 1.1, эти проблемы входят во вторую категорию. Главные проблемы автоматизации в ранее упомянутых разделах связаны с необходимостью определять аксиомы и правило динамической памяти от общих логических и иных правил, которые не связаны с преобразованием элементов динамической памяти. Если установить формальную теорию, основанную на равенствах и неравенствах куч, и эту теорию записать в отдельный набор правил, то набор уменьшается и верификация упрощается. Такая попытка значительно уменьшила бы численность и сложность данных правил. Необходимо улучшить сравнение спецификации с данным состоянием куч, которое к большому сожалению проводится в существующих подходах практически вручную (см. далее разделы и главы). Когда данную теорему нужно использовать, а когда сопоставить с нужными символами — трудно предсказать. Проблема также существует при преобразовании из одного состояния кучи в другое. Локальное оптимальное решение доказательства может всё равно

привести к полной нерешимости. При преобразовании куч с помощью дедуктивного метода, также стоит задуматься об улучшении сходимости доказательства, подключив например абдукцию. Основным теоретическим ограничением может выступать выразимость формул. Термы могут быть, либо не полностью определены во время статического анализа, либо они нерешимы в принципе. Ограничение офсетов в арифметических выражениях приводит с одной стороны к ограничению выразимости, с другой стороны к повышению уровня автоматизации. Возникает практический вопрос, насколько полезны определения алгебры куч? Насколько достаточна строгая типизация во входном языке программирования?

## 2.4 Проблемы в связи с оптимальностью

### Пример 8 – Проблемы в связи с быстродействием.

Проблемы быстродействия конкурируют напрямую с корректностью (см. рисунок 1.1). Выявление утверждений *«указатели обязательно ссылаются»* или *«обязательно не ссылаются»* важнее, чем *«указатели могут ссылаться»*, но их одновременно труднее выявить. Первое и второе утверждения имеют более сильный эффект на генерацию кода. Чем больше сокращается время запуска соответствующего кода, тем эффективнее он, т.к. отсутствие необходимости сохранения процессорных регистров в стек, означает ускорение. Увы, анализ зависимостей с указателями данных сложнее, чем с локальными переменными потому, что содержимое указателя  $p$  может меняться не только там, где имеются присвоения к  $p$ , но теоретически в любом другом программном операторе.

Анализ псевдонимов является трудной частью, т.к. необходимо отслеживать все использования и вызовы процедур, что может привести к самым различным результатам. Часто, в фреймворках с указателями наблюдается, либо наиболее обобщённые утверждения, которые могут оказаться полностью без эффекта, либо утверждения вообще не рассматриваются.

Также важно заметить, что если структура данных используется только в одном месте и дубликаты отсутствуют, то требуется меньше расходов для непосредственной манипуляции. В реализациях наблюдаются в основном два подхода: либо вручную передаются простые и объектные переменные (но возможно с вспомогательными замечаниями, например, в Си с помощью ключевого слова `register` [252]), либо в исключительных случаях проводится анализ псевдонимов (в основном исключительно внутри процедур [253]). Этот случай не исключается в языках Си [252], используя ключевое слово `const`. В качестве мотивирующего алгоритма, например, реверса списка с изменением существующей структуры [223],[196], изначальный список исчезает, но это в зависимости от контекста может вполне устраивать. Алгоритм Рейнольдса только один раз проходит через линейный список (без явных и обратных указателей). Таким образом, отпадает необходимость копировать список. Все операции производятся по данному списку, что сильно ускоряет. Если бы знать заранее, что структура данных будет меняться и оригинал будет не нужен, почему бы и не забросить старый

список и таким образом резко ускорить алгоритм? Данные компиляторы [252],[253] пока не в состоянии проводить такой анализ, по крайней мере, не детально. Далее, можно ли поменять «ABI» при компиляции так, чтобы не используемые объекты удалялись из стека вызывающей стороны [243],[252] безусловно, и уже существующие объекты в динамической памяти были бы использованы непосредственно, если объекты строго не меняются при вызове? Надо отметить, что модель указателей в [240] не стандартная, затраты для линейных списков сильно сокращаются, сбор мусора изменен до неузнаваемости.

В частных случаях, когда заранее известно число итераций циклов статическим анализом (см. [12]), то можно оптимизировать место расположения ячеек памяти.

Проблемы оптимальности сбора мусора, начиная с *алгоритма Уэйт-Шора* [234] доныне, можно считать, более чем достаточно исследованы [127]. Микрооперации в связи с динамической памятью производятся операционной системой, которая следует за свободными ресурсами, в том числе, куча и стек.

### **Пример 9 – Проблемы в связи с целостностью и безопасностью программы.**

В качестве оптимизации (см. рисунок 1.1) также рассматриваются проблемы в связи с анализируемой программой, где предполагаются корректность и полнота.

Аналогично к *переполнению стека* [138], когда стек наполняется нежелательными данными, с целью передвижения актуального указателя за пределы актуального стекового окна при вызове или возврате с процедуры — имеется такая же попытка вталкивания обратной метки, например, при сборе мусора в куче [130],[8]. Очевидно, что в отличие от стека, куча не содержит адреса программного кода, следовательно, атака включения вредного кода прямым образом не сможет сработать априори. «*Переполнение куч*» может привести к потенциальной уязвимости процесса, либо целой системы.

Особенно критично стоит вопрос о безопасности с интерфейсами дальних серверов и служб, а также со спецификациями, когда вызывается некоторый системный доступ к драйверу [71]. Так как, драйвера могут запускаться несколькими инстанциями одновременно, то неисправность в связи с ошибкой в динамической памяти является особенно критической. В худшем случае нестабильность может привести к краху ядра ОС, как это имело место быть в случае с монолитной архитектурой ОС «*GNU Линукс*» с одним ядром.

### 3 Выразимость формул куч

Предположим, что имеется некоторая императивная программа, где граф потока управления [137] выглядит, как представлено на рисунке 3.1. Пример из [76] послужит нам образцом выявленных разниц между автоматически и динамически выделенными переменными. Особенность автоматически выделенных переменных заключается в том, что они выделяются и уничтожаются автоматически открытием и закрытием стекового окна. Стековое окно содержит все локальные переменные и параметры, которые поступают во внутрь и выходят наружу. Нельзя это путать с «*fan-in*» и «*fan-out*».

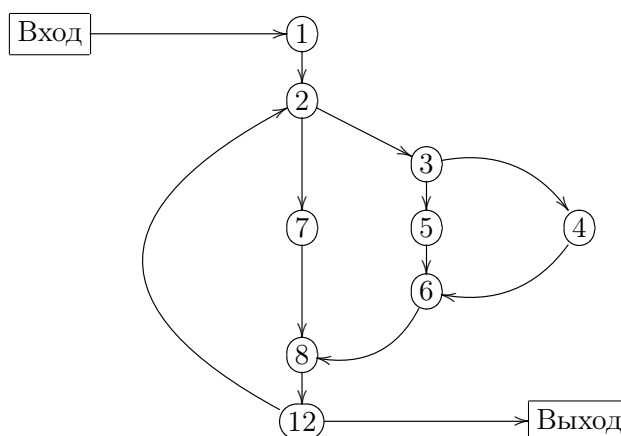


Рисунок 3.1: Пример графа потока данных.

Согласно рисунку 3.1 определяются *интервалы видимости*. Интервал видимости всей процедуры, например  $[Вход, Выход]$  означает, что передаваемые переменные видны на всех вершинах между начальным блоком «*Вход*» и конечным блоком «*Выход*». Блок определяется как объединение последовательных неразветвляющихся программных операторов. Далее, во избежание коллизий между различными переопределениями, предпочитается форма блоков в виде «*SSA*» [76],[237],[245]. Разветвлением может быть любой условный или безусловный переход. Программные операторы безусловного перехода исключаются.

Допустим, в некоторых блоках определены локальные переменные по «*SSA*»-форме как имеется в рисунке 3.2.

Предположим, все остальные блоки, либо искусственные, пустые, либо не содержат определения только что введенных переменных. В блоке № 5: **f**, является процедурой, которая принимает одну переменную. На первый взгляд в этом нет ничего не обычного, однако, если внутри **f** произво-



в блоке № 2:  $b_0=4$ ;  $a_0=b_0+c$ ;  $d_0=a_0-b_0$ ;  
 в блоке № 7:  $b_1=a_0-c$ ;  
 в блоке № 3:  $c_0=b_0+c$ ;  
 в блоке № 5:  $c_1=a_0*b_0$ ;  $f(a_0)$ ;  
 в блоке № 4:  $a_1=a_0+b_0$ ;

Рисунок 3.2: Пример SSA-присваивания по блокам

дится доступ к содержимому параметру, в Си это производится с помощью  $\&$ , тогда переменные могут меняться за пределами  $f$ . К счастью этот сценарий можно выявить с помощью предыдущего анализа всех входных и выходных переменных от  $f$ . Хотя описанный сценарий на практике может встречаться не часто, сценарий является причиной, почему множество оптимизаций не могут совершаться, спасая корректность в общности. Если в блоке № 6  $a$  определяется заново, то он выглядит так:  $a_2 = \phi(a_1, a_0)$ . Функция  $\phi$  является вспомогательной и означает объединение различных определений одной переменной.  $\phi$  неявная функция (отсюда и название с англ. «*phony*», что означает *ненастоящая* или *поддельная*). Концепция неявно определённой функции интересна, с ней мы встретимся позже при определении куч. Такого рода определения данных переменных программ является определением зависимостей данных и может быть применено к любым автоматически выделенным данным в императивных программах. Структура зависимостей в общем случае не может быть деревом, это запрещают  $\phi$ -функции когда имеются зависимости, которые показывают на более ранний блок, который снаружи от циклов.

Как бы ни было, нас интересует, какие подходы имеются для вычисления интервалов видимостей с помощью  $\phi$ -функций [245] локальных переменных. Мы обходимся тем замечанием, что имеются максимальные границы, которые определяют интервалы. Эти границы могут определяться рекурсивно по стеку. Границы вычисляются с помощью графа потока данных и  $\phi$ -функций для каждого из локальных переменных (алгоритмы при поддержке вершин доминаторов представлены, например, в [245], классический подход смотри в [76]). Очевидно, что если переменные определяются в двух разных ветвях заново, то содержание после ветвления может отличаться, а индекс повышается.

Если попытаться применить «SSA»-форму к динамически выделенным переменным, тогда такая попытка не удастся из-за следующих причин: (1) существуют программные операторы, которые выделяют и уничтожают ячейку в динамической памяти явным образом. Эти операторы могут лежать за пределами блоков и процедур. Ячейки могут, безусловно, существовать за пределами процедур и после уничтожения переменной указателя. Это означает, что место определения (в том числе переопределений и уничтожения) и использования сильно отличаются от автоматически выделенных переменных, т.е. места не обязательно совпадают с местами употребления в программе. Даже могут меняться содержимые указателей, когда на указатели, вообще, ничего не ссылается и указатели

уже давно утилизированы. (2) размер, частота и контекст выделения памяти в куче в общности не всегда определены (см. рисунок 2.1).

Аналогично к графу потока данных, можно приписывать не только содержимое переменных к вершинам графа аннотации о свойствах программы [97], но также, например, утверждения о динамической памяти. Примером тому, является фреймворк представленный в [137], который анализирует для каждого указателя при сильно консервативном подходе – возможность о псевдониме для всех остальных указателей (через расширенный алгоритм вычисления транзитивного замыкания). Следовательно, после каждого программного оператора высчитывается не только явно выявленный указатель, но также возможно всё связанное с ним. В качестве структуры данных, используется длинное битовое поле, алгоритм подлежит улучшению, но это не делается, потому, что битовое поле является ключевой структурой данных. Принудительные проверки всех взаимосвязей почти нельзя упростить из-за транзитивности операций сравнения и выявления возможной связанности. Сравнение псевдонимов основано на методе Хорвицы [118] и Мучника [181].

**Наблюдение 3.1** (Организованная память и свежий контекст). *Стек организован. Последовательность элементов в нём определена. Выделение и утилизация всегда происходят при каждом вызове процедур (иногда даже для разветвлений). Свежее выделение памяти означает, каждый раз всё новый контекст, в котором при вызове внутри процедуры все элементы существуют и изначально совершенно независимы друг от друга.*

Далее, можно привести аналогию, при которой каждое состояние динамической памяти записывается как одно состояние. Очевидно, могут иметься любые переходы состояний, но всегда имеется начальная и конечная точка, когда все кучи пусты. При выходе можно смоделировать вспомогательное состояние такого рода, что все конечные состояния присоединяются к общему выходу.

**Наблюдение 3.2** (Неорганизованная память и единый контекст). *Куча (как элемент динамической памяти) неорганизована. С одной стороны — элементы могут находиться в любой последовательности и размещаться по любому. С другой стороны — граф кучи меняется последовательно в каждом программном операторе и выделение новых контекстов отсутствует.*

Например, функциональные языки программирования исходят из принципа независимости данных как главной концепции. Она гарантирует новый и всё свежий контекст. В нём находятся начальные параметры без взаимосвязей. Затем результат присваивается и передаётся высшей инстанции при возврате. Продолжение (с англ. «*continuation*») [180],[273],[78],[255] может быть характеризовано как состояние вычисления, которое передается другой инстанции. При переходе состояние не прерывается, т.к. контекст не меняется. Когда речь идет об одинаковом состоянии, то подразумевается всё состояние вычисления, т.е. прежде всего, нас интересует стек и динамическая память (см. рисунок 2.1). Продолжение хорошо характеризуется денотационной семантикой в случае функций высшего

порядка, но денотационная семантика в общем означает вычисление как функции, т.е. без взаимосвязей с окружающей средой. Отмечается, что продолжения сохраняют и обрабатывают стековые окна. Джоэль [180] рассматривает *продолжения* для функционально-логически смешанного языка «LISP». Основные итоги таковы: (1) продолжения повлекут за собой копирование, вталкивание и выталкивание регионов памяти, адреса переходов и возврата из стека, (2) читаемость и моделирование алгоритмов может быть существенно улучшено. Пункт (1) не может не вызывать озабоченность в связи со скоростью [12]. Пункт (2) приоритетный и противоположен к первому пункту, поэтому на практике необходимо находить разумный компромисс.

**Наблюдение 3.3** (Феномен далёкой манипуляции). *Из-за делимости ячеек памяти и указателей динамической памяти, также наблюдается феномен далёкой манипуляции. Феномен гласит, что изменения в программных операторах, которые в своих (под-)выражениях не содержат некоторую переменную или различные указатели, всё равно могут быть изменены. Этот феномен также не ограничен локально на один блок или процедуру и может повлиять на переменные даже за пределами определяющей процедуры.*

**Наблюдение 3.4** (Интервал видимости переменных). *Интервал видимости указателя, в отличие от статических и логических переменных, может с момента выделения до момента уничтожения прерываться, хотя указатель не меняется (см. рисунок 3.3). Это происходит потому, что куча меняется. Это может произойти нечаянно или намеренно, а также временно при исполнении некоторых программных операторов.*

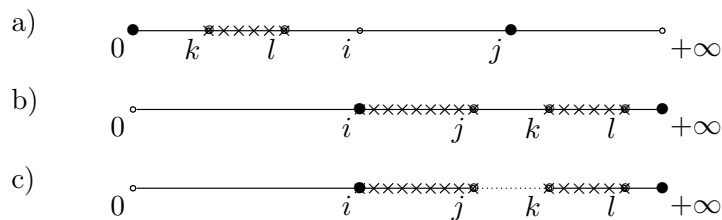


Рисунок 3.3: Видимость локальных переменных в a), b) и динамической в c)

В примере на рисунке 2.3, если `free(o.B)` приводит к тому, что объект *C* утилизируется из динамической памяти, то доступ к *C* через *o2* недоступен. Если *o.B* снова присвоить новый выделенный объект, то *C* об этом не заметит, кроме, если новый объект находится по тому же адресу где находился старый объект. Ради формализма можно согласовать, что всякие указатели присваиваются ради простоты и универсальной модели `nil`, если даже в реальности указатель содержит устаревший адрес. Это означает, что при анализе зависимых указателей необходимо рассматривать также все другие указатели, которые могут быть псевдонимом вершин пути доступа, т.е. *o* или *o.B* (см. проблемы из главы 2). Интерпретация ячеек динамической памяти определяется в зависимости от

типа указателя, который проверяется согласно определению во время семантического анализа. Тип не меняется, ради исключения подклассов (см. главу 4).

### 3.1 Граф над кучами

Теперь, когда основные аспекты динамической памяти были подробно обсуждены (в том числе набл.3.1 и набл.3.2), пора задуматься о представлении графа. Заранее оговаривается, что уточнённые модели динамической памяти будут вводиться в главах 5 и 6. Основными операциями манипуляции динамической памяти являются `malloc`, `free` и манипуляция с указателями.

**Граф динамической памяти как регулярное выражение.** Для начала рассмотрим простой граф  $A_1$  с упомянутыми ранее условиями, который мог бы быть описан регулярным языком и, следовательно, распознан простым конечным автоматом (см. рисунок 3.4).

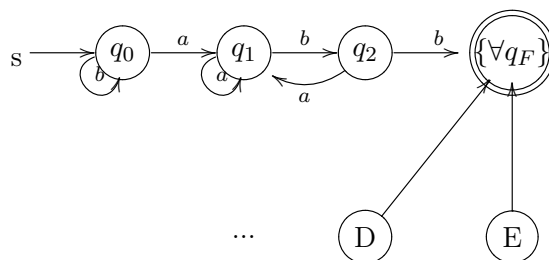


Рисунок 3.4: Пример конечного автомата  $A_1$

С помощью леммы Ардена [79] этот детерминированный автомат может быть представлен следующим регулярным выражением:  $b^*(a^+b)^+b$ . Что теперь означают этот граф и соответствующее выражение? Граф содержит вершины и грани. Вершины представляют собой не пересекаемые ячейки памяти. Имеется некоторое объединённое финальное состояние  $\{q_F | \forall q_F \in F \subseteq Q\}$ . Естественно, все конечные состояния можно обозначить одним состоянием. В графе вершины  $D$  и  $E$  означают, например, некоторые состояния, которые объединяются в  $\{q_F\}$ . Грани означают ссылки, которые записаны в качестве адреса в размере процессорного слова в источнике грани. Возникает первый вопрос: что представляют собой наименования над гранями? Это могут быть указатели или поля объектов, т.е. локации. Оба случая не очень удобны: во-первых, указатели должны выделяться отдельно от ячеек, т.к. они расположены в стеке. Во-вторых, наименования всех граней должны различаться друг от друга. Допустим это так, тогда регулярное выражение уже никак не вписывается в данное компактное представление (см. далее). Это означает, что высокая изначальная компактность сильно страдает. В-третьих, не совсем ясно, что всё-таки означают «начальные» и «конечные» состояния? Начальное состояние можно всегда обозначить переходом одной стековой переменной, это всегда допустимо. Конечные состояния обозначить гораздо тяжелее и неординарно: является ли это обозначение конечным состоянием для вычисления? Если опустить  $q_F$ , то выражение просто не определено. Можно ввести новое состояние, которое принимает от всех состояний те переходы,

которые сигнализируют окончательное вычисление структуры данных.

Допустим, все обозначенные проблемы в данный момент соблюдаются с достаточно удовлетворительным способом и мы продолжим вопрос о внесении изменения динамической памяти с помощью программных операторов после выявления (не-)удобств регулярных выражений. Если запись окажется компактной, то надо проследить, насколько она стабильная при манипуляции. Если имеется маленькая манипуляция, например, меняется только одна грань, то выражение не должно сильно меняться, и тогда можно было бы нотацию считать практичной. Если вдруг запись не устраивает, то необходимо выявить причину и искать другую модель представления памяти. К примеру, для последнего графа добавляется новая грань  $b$ , после ввода граф выглядит как  $A_2$  в рисунке 3.5.

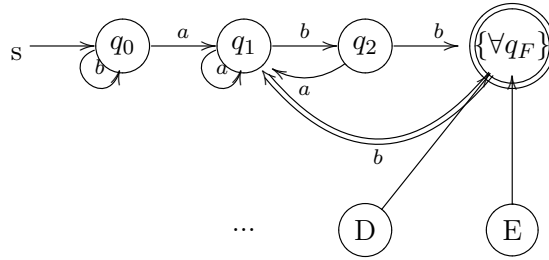


Рисунок 3.5: Пример конечного автомата  $A_2$

Это эквивалентно выражению  $b^*a+b((a+b)^* + bba^*b(a+b)^*)^*b$ . Теперь мы удаляем грань  $a$  и получаем  $A_3$  (см. рисунок 3.6).

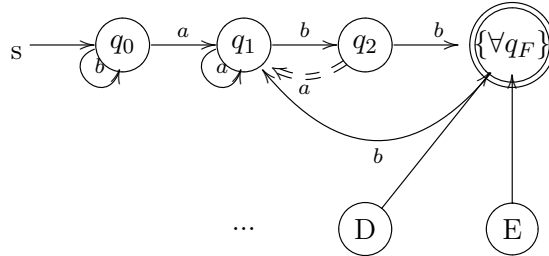


Рисунок 3.6: Пример конечного автомата  $A_3$

и получаем регулярное выражение  $b^*a(a^*bb(\varepsilon + ba^*bb))^+$ . Допускается, что все три выражения могут быть переписаны и далее упрощены, но здесь это не решающий фактор. Проблема заключается в том, что если грань вставляется в любое место графа, а из этого надо исходить, то данное регулярное выражение может очень сильно поменяться, в реальном и худшем случаях практически полностью. Чем больше граф, тем труднее записывать регулярное выражение, которое было бы оптимальным и как можно больше похоже на предыдущее выражение. Причина лежит в том, что система линейных уравнений по Ардену при манипуляции подвергается малым изменениям одной грани. Например, первый граф описывается как в рисунке 3.7.

Нетрудно убедиться в том, что система линейных уравнений имеет решение, но оно сильно отличается от предыдущего. При этом, уравнения почти не меняются. Однако, соответствующая регу-

$$\begin{aligned}
Q_0 &= aQ_q + bQ_0 \\
Q_1 &= aQ_1 + bQ_2 \\
Q_2 &= aQ_1 + bQ_F \\
Q_F &= \varepsilon + \underline{bQ_1}
\end{aligned}$$

$\underline{bQ_1}$  означает добавление отходящей грани от  $q_F$ .

Рисунок 3.7: Равенства описывающие автомат

лярная грамматика сильно отличается.

**Граф при манипуляции.** Сначала необходимо рассмотреть последовательность типичных программных операторов, чтобы обсудить некоторое «адекватное» представление динамической памяти, а также ситуацию с описаниями указателей и переходов. Рассмотрим инверсию списка из [223]. [223] и особенно [196] содержат многие примеры, но остановимся на выбранном примере, который можно вполне считать представительным. Дана следующая программа из рисунка 3.8 на диалекте Си со специфическим синтаксисом касательно указателей.

```

j:=nil;
while (i!=nil){
    k=*(i+1);        // доступ к последующему элементу от i
    *(i+1)=j;        // следующий от i указатель меняется
    j=i;
    i=k;
}

```

Рисунок 3.8: Пример код Си программа для инверсии списков

В программе  $i$ ,  $j$ ,  $k$  обозначают указатели. Доступ к последующему элементу в данной программе реализуется к примеру с помощью неявного оператора  $*(i+1)$ . Подразумевается, что элементы связаны между собой, а не только являются единым, монолитным, непрерывным регионом в динамической памяти, даже если об этом напоминает похожий синтаксис. Семантику программу можно пояснить на примере рисунка 3.9, где номер, означает шаг итерации до посещения цикла.

До входа в цикл,  $i$  содержит список, при выходе  $i$  пуст, а обратный список содержится в  $j$ . Копии не создаются. Входной список итерируется ровно один раз. Замечаем, что грани в примере не подписаны, но это в связи с выбранным неявным определением оператора над указателями. Конечно, в общем грани могут быть подписаны. Однако, существуют указатели, которые являются локальными переменными. Они присваиваются к вершинам графа, либо не присваиваются, тогда, когда указатель неинициализирован. Не инициализированным указателем является  $k$  до вступления в цикл.

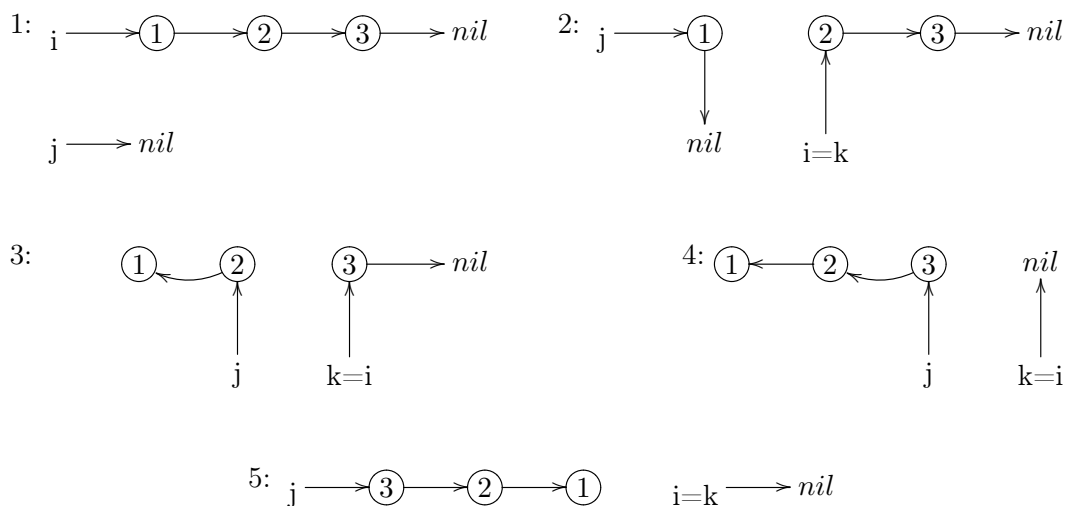


Рисунок 3.9: Пример состояние динамической при выполнение программы

Нетрудно заметить, что компактная нотация даже при простых манипуляциях, например от (1) к (2), совершенно не приспособлена — главная причина лежит в выразимости и адекватном представлении графа. Хотя представленная регулярная запись для данной проблемы не пригодна, всё равно вопрос о компактном представлении также сыграла роль при формулировке автоматизированного подхода в главе 6.

Также нетрудно понять, почему другие глобальные подходы, которые были введены в главе 1, например ВР, АО и т.д. не являются настолько успешными. Причина та же самая, которая была продемонстрирована в последних двух примерах. Подход Доддса [87] далее не рассматривается, хотя он описывает графы до и после трансформации и сосредоточен на описание трансформаций графов, но подход не специфичен для указателей и не рассматривает входной императивный язык программирования (см. главу 1).

**Наблюдение 3.5** (Графовое представление динамической памяти). *Необходимо определить граф динамической памяти (кучу) как тройку  $(V, E, L \times V \cup \{nil\})$ , где  $V$  это множество вершин,  $E$  — множество граней и  $L$  — множество наименований указателей. Указатель ссылается, либо на  $v \in V$ , либо на  $nil$ , т.е. не инициализирован.*

Исходя из наблюдения, граф можно описывать различными способами: (1) или каждую компоненту по отдельности, (2) или предпочесть смешанную форму. Очевидно, что подход (1) не целесообразен потому, что описывать вершины по отдельности (при этом независимо от наименований указателей) может быстро оказаться нечитаемым и не удобным, а спецификация к графу должна быть удобной, короткой и адекватной. Подход (1) требует все вершины графа обозначить по отдельности и затем включить их в спецификацию. Как было обнаружено в примерах введения, такого рода подходы подвергаются целому ряду недостатков и поэтому, в наших целях это не допустимо. Подход (2) подразумевает, либо (2a) **описать вершины**, либо (2b) **границы**, а другую компоненту вы-

явить из данной. Проблемой при описании только вершин (2а), являются дубликаты в спецификации потому, что вершина полностью идентифицируется гранями. Каждая грань имеющая отношение к данной вершине должна приписываться. Это равнозначно тому, что к данной вершине надо иметь список всех соседних вершин. Увы, такой подход очень неудобен. Например, одна вершина связана с двумя или более того вершинами. Это означает, что все соседние вершины также должны вписывать вершину в свои списки. Кроме того, если происходит манипуляция кучи, то, спецификация подвергается сильному изменению, а этого нужно обязательно избегать.

Подход (2b) наоборот описывает только грани, а вершины в них включаются. Этот подход содержит меньше дубликатов и ближе к программе, т.к. ссылки проводятся над реально существующими указателями программы. Если только направленные грани разрешаются, то проверять нужно на половину меньше. Несколько исходящих граней от одной и той же вершины запрещается потому, что один указатель не может ссылаться одновременно на несколько ячеек. Кроме того, программные операторы при более внимательном наблюдении меняют состояние вычисления чаще, чем вершины. — Перемещение, утилизация и выделение являются дорогими операциями, которые включают в себя вызовы операционной системы, а манипуляция указателями является недорогой. В худшем случае, упомянутый подход только выделяет и утилизирует элементы, а указатели мало или вообще не меняются. Тогда принципиально важно задаться вопросом, не подлежит ли подход исправлению?

На рисунке 5.1 (а) указан регулярный граф, вершины которого имеют степень «3», при этом подразумевается, что каждая вершина представляет собой объект, который содержит ровно три указателя. Для подхода (2а) необходимо специфицировать 11 вершин, каждая из которых имеет три грани, при этом, количество входящих и выходящих граней может различаться и это необходимо рассматривать отдельно. В общем, имеется 18 граней. При подходе (2b) необходимо специфицировать только 18 граней. При этом вершины связанные больше чем с одной гранью могут обозначаться символами. Чем больше данный граф отличается от полного графа, тем меньше граней необходимо специфицировать. Если меняется грань, то меньше нужно менять в спецификации, точнее, только меняющуюся грань. Если меняется вершина, то необходимо проверить все связанные грани. Направленный граф можно обыскать за линейное время все левые и правые стороны граней. Согласно подходу (2b) и данному набору вершин с помощью предикатов, проводить доказательства будет удобнее. Для описания вершин используются указатели или выражения доступа к полям (см. главу 4). Не доступные поля при спецификации нас не интересуют, т.к. такие ячейки по определению являются мусором (см. главу 1), навсегда потеряны и не подлежат восстановлению, однако, при спецификации мы заинтересованы выявить такие места, если таковы имеются.

Джоунс [127] определяет *кучу* как непрерывный сегмент операционной памяти размером  $2^k$  с  $k \geq 0$ , который представляет некоторую структуру данных — альтернативно, как последовательность прерывных блоков непрерывных слов. Например, дерево может иметь между вершинами свободные



элементы, но отдельные вершины не интерпретируются иначе, чем как данным(и) процессорным(и) словом(-ами). По Джоунсу объект является множеством ячеек памяти, которые не обязательно связаны между собой, но чьи поля адресуемые. Каждая выделенная ячейка памяти имеет указатель и с момента выделения до момента утилизации возникает вопрос, а жив ли содержимый объект или нет? Трудно не согласиться с Джоунсом в том, что фрагментация является проблемой, однако фрагментация внутри объекта исключается.

Коурмен [72] на стр. 151 определяет, также как и Бурстолл [51] любую структуру данных в динамической памяти, обязательно как деревом. Дерево не только подразумевает некоторую связь « $\leq$ » к дочериним вершинам  $V_j$ , а также обязуется к соблюдению упорядоченности  $f(V_{parent}) \leq f(kid(V_{parent}, j)), \forall j$ . Аталлах [18] рассматривает кучу как массив, интерпретируемый как дерево с более широким использованием памяти, но с более высокой гибкостью. В работе Атталаха можно заметить некоторые различающиеся определения куч. Сначала куча определяется как реализация «*очередь с приоритетом*» (на стр. 79). Приводятся и обсуждаются «*кучи Фибоначчи*» [101] как специализированные и эффективные очереди для добавления и удаления. Далее, на стр. 105 куча определяется как двоичное дерево, сохраняемое все элементы очереди с приоритетом. В отличие от свободного доступа к операционной памяти, Атталах подчёркивает на стр. 111 важность доступа исключительно через имеющийся указатель. Таким образом, произвольная адресация исключается. Ответственность деления куч ложится на мало связанные кучи неявным образом, исключительно, на программиста и на моделирование ПО. Мало связанные кучи можно хорошо делить и обрабатывать эффективными методами. С Коурменом можно не соглашаться касательно произвольной адресации, из-за ранее упомянутых постановлений выразимости. Однако, с ответственностью программиста за создаваемую структуру данных нельзя не согласиться, если даже имеются строгие предпосылки в том, что все структуры данных должны являться деревьями, а не графами — если даже на практике это часто так. Слитор [241] предлагает, для ускорения доступа к куче и минимизации операций сбора мусора, балансировать деревья равняя соседние вершины в отличие от других сбалансированных деревьев, что позволяет произвести быстрый поиск за  $\Theta(n)_{min} = 1$  и удаление за  $\Theta(n) = \log(n)$ . Хотя предложение интересное, всё равно далее подход не будет рассматриваться в данный момент из-за отсутствия острой необходимости реализации, в связи с относительно маленьким объёмом конъюнктов и возможности линейного поиска по локации.

Рейнольдс определяет множество куч, как объединение всех отображений от адресного множества на не пустое значение ячеек памяти. Следуя этому определению, одна куча — это некоторое множество адресов, которые ссылаются на некоторую определённую структуру данных (без дальнейшего уточнения). Рейнольдское определение структуралистское, т.к. куча, как отдельная, отличающаяся и независимая единица просто не существует (см. главу 5). Павлу [199] определяет кучу как любой граф, который не обязательно связан — с этим трудно не согласиться.

## 3.2 Предикаты

Кроме ряда замечаний и конвенций, наиболее важными требованиями остаются: (1) после выполнения каждого программного оператора имеющего отношение к динамической памяти граф кучи меняется наименьшим образом, (2) спецификация соответствующая куче должна также меняться минимальным образом.

Для описания состояния, соблюдая все требования куч, возникает вопрос, как это лучше описать, если очевидно, что сделать это не так просто?

В древней Греции в философской школе Платона эпистемология некоторого описываемого объекта предлагалось известной аллегорией при чётко урегулированном порядке задачи вопросов и ответов между двумя сторонами: человеку, который имеет объект и находится на свободе и человеку, который заперт в пещере и желает понять сущность того объекта с ограниченными возможностями. Запертая персона коммуницирует исключительно речью, а также имеется факел, который горит не бесконечно. Свет факела попадает на обсуждаемый объект и оставляет за собой на пещерной стене тень и силуэты — это неточное изображение и речь, всё, что воспринимает персона в пещере. Эта аллегория лучше известная как «*миф о пещере*». Она предлагает описание объекта через двустороннюю коммуникацию с целью последовательного выявления непосредственных свойств при имеющихся внешних преградах. Существует множество философских «*Геданкенишпилей*», например, лишённый естественной речи диалог, эксперимент искусственного интеллекта «*Китайская комната*» философа Серл. Мы ограничимся мифом о пещере ради классического и древнего характера, который содержит всё, что необходимо для понятия предикатов и куч. Казалось бы, интуитивно понятно, но абстрактное объяснение, получает конкретное присвоение входящих и не входящих свойств (так называемая «*ре-ификация*» — концепция от абстрактной мысли к конкретной реализации). Более современный философический дискурс по течению идеализма наблюдается в классическом подходе Гегеля: тезис, который устанавливается из отмеченных наблюдений, затем выводятся для более тщательного анализа и определения свойств противоположных тезисов, что часто из-за не правильного или неточного определения тезиса приводит к конфликту, который затем разъясняется и выводится общий вывод — синтезированное утверждение. На данном этапе можно считать, что выявленные свойства и требования касательно куч проводились достаточно тщательным образом, чтобы предложить первые предложения. Для обнаружения неточностей и выявления более тщательного определения куч применяется подход, близок к этой концепции.

Что касается реификации, то кучи должны иметь синтаксическое и семантическое обоснование, и они будут основываться на предикатах. Предикаты уже были предложены Аристотелем, которые у него назывались «*силлогизмом*». Силлогизм, это единство трех компонентов логического правила формой: если «*A*» и «*B*», то следует «*C*». Здесь нечего добавить, кроме того, что большинство логических систем основаны на слегка модифицированной модели.

Предыдущая попытка представить кучи с помощью регулярных выражений не увенчалась большим успехом потому, что изменения происходят в переходах графа. Минимальное требование применяется к правилам, но не к выражению, т.к. представление не слишком стабильное для данной проблемы. Но, в общем, выявление и преобразование представления из одной формы в другую очень широко обсуждается и расследуется. Довольно наглядно на примерах клеточных автоматов [278] можно наблюдать за установлением инвариантов выражений. В главе 6 наблюдается обратный подход: наблюдаются образцы, из которых выводятся свойства о правилах.

Предикаты как связывающие вершины графа единицы языков, будь-то формальных/естественных, по семиотике имеют два значения: (i) интуитивное значение, это касается вопроса — *что собой представляет на самом деле «куча»?*, (ii) коннотативное значение — *с чем «куча» ассоциируется?* Далее, задаётся вопрос о представлении кучи: должна/может ли куча использовать символы и реляции и что они означают для ее представления? Может ли куча определяться не полностью?

Оценив (i) нужно заметить, что куча представляет собой множество указателей, которые «*как-то*» связаны между собой и указывают на объекты, которые находятся в динамической памяти. Так как в прошлом большие трудности могли быть выявлены в связи с многозначностью и резкими ограничениями, поэтому необходимо будущие определения редуцировать к минимуму. Оценив (ii) можно выявить, что связь всех компонентов задается данной программой, и куча не организована, это означает: вершины графа могут поступать в любом порядке, в любом месте и непрерывность сегмента памяти естественно не должна соблюдаться. Кучи могут быть связаны с другими кучами.

Важно заметить, что предикат должен иметь не только возможность выразить связь между вершинами графа кучи, но также должен существовать эффективный способ выразить, что две кучи не связаны. Если такой возможности нет, то разделение автоматически получается неявным результатом анализа всех куч, что плохо из-за эффективности. Также у определения связанности имеются различные модусы: «*связан*», «*возможно связан*», «*не связан*», «*возможно не связан*». Модальность кучи, также как указание времени, не имеет большого значения: во-первых, «*связан*» и «*не связан*» можно проверить за линейное время, из графа кучи. Во-вторых, время отслежки дискретно и до/после каждого программного оператора. В предикатах вариации куч с помощью логических операторов необходимо учесть, что логическая дизъюнкция, несмотря на принцип неповторимости и отрицания предиката, должна быть выразительной — этот вопрос решается в главе 4, где доказательство куч будет основываться на логическое программирование. Нам нельзя упустить, что в качестве указателей могут действовать любые допустимые указатели, это локальные и динамические переменные, а также поля объектных экземпляров.

В первоначальной работе по «*логике распределённой памяти*» [224] вводится оператор  $\star$  над кучами и устанавливает законы следующим образом:

**Теорема 3.6** (Свойства динамических ячеек по Рейнольдсу). *Для утверждений о кучах  $p, p_1, p_2$ , множества свободных переменных символов  $FV(.)$  и бинарного пространственного оператора*

дизъюнкции в силе следующие правила (1-6):

- (1) несжимаемость:  $p \not\rightarrow p \star p, p \star q \not\rightarrow p$ , если  $\exists q, q \neq \text{emp}$
- (2) коммутативность:  $p_1 \star p_2 \Leftrightarrow p_2 \star p_1$
- (3) ассоциативность:  $(p_1 \star p_2) \star p_3 \Leftrightarrow p_1 \star (p_2 \star p_3)$
- (4) нейтральный элемент:  $p \star \text{emp} \Leftrightarrow \text{emp} \star p \Leftrightarrow p$
- (5) дистрибутивность:  $(p_1 \vee p_2) \star q \Leftrightarrow (p_1 \star q) \vee (p_2 \star q)$   
 $(p_1 \wedge p_2) \star q \Leftrightarrow (p_1 \star q) \wedge (p_2 \star q)$
- (6) квантификация:  $(\exists x.p) \star q \Leftrightarrow \exists x.(p \star q)$ , если  $x \notin FV(q)$   
 $(\forall x.p) \star q \Leftrightarrow \forall x.(p \star q)$ , если  $x \notin FV(q)$

Несжимаемость подразумевает, что из  $p$  нельзя следовать  $p$  дважды – это совпадает с запретом повторимости описаний [219] кучи. Однако,  $p \star q \not\rightarrow p$ , в отличие от классической логики утверждений, не подразумевает строго  $p$  как отдельное утверждение. Причина не в том, что  $p$  обязательно связана с  $q$  – как это можно предполагать, исходя из описаний и интуиции о делящем операторе, как было введено и продемонстрировано. Это лишь как частный случай. Проблема заключается в описании нахождения элементов в динамической памяти. Это означает, что из утверждения о двух кучах, как бы они не были связаны между собой, не следует, что одна куча вдруг может исчезнуть. Такую разницу обстоятельств надо иметь ввиду.

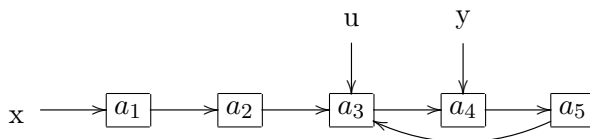
Правила (2-4) понятны и нетрудно обосновать графом кучи. В верности правила (5) можно убедиться, используя индукцию к остальным правилам, при этом, различив два случая, когда  $p_1$  и  $p_2$  конфликтуют и когда не конфликтуют с  $q$ .

Правила (6) кажутся понятными. Эквивалентности с обеих сторон могут казаться очевидными. Предпосылками являются возможные конфликты термов утверждений согласно конфликтным ситуациям, разрешив заранее проводимое переименование, что согласно  $\lambda$ -вычислению конгруэнтно  $\beta$ -конверсии.

Однако, ни правило (1), ни (6) и ни другие, не исключают возникновения свободной переменной в двух кучах, которые связаны с помощью оператора  $\star$ , что и наблюдается в разработках, «smallfoot» или «jStar». То есть, одна переменная может быть использована, например, в качестве указателя в одной куче, а в качестве используемой ссылки в подтерме, в другой куче.

Рассмотрим кучу из рисунка 3.10.

В куче имеются указатели  $x, u, y$ . В прямоугольниках находятся содержимые  $a_1, a_3$  и  $a_4$ . Бурстолл предлагает обозначить содержимое не напрямую, а лишь на адрес. Также он предлагает записывать в минимальную модель все промежуточные ссылки или наименования, таким образом, можно большую часть кучи описать лишь одним выражением  $x \xrightarrow{a_1, a_2, a_3} y$ . Это выражение означает многое: существуют указатели  $x$  и  $y$ , существует гарантированный путь между ними и естественно весь путь описывает отрывок графа кучи. В линейных списках, а также в деревьях указатели `next`, если тако-

Рисунок 3.10: Пример кучи с указателями  $x, u, y$ 

вы имеются, упускаются по умолчанию. Фактически одно выражение по Бурстоллу описывает весь линейный список. Совокупность таких выражений описывает граф кучи. Рассмотрим следующую структуру данных в виде “кактуса” (см. рисунок 3.11):

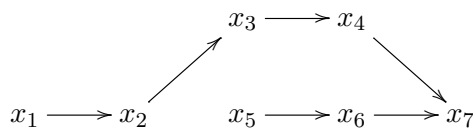


Рисунок 3.11: Пример кактуса динамической памяти

Для описания понадобятся лишь два выражения.  $x_1$  является указателем, который выделен на стеке. Допустим, все остальные ячейки не имеют указателей, следовательно, она расположена в динамической памяти. Удобная, хотя и компактная, запись ломает обозначенные ранее критерии минимальности. Поэтому, запись Бурстолла далее не используется, но, рассматривается возможность абстракции с помощью предикатов для более гибкого описания, которое основано на гранях графа.

Для полноценного анализа указателей и динамической памяти со стороны выразимости нет необходимости разрешать доступ к динамической памяти с любыми выражениями вычисления адреса. Все операции доступа целиком могут быть выявлены писанием и чтением динамических ячеек. Если по причинам быстродействия нужен произвольный, последовательный доступ к памяти, то это является типичным сценарием применения стека. Однако, выделение любого доступного размера, допускается как для стека, так и для динамической памяти. Надо понимать, что память выделяется как один непрерывный сегмент и проверка на «*неверные*», «*висячие*» ячейки и на свойство «*мусора*» там, естественно, не проводится. Нетрудно себе представить вспомогательные функции, которые могут произвести любые преобразования типов с данным сегментом памяти, поэтому этот вопрос далее не рассматривается. Обоснования и путь доступа к объектам имеют только указатели. Объектные экземпляры рассматриваются как структуры, выделенные в динамической памяти, в отличие от классических структур, например в Си [243], которые, либо расположены на стеке,

либо в процессорных регистрах. Второй вариант принципиально не исключает подключение динамической памяти. Нужно отметить, что  $e_1.f_1 \mapsto val_1$  может означать, что объект  $e_1$  может иметь ровно два или более полей и в откомпилированном коде  $e_1.f_1$  совершенно иным путем будет реально обрабатываться и располагаться, чем  $e_1.f_2$  в процессе различных оптимизаций кода [135],[181],[252] и, несмотря на введенные конвенции, всё равно это так. Это не означает раздробление на уровне входного языка и спецификации, где объект должен моделироваться как единый не делимый регион памяти, где поля могут иметь ссылки иных экземпляров.

Как во введении уже было изложено, в ЛРП были внесены целый ряд предложений [224], [223], [51], [121], [193],[195],[194], [44], [281], [27], [26],[190], [233] (см. главу 1). Рейнольдс [224] вводит *оператор последовательности* «,» для неявного определения линейного списка. «Неявно» означает, что не уточняется, каким образом последовательные содержимые связаны между собой, лишь уговаривается, что элементы связаны. Таким образом, из раннего примера кактус можно определить  $x_1 \mapsto x_2, x_3, x_4, x_7 \wedge x_5 \mapsto x_6, x_7$ . Альтернативы неявному определению Рейнольдса можно считать, либо явное определение, которые всегда допустимое и полностью покрывает неявный оператор, либо по определению ограниченные по длине списки Бозга [45].

Если мы хотим параметризовать кучу, необходимо вводить символьные переменные в утверждениях. Утверждение верное или неверное для данной кучи. Символы априори не типизируются, как термы по Чёрчу, а информация о типе поступает из окружения, таким образом, оно более похоже на типизацию по Карри. Вводя (символьные) переменные, определение предиката относится как параметризованный терм в  $\lambda$ -вычислении, т.е. предикат абстрагирован и подлежит к применению с другими предикатами. Однако, предикат не возвращает иного результата как «истина» или «ложь», а присутствие входных и выходных данных отличается от классических функций (см. главу 4). Рассмотрим рекурсивный пример двоичного дерева из [224]:

$$tree(l) ::= \text{nil} \mid \exists x. \exists y : l \mapsto x, y \star tree(x) \star tree(y)$$

Согласно определению Рейнольдса, оператор  $\star$  определит кучу, которая состоит из двух разделяющихся куч. Нужно отметить, как было упомянуто ранее, что изначальное определение  $\star$ -дизъюнкции может иметь соединяющие элементы. Таким образом, от одного дерева  $x$  всё-таки можно попасть в соседнее дерево  $y$ , несмотря на то, что имеется  $tree(x) \star tree(y)$  и предполагается, что весь регион под  $x$  действительно не пересекается с регионом под  $y$ . Нужно, чтобы без изменения данного предиката это могло оказаться невозможным. Однако, при дальнейшей параметризации и при манипуляции предиката (ср. предикат **tree** в главе 4) проблема принципиально остается. Более наглядно это можно увидеть в рисунке 3.12.

На основе определения по Рейнольдсу, Бердайн [26] вводит соотношения выполнимости куч в опр.3.7.

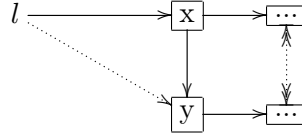


Рисунок 3.12: Пример схематической делимости динамической памяти

**Определение 3.7** (Соотношение выполнимости интерпретаций куч). Для этого используется понятие « $\models$ » модели формул куч  $s \in \Sigma$ , стекового указателя  $h \in \Pi$  и  $r(t_i)$  как  $t_i$ -ая компонента структуры  $r$  (см. рисунок 3.13).

$s \models E = F$	если $\llbracket E \rrbracket s = \llbracket F \rrbracket s$
$s \models E \neq F$	$\llbracket E \rrbracket s \neq \llbracket F \rrbracket s$
$s \models \Pi_0 \wedge \Pi_1$	$s \models \Pi_0$ и $s \models \Pi_1$
$s, h \models E_0 \mapsto t_1 : E_1, \dots, t_k : E_k$	$h = [\llbracket E_0 \rrbracket s \rightarrow r]$
$s, h \models \text{emp}$	$h = \emptyset$
$s, h \models \Sigma_0 \star \Sigma_1$	$\exists h_0, h_1. h = h_0 \star h_1, s, h_0 \models \Sigma_0, s, h_1 \models \Sigma_1$
$s, h \models \Pi \wedge \Sigma$	$s \models \Pi$ и $s, h \models \Sigma$ .

Рисунок 3.13: Формальное определение кучи по ЛРП

Денотационная функция  $\llbracket \cdot \rrbracket$  имеет тип  $\Phi \times \Sigma \rightarrow Bool$ , где  $\Phi$  множество утверждений о куче, а  $Bool$  булево множество. Для линейного списка  $s, h \models E_0 \mapsto t_1, \dots, t_k$  с соответствующими типами  $\forall i, j \in \mathbb{N}_0. E_j, r(t_i) = \llbracket E_i \rrbracket s, 1 \leq i \leq k$ . Слева от  $\models$  записывается состояние вычисления, которое имеет тип  $\Pi \times \Sigma$ , справа имеется любое булево утверждение.

В [27] Бердайн указывает на проблемы, что фрейм может дистанционно поменяться (см. набл.3.3). Это является проблемой, которую предикатам необходимо учесть. Однако, удаление содержимого ячейки памяти, на которой ссылается глобальный указатель, является отдельной проблемой (см. главу 1). Аналогичное касается подпроцедур, которые далее здесь не рассматриваются. Необходимо отметить, что встроенные процедуры с точки зрения выразимости в общем случае могут лишь усложнить спецификацию и верификацию, но вычислимость они не увеличивают. Бердайн справедливо обращает внимание на то, что применение правила фрейма в общем случае может привести к определённым трудностям в связи с недетерминированностью сопоставлений символов. Однако, когда речь идёт о кучах, недетерминированность можно ограничить наименованиями и дополнительными конвенциями (см. главы 5,6). Более того, логические конъюнкты вписываются прямо в язык логического программирования (см. главу 4).

Далее уточняем определение графа кучи согласно [294] и затем вводим синтаксическое и семантическое обозначение согласно предыдущему анализу.

**Определение 3.8** (Конечный граф кучи). *Конечный граф кучи является направленным связанным графом, который расположен в динамической памяти. Граф может содержать циклы, но между двумя вершинами графа разрешается не более одной грани. Каждая вершина имеет содержание, тип и адрес ячейки памяти и занимает последовательный регион памяти. Конечный адрес получается из начального адреса и размера, который получается из типа. Вершины графа не пересекаются. Ради простоты, но без ограничения общности, каждая грань ссылается на абсолютный адрес в динамической памяти.*

Неявные определения, например, в «SSA»-форме, пользуются успехом, несмотря на то, что чёткого определения, например «зависимости данных» нет и не нужно. В изначальных определениях также имеются неявные определения, которые касательно куч уточняются в этой работе. Соотношения, пространственные операторы и частично неявное определение касаются вершины графа, типизацию символьных переменных и т.д. В главе 5 пространственные операторы куч ужесточаются и для константных функций вводится дополнительное обозначение для объектов.

Из трм.3.6, опр.3.8 и предыдущих конвенций (см. [294]) терм кучи может быть определен следующим минимальным образом:

**Определение 3.9** (Терм кучи). *Терм кучи  $T$  описывает граф кучи следующим образом:*

$$\begin{aligned}
 T ::= & \text{ loc} \mapsto \text{val} && \dots \text{ обыкновенная (базисная) куча} \\
 & | T \star T && \dots \text{ конъюнкция} \\
 & | \underline{\text{true}} \mid \underline{\text{false}} \mid \underline{\text{emp}} && \dots \text{ константные предикаты куч} \\
 & | ( T ) && \dots \text{ скобочное выражение}
 \end{aligned}$$

где  $\text{loc}$  обозначает локацию. Локацией может послужить сложное выражение или символ представляющий кучу.  $\text{val}$  является совместимым типом вершины графа с обозначением.

$\underline{\text{true}}$  обозначает тавтологию независимо от того, как выглядит данная куча. Обратное действительно для  $\underline{\text{false}}$ . Предикат  $\underline{\text{emp}}$  верный только тогда, когда данная куча пуста, во всех остальных случаях ложна. В главе 5 конъюнкция ужесточается и распадается на две операции. Для логических утверждений вводятся логические конъюнкции в рекурсивное определение  $T$ .

**Определение 3.10** (Расширение термов куч). *Определение термина  $ET$  является расширением  $T$  из опр.3.9. Оно включает логические конъюнкции и определено как:*



$ET ::=$	$T$	... терм кучи
	$  p(\alpha)$	... вызов абстрактного предиката
	$  \neg ET$	... логическое отрицание
	$  ET \wedge ET$	... логическая конъюнкция
	$  ET \vee ET$	... логическая дизъюнкция

Логические конъюнкции « $\wedge$ ,  $\vee$ ,  $\neg$ » не нуждаются в объяснении. Вывоз предиката подразумевает, что соответствующий предикат определен в  $\Gamma$  (при опр.6.5 и закл.6.6). При запуске предиката с целью сравнения с актуальной кучей, все свободные символы должны быть унифицированы термами не содержащие свободные переменные, иначе данный запуск не определён (см. главу 4).

Если в связи с символьными переменными использовать логический язык программирования, то ограничения как одностороннее присвоение и невозможность использования символа вместо значения и многие другие ограничения [26], [27], [195], [193] можно будет снять. Если унифицировать термы, то сравнение простое и «дыры» наполняются нужным содержанием, иначе нужно вручную все подтермы сравнивать и вставлять необходимые термы в нужные места подтермов. Это возможно, но необходимы дополнительные условия, вследствие чего, вводятся всё новые ошибки и ограничения. На практике ограничения наблюдаются в основном тогда, когда ради используемого нелогического языка упускаются полные сравнения или деградируется символьное использование полностью, как вызов по значению. Обычно это наблюдается в императивных и большом количестве функциональных языках программирования.

Возьмем к примеру вызов предиката (подробно о логическом представлении в главе 4)

«?-pred1(s(s(zero)),\_)»

где ради простоты первый терм входной, а второй выходной. Запрашивается, существует ли таким образом, анонимная переменная «\_», чтобы предикат **pred1** был выполнен для входящего термина **f(a)**? Если ответ верный, то подцель успешная и результат забрасывается. Если нет, то предикат **pred1** не соблюдается. **s(s(zero))** представляет собой целое число Чёрча «2». Если например, вместо **s(s(zero))** представить **s(s(\_))** и возможно предъявить результат, например **s(s(s(zero)))**, то без изменений запрос (см. опр.4.3) можно поменять на «?-pred1(s(s(\_)), s(s(s(zero))))», подразумевая, что предикат определён двунаправлено. Принципиально это касается не только двух, а нескольких направлений. Пролог имеет строгий порядок присвоения и вычисления термов слева направо. Это означает, что символы могут замещать конкретные кучи, но если подцель потребует конкретную кучу, а куча присваивается только в одном из следующих подцелях, то можно, либо порядок подцелей поменять, либо вычисление не завершается успехом (см. главу 4).

Теперь необходимо рассмотреть свойства отображения между определением куч и графом куч, а также свойства отдельных ссылок.

**Свойство 1 – Корректность.** Если из синтаксического опр.3.9 следует строгое различие между связанной и несвязанной кучей (см. главу 5), то синтаксическое описание охватывает любой граф кучи, а также любой граф кучи может быть представлен данным синтаксическим определением. Если нормализовать согласно правилам трм.3.6, например, по *пренекс-нормальной форме*, то таким образом все, полученные формулы коммутуются. Константные предикаты являются исключением, и поэтому являются односторонним укрупнением: множество выполнимых куч отображается на один представитель множества. Не трудно убедиться в том, что такое отображение не обратимое. Если ещё исключить анонимные символы, то синтаксическое описание полностью соответствует графу куч. Исключив единственные очаги недетерминированного представления, легко убедиться в том, что отображение теперь изоморфное, а, следовательно, не могут быть выражены два различных графа куч из одного описания и наоборот.

**Свойство 2 – Полнота.** Очевидно, что граф кучи полностью описывается базисными кучами и аннотируется ссылками. Представление значения вершин графа может привести к синтаксическому парадоксу, если не различать адреса ссылаемого объекта. Например, если  $a \mapsto 3$  а также имеется  $b \mapsto 3$ , то на практике это вовсе не означает, что обе ячейки памяти содержавшие «3» идентичны. Для моделирования это именно то и означает. Если именно новый объект не выделяется и указатель на эту же ячейку не ссылается, то указатель становится псевдонимом. Избежать этой ситуации можно с помощью аннотации в объектном виде термина.

Указатели на указатели содержат целое число как адрес, и поэтому не отличаются от других указателей. Различие между целым числом и адресом производится за счёт типа переменной. При отображении от графа к формуле порядок вычисления и структура предикатов естественно не могут быть выведены однозначно. Отображение может быть сгенерировано, но оно может различаться. Если допустить, что дан набор определений абстрактных предикатов, то отображение в общем случае не решимо из-за проблемы приостановки. Отображение от графа кучи к  $T$  полностью определено, соблюдая упомянутые особенности. Обратное отображение очевидно полное. Если в  $loc \mapsto val$  левая сторона не указатель, то аннотацию можно принципиально решить дополнительным полем  $f: loc \mapsto_f val$ . Для общей структуры графа кучи дополнительная аннотация не имеет большого значения, поэтому дополнительные поля умалчиваются по определению.

**Свойство 3 – Отношение эквивалентности.** Для сравнения может быть использовано, что « $\mapsto$ » бинарный функтор, а  $a \mapsto b$  куча. Таким образом, можно убедиться в том, что отношение эквивалентности « $\sim$ » может быть обосновано, показав свойства (i-iii). Оговаривается, что « $\mapsto$ » имеет выше приоритет присваивания, чем « $\sim$ ». (i) Рефлексивность:  $a \mapsto b \sim a \mapsto b$ . (ii) Симметричность:  $a \mapsto b \sim c \mapsto d$ , то  $c \mapsto d \sim a \mapsto b$ . (iii) Транзитивность:  $a \mapsto b \sim c \mapsto d$  и  $c \mapsto d \sim e \mapsto f$ , то  $a \mapsto b \sim e \mapsto f$ .

**Свойство 4 – Локальность.** При удалении, изменении, добавлении указателя или его содержимого, граф кучи и соответствующий терм меняются минимально. Удаление грани приводит к редукции на одну базисную кучу, либо к параметризации или изменению используемых абстрактных предикатов, т.к. соотношение между обоими, формулой и графом, в общем случае не решимо (см. ранее), поэтому, необходимо предикаты рассматривать отдельно. В свободном от предикатов случае максимальное изменение может повлечь за собой удаление вершины, потому, что это означает удаление вершины и всех граней, которые с этой вершиной связаны. Аналогичное добавление не является «сложной», потому, что добавление грани производится пошагово. В худшем случае для предикатов — использование предикатов может привести к формуле, которая совершенно не похожа на предыдущую. Поэтому, рекурсивные структуры эффективно описываются рекурсивными описаниями и изменение одного элемента не затрагивает остальные элементы, кроме соседних. В общем случае, это лишь эвристика и зависит от конкретного алгоритма и от делимости графа на наиболее независимые подграфы. Также как и эвристика: *«граф кучи описывается абстрактными предикатами лучше, чем компактными описаниями»*. Делимость проблем остается общей проблемой далеко за пределами этой работы, но возможность делить кучи на «удобные» кучи. Это интересно с точки зрения подключения логических решателей (см. главы 5, 6, см. [85]).

В заключение этого раздела, хотелось бы сказать о предикатах высшего порядка. Они не имеют теоретическую значимость для сравнения куч с практической точки зрения. Принципиально предикаты, которые используют предикаты в качестве параметра, интересны с точки зрения выразимости простых и коротких описаний. Однако, рассматриваемые предикаты куч имеют индуктивное определение, а произвольные предикаты высшего порядка в состоянии сломать эти свойства, если не вводить дополнительные ограничения. Поэтому, считается не целесообразно использовать иное, чем формально-грамматическое представление (см. главу 6). Нужно отметить, что при включении предикатов высших порядков меняются в общности существенные свойства доказательств, как например, поток и вызовы с продолжениями, порядок вычисления и обработки подцелей, а также статическая типизация, но с точки зрения выразимости кучи ничего не меняется, по крайней мере после обсуждения такая необходимость отпадает.

## 4 Логическое программирование и доказательство

В этой главе рассматривается вопрос, как куча (см. главу 3) может быть представлена в Прологе, а затем теоремы о кучах логически выведены с помощью Пролога. Для этого берётся Пролог и определяется синтаксис термов и правил, обсуждается отсечение и применимость рекурсивных определений. Это способствует логическому выводу, который будет решать задачи верификации. Более детально анализируется декларативный характер абстрактных предикатов, который в главах 5 и 6 используется для сближения языков спецификации и верификации. Представление языков с помощью Пролога выявляется на основе выразимости реляций, а также на основе метрик. Предлагается система, основанная на Прологе. Представление и интеграция объектных экземпляров обсуждается.

### 4.1 Пролог как система логического вывода

Данный раздел не представляет собой введение в Пролог, а лежит в основе построения архитектуры верификации куч на основе Пролога. Тем не менее, раздел ссылается на первоисточники Пролога, как [248] и [46], которые рекомендуется изучить досконально, прежде чем, читать далее. Пролог уже оправдался решением трудных теоретических и практических проблем. Например, при доказательстве теоремы Фейгенбаума [142], при доказательстве ошибки с делением вещественных чисел в процессорах «*Intel Pentium*» первого поколения с помощью логических диалектов «*ACL2*» [133]/«*HOL Light*» [213] или при обработке и проверке слабоструктурированных данных [292].

Программа в Прологе задаётся базой знаний, которая задаётся правилами Хорна. Запрос к базе знаний можно задавать одной или более подцелями. Для определения правил и подцелей необходимо ввести определение прологовского выражения терма.

**Определение 4.1** (Генеричный терм в Прологе). *Терм  $T$  в Прологе определён как:*

$$T ::= \begin{cases} x & \text{символ } x \in \overline{X} \text{ из множества допустимых символов} \\ X & \text{символьная переменная } X \in \overline{X} \\ [] & \text{пустой список} \\ [T \mid Ts] & \text{терм голова списка } T \in \overline{X}, \text{ а где } Ts \text{ список остаток} \\ [T_0, \dots, T_n] & \text{список с } T_j \text{ терм, } 0 < j \leq n \\ f(T_0, \dots, T_n) & f \text{ функтор, } T_j \text{ термы, } 0 \leq j \leq n \\ p(T_0, \dots, T_n) & p \text{ предикат, } T_j \text{ термы, } 0 \leq j \leq n \end{cases}$$

Символ представляет некоторый логический объект, например: «я», «дед мороз», число «33» или некоторая локальная переменная. В Прологе символ начинается всегда с маленькой буквы, а далее следуют любые буквы или цифры.

В отличие от символа, символьная переменная всегда начинается с большой буквы, либо является зарезервированным знаком «\_». Например, переменной  $X$  присваивается (*унифицируется*) некоторое значение, например «33», после чего  $X$  может быть использован далее в сложных термах или подцелях (см. опр.4.2). Когда используется «\_», тогда ссылаться на это же значение будет не возможно. Поэтому, «\_» используется исключительно в тех случаях, когда должен приниматься ровно один терм, но дальнейшее использование этого значения не предусмотрено, как это обычно бывает в случае сопоставления с образцами термов. Видимость переменных ограничивается данным правилом.

Списки, которые определяются с помощью бинарных функторов «,» или «|» должны иметь хотя бы два компонента. Проверка, является ли список линейным, проводится самими предикатами, которые принимают термы. Примерами списков являются:  $[12 \mid []]$  или  $[1, 2, [4 \mid 5]]$ . Таким образом, базисный тип списков является записью.

Функтор является структурным оператором, который связывает термы и обозначает новое сложное значение. Значение может быть символьным, аналогично объектному экземпляру или записи. Например, функтор списка конструктор «.», который, применив к голове  $H$  и списку  $HS$  работает аналогично  $[H \mid HS]$ . Иной пример, это наследник натурального числа *succ*, который имеет арность 1, либо принимает терм Чёрча по арифметике, который опять же снаружи имеет функтор *succ*, либо терм *zero* с арностью 0, т.е. является константной (см. набл.4.13).

На вопрос «да/нет» предикат даёт ответ в зависимости от того, связаны ли некоторые термы согласно данному соотношению или нет — если предикат тотален (см. следующие главы). Например, высказывание *older(plato, aristotle)* означает утверждение «Платон старше Аристотеля».

**Определение 4.2** (Правило Хорна). *Правило в Прологе состоит из головы  $p$  и тела  $q_0, q_1, \dots, q_n$  для каждой подцели  $q_j$  и  $j, n \in \mathbb{N}_0, j \leq n$ . Подцели вычисляются последовательно для  $j \geq 0$ . Голова  $p$  может содержать любое количество термов (вектор термов), которые могут быть использованы в теле. Правило с пустым телом, где  $j = n = 0$ , называется фактом.*

Синтаксис правила *pred* в расширенной форме Бэккуса-Наура можно определить как:

$$\begin{aligned}\langle head \rangle &::= \langle ID \rangle \langle ' \rangle \langle term \rangle \{ \langle ' \rangle \langle term \rangle \} \langle ' \rangle & \langle goal \rangle &::= \langle term \rangle \langle rel \rangle \langle term \rangle \mid \langle call \rangle \\ \langle rel \rangle &::= \langle '=' \rangle \mid \langle '!= ' \rangle & \langle body \rangle &::= \{ \langle goal \rangle \langle ' \rangle \} \langle goal \rangle \\ \langle call \rangle &::= \langle ID \rangle \langle ' \rangle \langle term \rangle \{ \langle ' \rangle \langle term \rangle \} \langle ' \rangle & \langle pred \rangle &::= \langle head \rangle [ \langle ':- ' \rangle \langle body \rangle ] \langle ' \rangle\end{aligned}$$

Здесь *ID* идентификатор, который является символом, но не символьной переменной. Бинарные операторы «=» и «!=» обозначают унификацию, либо утверждение о невозможности унификации данных термов. «.» обозначает конец определения правила. Символьные переменные видны только внутри одного определения правила. Разделитель «:-» определяет голову *head* от тела *body* правила.

Для иллюстрации, рассмотрим примеры из рисунка 4.1. Рисунок 4.1 а) содержит два очевидных факта из древнегреческой мифологии: (1) Сократ человек и (2) Цевс бессмертен. Аналогично можно определить иные факты, за достоверность, за что отвечает создатель базы знаний. Фактом по умолчанию является только неоспоримое утверждение из проводимой области дискурса. Третье правило гласит: «любой человек смертный». Технически более подробно это означает: если некоторый терм *X* имеет предикат «человечно», то терму *X* безусловно приписывается предикат «смертно». «Безусловно» подразумевает в прямом смысле отсутствие дальнейших подцелей, кроме подцели *human(X)*.

$\begin{aligned}\text{human(socrates)} . \\ \text{noneternal(zeus)} . \\ \text{mortal(X):-human(X)} .\end{aligned}$	$\begin{aligned}\text{a2(0,M,Res):-Res is M+1.} \\ \text{a2(N,0,Res):-N1 is N-1, a2(N1,1,Res).} \\ \text{a2(N,M,Res):-N1 is N-1, M1 is M-1,} \\ \qquad \text{a2(N,M1,Res2),} \\ \qquad \text{a2(N1,Res2,Res).}\end{aligned}$
(a)	(b)

Рисунок 4.1: Факты и правила в Прологе на примере предиката Аккерманна

Унификация термов из опр.4.1 является в частном случае инфикс-нотации, опираясь на *r1*. Более обобщённое соотношение записывается с помощью предиката  $p(T_0, \dots, T_n)$ . Далее, рисунок 4.1 б) на примере *функции Аккерманна* демонстрирует, что любая рекурсия (левая, правая, примитивная, взаимная, и т.д.) может быть определена в Прологе — не только примитивная. Оператор *is* является арифметическим функтором. Функторы по Карнапу [57] являются арифметическими вычислениями термов и не являются логическими. Функция Аккерманна является определённой и тотальной, однако, на практике, из-за ограничения памяти и операционных средств вычисление может быть прервано. По этой причине, предикат полученный из функции может оказаться полезной для проверки терминации алгоритмов для индуктивно-определённых входных данных. Примером могут послужить натуральные числа или списки.

Подцели *goal* как пересчитываемые подусловия выполнимости предиката обозначаются следующим образом:

**Определение 4.3** (Запрос в Прологе). *Запрос подцелей в Прологе определяется как последовательность унификаций вызовов определённых правил. Связанные символы вталкиваются в среду символов и унифицируются с каждым вызовом подцели. Вызов подразумевает подходящий предикат с подходящей аргументностью, где все вызывающие предикаты, в отличие от термов, должны быть полностью определены во время вызова. В случае, когда имеются несколько альтернатив вызова, то предикат выбирается ближе к началу программы, а остальные предикаты рассматриваются обязательно позже в случае присутствия альтернатив. Альтернативы могут быть исключены с помощью отсечения (см. позже). Альтернативы рассматриваются по порядку рекурсивного подъёма при процедурном вызове предикатов. Основной разницей являются процедурные вызовы, передача аргументов термов и принудительный поиск альтернативов (см. позже, см. опр.1.5).*

Интроспекция в Прологе [83] разрешает проверку и определение типы классов, так называемые «виды», данного терма, которые могут быть: `var`, `atom`, `number`, `compound` и иные мало значимые встроенные предикаты. Слияние общих случаев термов к одному виду разрешается, а следовательно, предикаты могут сильно упростить определение предикатов. `atom` проверяет свойство символа, например `atom(a)` или `atom([])` верны, но `atom([1])` или `atom([1,2])` не верны. `var` проверяет, является ли данный терм символьной переменной, которая свободная. Поэтому, `var(X)` верно, но `X=1, var(X)` не верно. `list` проверяет, является ли данный терм (слабо-типизированным) списком (как это принято считать в Прологе, ср. [83] с [243]). Таким образом, `list([])` или `list([1,2,3])` верны, а `list(a)` не верен. Необходимо отметить, что для определения свойства структуры списка, также как и для встроенного предиката `compound`, используется встроенный предикат «=`..`», который разбивает данный функторный терм на сам функтор и на список передаваемых термов.

Вызов предикатов рассматривается чуть позже. Однако, унификация термов по определению не производится в Прологе по умолчанию из-за необходимости полного анализа всех подвыражений термов. Поэтому, можно по определению унифицировать `X=X`, но `X=f(g)` нельзя. Пролог, в зависимости от реализации, может замечать рекурсию в термах, т.к. иногда производится унификация исключительно на верхнем термовом уровне. Но, в примере `X=f(g(X,X))` часто унификация приведёт к провалу или приостановке в лучшем случае, а в обычном случае к выходу WAM из строя (из-за переполнения стека в версии 1.3.0 «GNU Prolog» [83]). По осторожным оценкам на практике в 95% всех случаев не требуется проверка унифицируемости термов, однако, в общем случае необходимо рассматривать именно это, когда речь идёт об обобщённых утверждениях оценок стабильности, например преобразователей. Также нужно рассматривать границы ресурсов, например с помощью предиката Аккерманна. Поэтому, далее даётся алгоритм, который позволит полностью исключить те 5%, которые корреспондируют с проблемой, например, с определением объектного экземпляра

из раздела 1.2, см. рисунок 4.2.

```
unify_with_check(X,Y):-var(X),var(Y),X=Y.
unify_with_check(X,Y):-var(X),nonvar(Y),not_in(X,Y),X=Y.
unify_with_check(X,Y):-nonvar(X),var(X),not_in(Y,X),Y=X.
unify_with_check(X,Y):-nonvar(X),nonvar(Y),X=..[H|L1],Y=..[H|L2],
    unify_list(L1,L2).
```

Рисунок 4.2: Пример кода унификации с проверкой рекурсивного повтора

Термы структуры и списки необходимо определить как изложено в рисунке 4.3.

```
unify_list([],[]).
unify_list([H1|L1],[H2|L2]):-
    unify_with_check(H1,H2),
    unify_list(L1,L2).

not_in(X,Y):-var(Y),X\==Y.
not_in(X,Y):-nonvar(Y),Y=..[_|L],not_in_list(X,L).

not_in_list(X,[]).
not_in_list(X,[H|L]):-
    not_in(X,H), not_in_list(X,L).
```

Рисунок 4.3: Пример кода унификации списков

В главе 6 и на рисунке 6.3 дан пример, когда предикат используется в качестве передаваемого терма. Стоит заметить, что аналогично анализу функтора, вызов предиката также производится для данного стекового окна. Разница заключается в представлении вычислительной модели реализации Пролога и синхронизации стека (см. следующий раздел) между вызванной и вызывающей сторонами.

Далее рассмотрим пример из рисунка 4.1 b) с подделями и их вызовами. Дана программа и дана подцель «?-a2(X,1,3)» (см. рисунок 4.4), которая задается интерактивным интерпретатором. Сначала берётся правило № (1) с сопоставлением  $\sigma_1$ . Это приводит к противоречивости, т.к. Res=2. На рисунке 4.5 определена система сопоставлений для данного примера дерева вывода). Поиск по данному предикату приостанавливается, т.к. завершился провалом и берется следующий альтернатив (см. опр.1.5). Выбрав третье правило с сопоставлением  $\sigma_2$ , а затем, применив второе правило с  $\sigma_5$  и наконец, применив первое правило с  $\sigma_9$ , успешно находит решение. Так как отсечение (см. далее) не производится, проверяются также альтернативы и наконец находится ещё одно положительное ре-



шение данной подцели. Путь от начальной подцели через все новые подцели до успешного решения, которые на рисунке обозначены чёрными квадратиками, назовём «*путь к решению*». Для решения данного примера существуют два пути, при этом, результаты не отличаются друг от друга. Нужно обратить внимание, что если даже пути к решению доказательства различны, то после вычисления первого, решение можно было бы полностью приостановить. Однако, это нельзя обобщать.

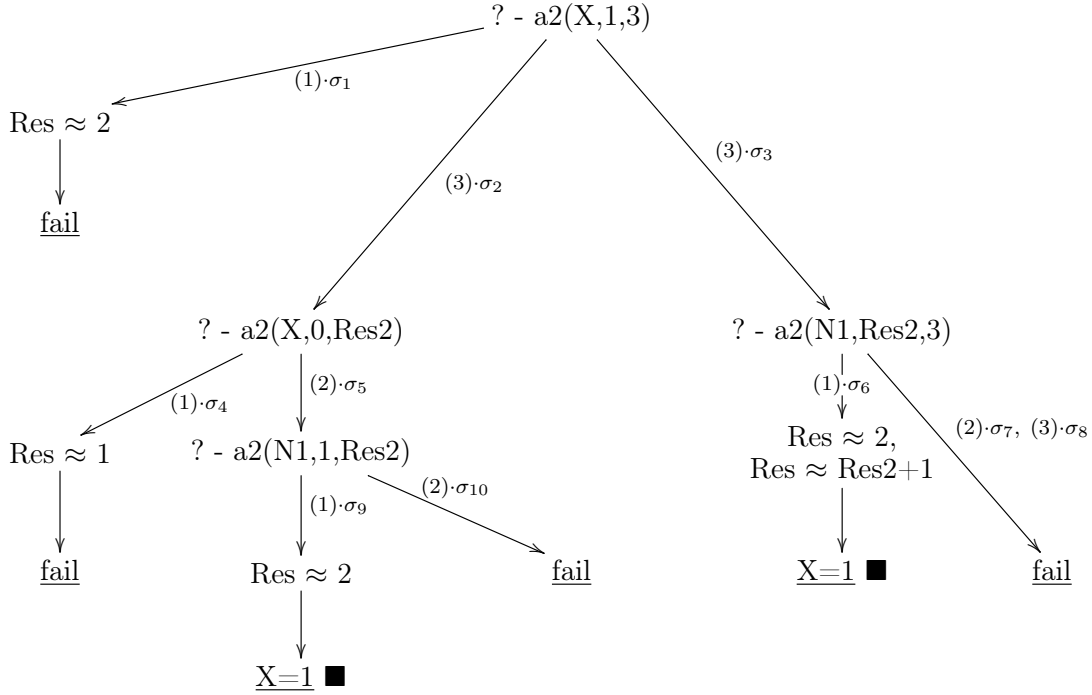


Рисунок 4.4: Дерево вывода для предиката Аккермана

Рассмотрев пример из рисунка 4.5, замечаем, что стратегия вычисления может кардинально измениться, если в декларативной парадигме использовать механизм, который позволит отсекать и менять путь расследования, по крайней мере, по каждому уровню подцели. Например, если приостановить логический вывод на примере из рисунка 4.4 после подцели « $? - a2(N1,1,Res2)$ » и мы уверены в этом, то всю ветку  $(3) \cdot \sigma_3$  можно не проверять. Для этого мы вводим определение отсеечения.

**Определение 4.4** (Отсечение решений). *Отсечение решений приостанавливает локальный поиск альтернатив при обнаружении первого пути к решению. Локальность применяется ко всем подделям внутри рассматриваемого предиката до обнаружения оператора отсеечения «!».*

Отсечение не означает, что любой алгоритм станет эффективнее или *лучше*, это вовсе не так. Отсечение даёт лишь возможность, в зависимости от предложенного описания решения проблемы, ускорять нахождение предложенного решения, либо исключать дубликатные решения. Отсечение вставляется в тело предиката как подцель. Рассмотрим теперь семантику «!» на примере факториала в предикате **fact** из рисунка 4.6.

$$\begin{aligned}
\sigma_1 &= X \approx 0, M \approx 1, \text{Res} \approx 3 \\
\sigma_2 &= N \approx X, M \approx 1, \text{Res} \approx 3 \\
\sigma_3 &= \text{Res} \approx 3, N \approx X, M \approx 1, N1 \approx X-1, M1 \approx 0 \\
\sigma_4 &= X \approx 0, \text{Res} \approx \text{Res2}, M \approx 0 \\
\sigma_5 &= N \approx X, M \approx 0, \text{Res} \approx \text{Res2}, N1 \approx X-1 \\
\sigma_6 &= N1 \approx 0, M \approx \text{Res2}, \text{Res} \approx 3 \\
\sigma_7 &= N \approx N1, \text{Res2} \approx 0, \text{Res} \approx 3, N1 \approx N1-1 \\
\sigma_8 &= N1 \approx N, M \approx \text{Res2}, \text{Res} \approx 3, N1 \approx N1-1 \\
\sigma_9 &= N1 \approx 0, M \approx 1, \text{Res} \approx \text{Res2} \\
\sigma_{10} &= N \approx N1, M \approx 1, \text{Res} \approx \text{Res2}, N1 \approx N1-1
\end{aligned}$$

Рисунок 4.5: Сопоставления для дерева вывода из рисунка 4.4

```

fact(0,1).
fact(N,Res):-N1 is N-1,fact(N1,Res2),Res is n*Res2.

fact2(0,1):-!.
fact2(N,Res):-N1 is N-1,fact2(N1,Res2),Res is n*Res2.

```

Рисунок 4.6: Пример код факториал на Прологе

Первый терм представляет входное целое число, второй терм представляет результат факториала. Предикат представляет соотношение между входным и выходным вектором, но в данном примере определён только порядок *вход-на-выход*. Предикат **fact2** почти ничем не отличается от первого предиката, разве что, в базисном случае имеется отсечение как единственная подцель бывшего факта. Отсечение приводит к тому, что первый альтернатив **fact2** только тогда вызывается, когда входной и выходной терм унифицируются с правилом при запросе согласно опр.4.3, например, «?-fact(5,R).» — это происходит лишь один раз при индуктивно-определённом спуске, когда входное число равно нулю. Когда эта ситуация возникает, отсечение гарантирует, что во всех обработанных подделях альтернативы далее не обыскиваются. При рекурсивном подъёме альтернативы дальше не рассматриваются. Если при рекурсивном подъёме нет совпадающей подцели, то альтернативные правила должны рассматриваться, кроме отсечённых альтернатив. Поэтому, отсечение в примере **fact2** не лишнее. Оно улучшает понятие вычисления подцелей и ее оптимизацию. Обратим также внимание на то, что порядок правил сильно влияет на корректность. Например, поменяв порядок правил или отсечения, см. рисунок 4.7.

В этом примере базисный случай не достижим потому, что 0 и 1 также унифицируются в первом альтернативе. Отсечение приводит к тому, что второй альтернатив не будет рассмотрен. Далее N1

```
wrong_fact2(N,Res):-!,N1 is N-1,wrong_fact2(N1,Res2),Res is n*Res2.
wrong_fact2(0,1).
```

Рисунок 4.7: Контр-пример код факториал на Прологе

присваивается -1 при вызове «?-wrong\_fact2(0,1)». В итоге, имеется левая рекурсия, которая никогда не остановится. В данном примере отсечение даже не важно, т.к. левая рекурсия не разрешает рассмотрение второго альтернатива, хотя все логические факты и правила явно определены. Дерево вывода имеет ветвь, которая бесконечна. В Прологе эта проблема лучше известна, как проблема приоризации. Поэтому, ради читаемости правила должны определяться не по любому порядку, а строго по порядку базисного определения: частные и специализированные правила и факты должны определяться первыми, иначе, они могут быть поглощены более общими случаями.

С другой стороны, не поглощённые правила означают, что правила альтернативы могут меняться по позициям. Ради простоты программы, предпочтение всегда должно быть за простыми правилами и одновременно без дополнительных условий касательно порядка. Для достижения этого, необходимо, насколько это возможно, высчитывать входные вектора (см. позже касательно высчитывания куч).

Нужно отметить, что отсечение является синтаксическим сахаром, потому, что отсечение всегда возможно удалить из предиката (см. дискуссию из главы 1, переписав его). Доказать корректность исключения отсечения возможно средством дополнительного аргумента предиката, который запоминает достижимость результата. Если результат достижим, то имеется базисный случай, который должен возникать только один раз. Во всех остальных случаях, имеется строго возрастающая цепочка. Поиск результата может быть обобщён и выражен как поиск плавающей точки для  $\mu$ -выражения. В логиках, в которых допускается отсечение, соблюдается правило (CUT):

$$(CUT) \frac{\Gamma \vdash \Phi, \Omega \quad \Phi, \Delta \vdash \Lambda}{\Gamma, \Delta \vdash \Omega, \Lambda}$$

В контексте Пролога верхние консеквенты формой  $A \vdash B$  могут рассматриваться как  $A$  подцель тела правила, а  $B$  голова предиката, т.е. логический консеквент (следствие). Для дальнейшего обсуждения не будут использоваться консеквенты. Отсечение, возможно, рассматривать, как попытку *детерминации*. При детерминации множество возможностей отсеивается, так именно и происходит при «!». Отсечение не гарантирует терминирования (см. ранее упомянутый пример) и полной детерминации, потому, что всё равно могут возникать альтернативные ответвления, например последующих «!» подцелях. Когда отсечение происходит намеренно и не угрожает потерям годных решений, то речь идет о «*зелёных отсечениях*» дубликатных решений, в противном случае о «*красных отсечениях*».

Обсуждённые решения, ограничения, моделирование правил и подцелей будут применены позже, в частности в главе 6. В связи с вопросом об отрицании, необходимо обсудить вопрос представления

литералов, т.е. утверждений и отрицание утверждений (если будет необходимо для определения выразимости утверждений о кучах). Когда речь идёт об отрицании, необходимо рассмотреть выражения, которые строятся термами на основании опр.4.1. Отрицание переменной может создать практическую проблему, т.к. оно определяется контекстом. В отличие от единственного случая, отрицание множества (бесконечного, но индуктивно-определённого), в общем случае не решимо. Нужно отметить, что метод представленный в главе 6 — решим, если даже допускаются индуктивные определения, благодаря обсуждённым свойствам, в частности, из-за *неповторимости* и локальности куч. Очевидно возникает похожая проблема нерешимости для предикатов. Ситуация для функторов в общем однозначно не определена: если дана функция, которая отображает из домена в определённую ко-область, тогда, каким образом можно универсально определить отрицание? — Это лишь гипотетичный вопрос потому, что отрицание по определению применимо только к булевым множествам. Следовательно, необходимо определить отрицание для предикатов в Прологе.

Отрицание в Прологе определяется как провал, т.к. Пролог ищет решение и должен его найти только отрицательным. Поэтому, в Прологе согласно обработке (см. следующий раздел) вводится на уровне тела встроенная зарезервированная подцель `fail`, которая сигнализирует прекращение поиска альтернатив и возврат к предикату вызова. В общих случаях уговаривается явное прологовское определение отрицания предикатов.

## 4.2 Логический вывод как поиск доказательства

С помощью Пролога было продемонстрировано множество применений в области обработки языков, формальных и естественных [201], [168], [285]. Как будет показано позже, слово данного языка необходимо сгенерировать и проверять согласно правилам. Естественные языки характеризуются мутациями [201], [131] в связи с культурно-социологическим развитием. Таким образом, они сильно отличаются от формальных и формализованных языков. Многозначность является основной разницей между естественными и формальными языками. Тем не менее, изучены методы распознавания естественных предложений [131],[200], которые приводят к интересным итогам в связи с вопросами искусственного интеллекта и обработки человеческой речи, в общем. Однако, данная работа, сосредоточена на сравнительно простом уровне лексики и толкования. Задача будет заключаться в нахождении максимально простой и адекватной модели динамической памяти и её проверки. Для реализации прототипа, а также для исследования ключевых задач, будут рассматриваться правила Хорна. Предложенный Уорреном подход «*использование предикатной логики для решения логических задач*» воплощается частично в жизнь, хотя Уоррен сам предупреждает о том, что его реализация Пролога не в состоянии полностью отобразить логическую «*мощь*» предикатной [144] логики, из-за теоретических ограничений в связи с вычислимостью. Простой пример проиллюстрирует одну фундаментальную сложность — допустим, имеется функция, где входной и выходной

вектор записывается как предикат, т.к. это может быть реализовано в Прологе. Тогда, возможно, для данного выходного вектора, не решимо или очень трудно решить обратный результат, например в случае разовых функций для шифровки сообщений, либо функции, которые не полностью определены (в обе стороны) и имеют, например, двухмерное отображение в качестве эллипса. [200] расследует подробно методы перебора, но, увы, теоретические ограничения синтаксического анализа не принимаются (полностью) к сведению, поэтому перебор оказывается сильно ограничен, в связи с распознавателями категории LL(1) или слабо расширенные от него анализаторы. Это очень ограничивает выразимость в целом. [298],[307] показывают, что если речь идёт о генерации и проверке данных (в статье представлены исключительно как термы), то:

1. Синтаксический анализ является непосредственно ограничивающим фактором выразимости. Если выражения расширены и искусственные ограничения, а также упомянутых, а также и существующих подходов исключаются, то синтаксис и семантика всего процесса проверки могут быть сильно упрощены без потери общности.
2. Упрощение также может быть достигнуто ради обобщённой формы промежуточного представления. Однако, в статье речь не идёт об императивных языках.
3. В частности, из-за выбранной в статье функциональной парадигмы для сравнения, возникают проблемы с незначительно раздутым описанием. Отметим, что функциональное описание состояния вычисления всё равно удобнее и проще любого императивного представления. В общем используются лишь 5 функционалов высшего порядка, как например, `concatMap`, `foldl`. Если бы использовалось логическое представление, то можно было бы семантику процесса проверки более близко сформулировать в стиле математико-логических термов, а это более важно в этой работе.

[292],[293], [299], [298] и [307] показывают, что на первый взгляд нестандартный путь, может привести к очень простому, но эффективному решению проблемы проверки с помощью термов. Причина разносторонняя: представление трансформации термов в виде правил и компактность (проводилась широкая экспертиза по отношению к иным функциональным языкам, т.к. императивные языки сильно ухудшают все показательные метрики измерения качества и компактности записи). Причина простоты лежит в сравнении данной программы и её описания, которая вписывается в *регулярное выражение* в более обобщённом виде, даже в контекст-зависимое выражение, в частных случаях, как было продемонстрировано на примерах вызовов процедур. Проводятся качественные анализы по каждому из входных операторов и количественные сравнения, основанные на метриках. Полученный результат неожиданный потому, что почти по всем сравниваемым параметрам представление и решение в Прологе торжествует к сравнению с функциональным представлением [292]. Далее результаты обсуждаются в разделе 4.3.

**Тезис 4.5** (Доказательство куч равно синтаксическому перебору). *Логическое программирование динамической памяти как доказательство равнозначно синтаксическому перебору.*

Предпосылкой этому тезису является то, что метод(-ы) синтаксического перебора позволяет решить проблемы доказательства динамических куч. Как это решить реально, какие для этого имеются условия и почему должны соблюдаться, объяснения даются в этой работе.

Позже вводится модифицированный диалект Пролога, который способствует к решению целого ряда проблем в связи с автоматизацией доказательства теорем (ср. главу 1). Таким образом, язык похожий на Пролог, квалифицируется не только как язык, а как теоретический метод и инструмент к целому ряду проблем. Особенность Пролога заключается в том, что доказуемо только то, что выводимо (см. набл.1.13). Если что-либо не выводимо, то причина может лежать в правилах, в представлении правил (например, в абстракции), в ограниченности основной выводимости, а также в формулировке подцелей. Также могут иметься практические ограничения со стороны синтаксического анализа.

Идея использовать синтаксический анализ, в качестве решателя куч, появилась после некоторых наблюдений (см. главу 6):

1. Прологовские правила довольно компактны и очень хорошо приспособлены к решению логических задач.
2. Программа Пролога представляется правилами. Вывод правил генерирует дерево. Доказательство теорем имеет структуру аналогично программе (изоморфизм обсуждённый ранее), это же относится к запросу или запуску программ, которые тоже генерируют дерево.
3. При необходимости выражения можно синтаксически анализировать. Локальность ссылочных моделей памяти ограничивает выразимость соответствующих генерируемых грамматик так, что ранг класса формальной грамматики не должен превышать ранг контекст-свободных грамматик. Это очень полезно с практической точки зрения.
4. Однажды, на одном мало интересном докладе пропагандист «*Semantic Web*» пробовал рекламировать свой новый подход, как подход будущего, как минимум в следующие 25 лет. В качестве защиты использовался аргумент, что «*Semantic Web*» решит почти все актуальные проблемы, технические и научные. Также упоминалось, что «*Semantic Web*» высшие по абстракции и устаревшие методы, как например, синтаксические, давным-давно решены и не имеют будущего. Как я сегодня понимаю и думаю — докладчик немного преувеличил, если даже считать, что синтаксический перебор можно действительно рассматривать как очень хорошо изученным. Несмотря на это, всё равно синтаксические методы могут пригодиться и сегодня, а также помочь решить реальные проблемы в области автоматизации доказательств.

Теперь необходимо заметить и сравнить цели дисциплин:

- |                                |                                      |
|--------------------------------|--------------------------------------|
| 1. Программирование            | нахождение решения                   |
| 2. Логическое программирование | нахождение (логического) решения     |
| 3. Синтаксический анализ       | нахождение (синтаксического) решения |

Если отображения пунктов 2 и 3 типизировать, то получаем характеристики из рисунка 4.8.

Логическое программирование:	
Входные данные:	прологовские правила, запрос
Выходные данные:	ответ «Да/Нет», дерево вывода и все сопоставления
к сравнению имеется:	
Синтаксический анализ:	
Входные данные:	формальная грамматика, входное слово
Выходные данные:	ответ «Да/Нет», дерево вывода

Рисунок 4.8: Характеристики типизации из пунктов 2,3

То есть, если удастся сблизить входные множества, то решение одним подходом может быть будет сопоставлено решению другим подходом.

Упомянутые ранее подходы для распознавания естественных языков отличаются от формальных языков тем, что в них имеется некоторая степень неверности или отклонений. В рассматриваемых языках, отклонения практически отсутствуют.

**Определение 4.6** (Двойная семантика предикатов Пролога). *Предикат в Прологе имеет двойную семантику: (1) семантика по вызовам и (2) интерпретация предиката как соотношение.*

Согласно опр.4.6. Семантика (1) по вызовам (*декларативное свойство предиката*) означает, что предикат рассматривается строго как процедура со стеком, которую можно выразить. То есть, создаётся стековое окно (см. рисунок 4.10) при вызове предиката, а после вызова окно утилизируется. Аналогично *параметрам по ссылкам* в императивных языках программирования, унифицируемые термы, которые используются в различных слоях предикатов (т.е. которые расположены в последовательных стековых окнах – нарушение последовательности без глобальных хранилищ приводит к потере термов) передаются вызывающим предикатам. В отличие от императивных языков программирования, переменный символ может передаваться не только непосредственно, но а также как частица сложного унифицируемого терма. Это означает, что передача из одного слоя предиката в другой по умолчанию, может практически привести к построению сложного объектного экземпляра, за нулевую стоимость и преобразовать довольно сложный процесс трансформации объектов, что очень замечательно. Обратим внимание, что благодаря анонимному оператору «\_», встроенному предикату `concat` и другим предикатам высшего порядка, нетрудно создать мощную выразимость.

Выразимость резко повышается ради простоты выражения и сравнения данных частей функтора и структур в общем (включая списки), потому, что исключается необходимость определять весь перечень элементов списка, но, в отличие от сопоставления с образцами, остаток не теряется, а лишь остаётся без изменений, что упрощает описание простых операций. Напомним, что верификация куч происходит пошагово после каждого программного оператора, обычно меняется только маленькая часть всех куч.

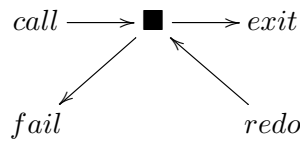


Рисунок 4.9: Вызов подцелей

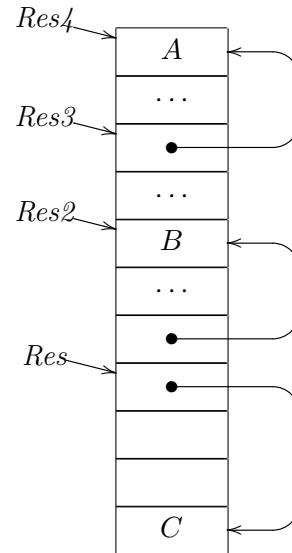


Рисунок 4.10: Пример стековых окон при вызове подцелей

Вызов предикатов из рисунка 4.9 осуществляется согласно модели «чёрного ящика», который можно отследить в Прологе, включив трассировку. Вызов предиката, т.е. подцели, обрабатывается независимо от содержимого предиката и имеет четыре возможных результата: (а) *call*, вызов, создаётся стековое окно, передаются присвоенные термы, (б) *exit*, возврат при успехе, когда все подцели тела выполнимы и голова предиката унифицируема, (в) *fail*, провал, когда хотя бы одна подцель невыполнима, или (г) *redo*, повторная попытка, когда (в) происходит в одной подцели, то находится альтернатив данному предикату, если он существует. Опасность не аккуратно сформулированных предикатов является не эффективной из-за широкого перебора альтернатив. Другой экстремальный случай возникает, когда применяется отсечение настолько часто, что годные результаты исчезают. Оба случая необходимо устранять. Предикаты *call* и *fail* из рисунка 4.9 действительно существуют в «GNU» Прологе [83], пятый случай «исключения» удалён из-за причин, обсуждённых в главе 1.

**Наблюдение 4.7** (Равенство о единицах доказательств). *Поиск доказательства является поиском решения подцели. Обе структуры поиска, доказательство теорем о кучах и запрос подцели, являются направленным деревом поиска.*



Семантика (2) интерпретаций предикатов подразумевает, что предикат является генератором недетерминированности. Общий предикат может иметь (как обсуждено в главах 1 и 6) а) различные определения для различных комбинаций входных и выходных векторов, а также б) различные выводимые результаты для одной подцели. Например, если имеется предикат `mortal` из рисунка 4.1 (а) и к нему ещё добавить несколько фактов, то согласно б) из подцели «?-`mortal(X)`» выводимы следующие результаты: `X=socrates`, `X=plato` и т.д. Если для функции Аккерманна вызывать «?-`acker(0,X,15)`» или «?-`acker(1,1,Res)`», то в зависимости от реализации, предикат, либо определён (но, возможно, очень долго занят или не подлежит практическому вычислению в связи с ограничением памяти), либо не (полностью) определён, как, например, на рисунке 4.1 (б). В примере для данного выходного `Res` предикат не определяет входной вектор, хотя математико-логическое определение явно существует согласно данной схеме, которая практически без изменений стоит в данных прологовских правилах (см. опр.1.5,набл.1.13). Дискуссия касательно этого ограничения, также в связи с обратимостью и представлением реляционной модели, продолжится позже в данной главе.

Как уже упоминалось, Пролог не является каким-то «*универсальным решением всех задаваемых логических проблем*» априори [266],[267]. Это правда, что не всякое доказательство можно решить прологовскими подцелями в связи с ранее обсуждёнными фундаментальными ограничениями. Однако, обратное всегда возможно. Задача заключается в минимизации проблем, которые будут лучше решаться в связи с сближением языков. Подмножество языков Пролога используется для решения целого ряда проблем верификации в связи с представлением и методом логического вывода. Охарактеризуем подцели и аргументы термы:

- Встроенные предикаты, как например `concat`, могут быть использованы везде в программах. Кроме того, имеются предикаты, которые могут добавляться в формальные теории, написанные в Прологе. Зарезервированные предикаты и функторы могут быть определены мультипарадигмально.
- Отдельное и перегруженное значение. Отдельные предикаты могут быть перегружены так, что предикат с подходящей арностью, который определен ранее, используется первым. Этот механизм позволяет подключение прологовских теорий, в том числе *мульти-парадигмальность*. Полученный эффект, это расширяемость, вариабельность, гибкость и полиморфизм. Кроме того, перегруженные предикаты, как например `concat`, позволяют гибкий порядок сопоставлений входных и выходных термов.
- Операции над кучами являются термами. Вместе с объектными экземплярами, соотношения местоположения куч может быть представлено термами. По сути, граф кучи представляется полностью термом кучи. Термы являются аргументами подцелей, т.е. (под-)доказательств теорем и лемм о кучах.

Из универсального описания механизма вызовов из рисунка 4.9 следует *поиск с возвратом* (с англ. «*backtracking*»). Эта стратегия просто реализуема рекурсией над предикатами, и точно описывает деревья логического вывода (см. контекст-свободность прологовских правил с главой 6). Описанные ранее ограничения необходимо обязательно учесть. Когда поиск завершается успехом, то ветка считается доказанной. Когда ветка опровержима, то она, либо не доказана или не доказуема в принципе, либо приводит к явному противоречию. Оба варианта достаточны для установления провала (под-)доказательства, т.к. всегда необходимо доказывать универсальность данной формулы. Если интерпретировать предикаты как декларативные, то предикаты можно с легкостью растолковывать как «*процедуры*» в императивных языках программирования, т.е. если даже это формально не совсем совпадает, то интуиция совпадает с подходом в решении задачи. Следовательно, можно в действительности рассматривать логическое программирование в Прологе максимально приближено к доказательству. Хотя специфицируются и доказываются свойства о программе, входная программа при этом не отлаживается и не запускается. Как будет показано в более поздних главах, особенность Пролога хорошо распространяется именно для решения постановки проблем динамической памяти. Представление модели куч, абстракции правил, рекурсивные предикаты, логические символы в формулах в итоге увеличивают выразимость. Например, в отличие от обобщённых формул логики предикатов первого порядка, нам удастся найти компромисс таким образом, чтобы построение графа кучи проводилось последовательно. При этом, описание графа проводится *снизу-вверх*, используя отдельные простые кучи. Логическое представление выражений в формулах разрешает широкую гибкость самих куч, но а также при реализации стратегий над кучами, в том числе и определения правил вывода.

**Наблюдение 4.8** (Дедукция с возвратом в доказательствах). *Поиск с возвратом в Прологе симулирует ход формального доказательства.*

Структура доказательства является деревом, если получается цикл внутри доказательства, тогда получается предыдущее состояние вычисления, которое означает ввод регрессии. Регрессия является индикатором того, что применённое правило не привело к новому состоянию прогресса, т.е. все следующие регрессии состояния подлежат к удалению. Ненужные состояния могли появиться только из-за неверных правил. Если доказательство правил завершается успешно, то и соответствующая подцель доказана, т.к. предикаты в обоих случаях не отличаются (кроме возможно необходимых переименований предикатов и используемых символов). Поиск Пролога опирается исключительно на дедукции (ср. главу 1): выводимо только то, что следует из фактов и правил, иные выводы, например *модус толленс* недопустимы. На практике это означает, что фальсификация предикатов может быть решимой в общности только, если рассматриваемые множества конечны (например, вводя общие символы для определения класса объектов) и полностью определены. То есть, исключаются случаи, которые могли бы сами себе противоречить. Следовательно, доказательства могут быть раз-

дутыми или классической дедукцией просто недоказуемыми (см. пример исключённого третьего из 1). Предикаты могут быть определены конечным образом, хотя входное множество бесконечное. Тогда, отсечение (см. опр.4.4), как провал, может представлять собой отсечение предиката, либо иное явное определение. Идея поиска с возвратом при провале заключается в *продолжении* следующего неверного предиката по родительской оси за счёт минимальной записи с тактикой переключения стека. В отличие от метода «возврат при провале», метод «возврат с направленным фокусированием» можно считать обобщённым, когда в самом простом случае фокус определяется некоторой целой отметкой и тактика переключения ориентируется на нахождение экстремума этой отметки. Отметка определяет, какая подцель будет доказываться следующей. Если внимательно задуматься, то такого рода задача может быть редуцирована без потерь в общую стратегию вычисления «ветвей и границ» [179]. Обобщением отметки является терм, который также отлично вписывается в головы левых сторон правил с помощью сопоставлений образцами.

**Наблюдение 4.9** (Стековая система вызовов). *(Декларативная) семантика машины Пролога основательно отличается от императивных языков программирования связями верхних зависимостей в термы, а также переключением стека между состояниями вычислений, для которых используется стратегия «поиск с возвратом».*

Тройку Хора в Прологе можно представлять в качестве предиката с минимальной арностью «3», при этом, первые две компоненты записываются как входные термы, а третья — в качестве выходного терма. Конечно, в общем, предикат может быть обратим, но в данный момент мы не рассматриваем вопрос, что же является объяснением данному следствию и правилу (см. раздел 4.3 и следующие). Причины зависимостей в дальнейшем подчёркиваются.

### Обсуждение декларативных стратегий основанных на стеке

- **Быстродействие.** В частности, речь идёт об улучшении производительности «реактивных систем» (систем под наблюдением и контролем другой системы). Хотя Уоррен [266] подчёркивает, что быстродействие не решающий фактор, однако, медлительность может иметь причину в самом алгоритме (проблема определения правил), например в копировании часто используемых сегментах стековых окон и построении сложных объектов из более простых. Имеются две причины: во-первых, часто копировать, т.е. использовать подвыражения — показывает на (очень) высокую зависимость данных, чего в общем можно избежать с помощью улучшения алгоритма и определения правил. Во-вторых, изменения обработки с поддержкой быстродействия извне (т.е. мульти-парадигмальной) адаптацией можно при таком подходе существенно ускорить медленные точки, несмотря на дополнительные затраты в связи с загрузкой. То есть, вторая проблема разрешима с помощью вариабельности. Это означает, что предикат может быть заменён иным поведением, но одинаковой спецификацией. Языковые расширения и вари-

абельность, тоже входят в поле рассмотрения, но уже в разделе 4.3, поэтому, их всегда стоит рассматривать, независимо от данных пунктов.

- **Порядок вычисления.** Бесконечные списки (как например **take** из главы 1) согласно данной операционной семантике [267] полностью исключаются. Операционная семантика полностью помещает список в стек, поэтому, частичное исследование для бесконечных списков отпадает. Частичное помещение в стеке всё равно, но только, если сравниваемый терм уже имеется в стеке, а разница между подвыражениями двух термов минимизируется. Изменение операционной семантики возможно, но требует полного изменения семантики, хотя порядок изменения и эффект вычисления сам по себе маленький. Надо отметить, что если прологовские правила унифицируются, то сравниваются разницы между термами (точнее кучами, см. главу 6). Естественным дополнением было бы вычисление снаружи внутрь (см. главу 1) термов параметров, но такое требование можно избежать, благодаря ранее не полностью вычисленным символам (под-)термов и не требует никаких изменений или дополнительных затрат, без ограничения общности. Единственный, возможно критический этап, касается обратимости предикатов. Следовательно, единственным фактором изменения порядка вычисления является само определение правил.
- **Семантический контекст.** Предикаты могут быть перегружены или определены мультипарадигмально. В обоих случаях проблема зависит исключительно от конкретного данного определения правил.
- **IR.** Промежуточное термовое представление может в ходе разработки оказаться устаревшим или несовместимым требованием к императивной программе. В этот момент необходимо иметь возможность добавлять и менять существующие термы (в том числе удаление). Поэтому вариабельность и расширяемость считаются ключевыми свойствами, которые Пролог допускает, в чём можно легко убедиться. Например, если ради локальной оптимизации и быстрого доступа к памяти, необходимо ввести В-деревья [72],[18], то это возможно с помощью индексации встроенных и/или скрытых предикатов, при этом, реализация предиката может по различным причинам проводиться на другом языке программирования.

Не использовать стек, не удастся, т.к. языки полностью свободные от стека и основаны только на рекурсивных схемах, на практике оказались совершенно не пригодными, тем более несколько не приспособлены к модуляризации и следовательно не рассматриваются далее по практическим причинам. Современные ЭВМ настолько оптимизированы, что процессоры в состоянии ускорить выделение и утилизацию стековых окон практически без больших затрат [123],[215]. Описывающая парадигма состояния вычисления кучи должна быть логической для решения её проблем верификации.

## Обсуждение поиска доказательства над реляциями

Предикаты связывают термы, которые состоят из символов, переменных и других термов, в любое определённое соотношение. Не трудно убедиться в том, что количество отображений между  $A$  входных и  $B$  выходных термов равно  $B^A$ . Множество отображений могут быть объединены в одном предикате. Вопрос о том, определено ли и решимо ли данное отображение для данного входного/выходного/смешанного вектора является отдельным и сложным вопросом (смешанный вектор ради простоты не рассматривается). Эта проблема может быть редуцирована на проверку выполнимости логической формулы. Естественно, с практической точки зрения только разработчик несёт ответственность за перегруженные по количеству отображений предикаты. Чем больше количество отображений одного предиката, тем выше его применимость. Для перегруженных предикатов вопрос об обратимости становится всё важнее, т.к. не совершение автоматически приведёт к нетерминации, а, следовательно, к не завершению доказательства. Позже это надо обязательно учесть. С точки зрения канторовых множеств предикат является перечисляемым множеством (возможно очень большое, но дискретное), которое объединяет элементы доменов с элементами ко-области — т.е. похожая концепция. С практической точки зрения это не маловажный универсальный вопрос. Если нам удастся показать универсальные свойства *реляционной алгебры по Кодду*, то мы её сможем симитировать. Это фундаментальное свойство, которое позволит достичь уровня выразимости, хотя бы реляционных и объектно-реляционных языков баз данных на практике, что уже покрывает огромную и значимую часть выразимости.

**Тезис 4.10** (Выразимость реляций в Прологе). *Если рассматривать правила Пролога как реляции, то зависимости между ними рассматриваются как конъюнкции и выбор. Любая дополнительная унификация как подцель урезает универсальность условия, т.е. проводит, таким образом, селекцию.*

*Доказательство.* Полное доказательство тому содержится в моей работе [292]. Доказательство простое и прямое: свойства Коддской алгебры, как например — проекция, связывание и т.д., можно заменить в правилах Пролога один к одному на каноническую запись без коллизий (см. рисунок 4.11).

Также обратно доказательство производится. При этом могут потребоваться шаги канонизации и удаления отсечений, что в общем случае можно всегда совершить, как это было ранее продемонстрировано. □

Здесь нельзя не заметить, что например [149] задавался частично проблемами более эффективного представления входных и выходных данных. Итог расследования приводит к тому, что удобное представление может быть дано логическим видом, но не функциональным. Работа [205] расследует

Объединение реляций:

$$T = R \cup S: \quad \begin{aligned} t(x_1, \dots, x_m) &: -r(x_1, \dots, x_m). \\ t(y_1, \dots, y_n) &: -s(y_1, \dots, y_n). \end{aligned}$$

Множественный минус:

$$T = R/S: \quad t(x_1, \dots, x_n) : -r(x_1, \dots, x_n), \text{not}(s(x_1, \dots, x_n)).$$

Картезианский продукт:

$$T = R \times S: \quad t(x_1, \dots, x_m, y_1, \dots, y_n) : -r(x_1, \dots, x_m), s(y_1, \dots, y_n).$$

Проекция:

$$T = \Pi_S(R): \quad \begin{aligned} t(s_1, \dots, s_n) &: -r(x_1, \dots, x_m). \\ \forall s_i \in \{x_1, \dots, x_m\} \end{aligned}$$

Выбор/Селекция:

$$T = \sigma_S(R): \quad t(x_1, \dots, x_n) : -r(x_1, \dots, x_n), s(x_1, \dots, x_n).$$

Переименование:

$$T = \rho_S(R): \quad t(x_1, \dots, x_n) : -r(x_1, \dots, x_n).$$

Рисунок 4.11: Реляционная модель применена к предикатам Пролога

фундаментальные свойства доменов как реляции. Она посвящается большей частью доказательству существования инварианта реляций. Использование инварианта остаётся интересным и частично открытым вопросом в связи с улучшением кэширования куч в спецификациях (см. главу 5). Примеры, где инварианты реляций пока что имели наиболее широкое поле применения, это «*нумеральная логика*» [204], [49].

После того, как мы ввели реляции и обосновали их свойства, сразу наблюдаются приятные свойства:

- **Декларативность.** При логическом выводе, арифметические вычисления не столь важны. Важны объекты, т.е. кучи и их взаимосвязи. Это не только вопрос «вкуса», но особенно является фундаментальным свойством именно логической системы. Пролог представляет собой среду представления и обработки знаний. Кучи являются атомами или сложными термами, а правила предикатами куч. Псевдонимы являются символьными переменными, которые используются в других местах или более того. Этого достаточно, чтобы определить теории куч.
- **Терм-Дерево.** По *теореме Биркгоффа* (о термовых продуктах) [202],[79] из абстрактной алгебры следует, что каждый терм корреспондирует с представлением в виде дерева. Преобразование, увы, не обязательно однозначное, если не проводить нормализацию. Таким образом, обратное преобразование однозначно определено. Отсюда сразу возникает необходимость, либо определить канонизацию для выравнивания деревьев, либо устранить, например ассоциатив-

ность, по которой выравнивание деревьев было бы дано неявно (см. главу 5). Мы не хотим сами себя ограничивать в том, что кучи могли бы быть только деревьями. Мы хотим, чтобы куча могла быть любым графом (см. главу 3), поэтому допускается описывать кучу только одной вершиной. При этом, всё равно, вершину представляет простая или сложная куча, т.е. она является обыкновенной кучей или объектным экземпляром.

- **Генерация термов.** Тематическое исследование [292] показывает на то, что Пролог отлично приспособлен для обработки термовых структур, в отличие от функциональных/императивных языков программирования и трансформации. В исследовании использовалось большое количество примеров. Также проводился количественный анализ. Использовались метрики как компактность, уровень выразимости и интеллектуального уровня языка и многое другое. Для широкого объёма примеров, практически без всяких исключений, Пролог превосходит чётко с большим отрывом.
- **Проверка термов.** Тематические исследования [298],[307] показывают, что если процессы генерации и проверки термов сблизить, то основным твёрдым ограничением является выразимость языка проверки. В исследовании, без потерь общности, рассматривается обобщённый регулярный язык. Процесс сравнения термов слабо структурируемых данных можно интуитивно понять, либо как сравнение одинарных элементов, либо иерархических элементов с возможными дырами, которые наполняются данными во время запуска. Терм, как обобщённое представление, является уникальным IR и может быть широко использовано в различных областях, например для верификации. Операторы описывают не программу, а структуру генерируемого документа. Статически, цикл не всегда может быть ограничен, поэтому цикл вершины  $a$  некоторого дерева в XML-документе описывает лишь  $a^*$ . Отсюда ясно, условия цикла могут быть представлены и проверены только самым общим видом. Естественно, конечным автоматом так и не удастся распознать  $a^n b^n$ , но это и не главная цель исследования. Выходит, что главным ограничением проверки всегда является выразимость языка утверждений (выражений). Далее, главным результатом вместе с [292] является то, что реляции лучше приспособлены для представления знаний с термами и логическими правилами трансформации, чем функции, которые вычисляют для каждой «дыры» необходимые данные. Получается, логические соотношения ссылаются на имеющиеся компоненты. В [292] под логическими правилами трансформаций в основном подразумеваются  $\tau \rightarrow \sigma \rightarrow \tau$ , где  $\tau$  представляет некоторый терм IR, а  $\sigma$  среда, содержащая символьные присваивания. Если эту трансформацию расценивать как реляцию в качестве предиката, то трансформацию можно расширить как  $\tau \rightarrow \sigma \rightarrow \tau \rightarrow \mathbb{B}$ , где  $\mathbb{B}$  булево множество. Теперь верификация куч может быть представлена таким же семейством трансформаций, как и верификация куч. Её главная разница заключается в использовании (абстрактных) предикатов и в методах автоматизированного

вывода. При трансформациях используемые термы — модели куч, различаются. В отличие от регулярных выражений и возможных распознавателей [50], схемы спецификаций в основном контекст-свободные. Предсказания следующего элемента в обоих методах являются одной из центральных операций, которые могут существенно отличаться.

**Тезис 4.11** (Упрощение с помощью термового IR). *Использование (прологовских) термов упрощает спецификацию и верификацию куч.*

Решение и доказательство этому тезису изложено не только в этой главе, но также и в последующих. Кроме терма можно использовать и другие промежуточные представления, как например тетрады, польско-инверсную запись, триады и другие. Преимуществом термов при описании состояний куч, как было упомянуто, в первую очередь является простота и максимальная выразимость. Термовое представление программных операторов может быть записано в Прологе непосредственно. Преобразование в другие IR отпадает [291], [181]. Однако, термовое представление в Прологе имеет то преимущество, что все термы и подтермы не нуждаются в дополнительных контекстах, конвенциях и дополнительных фазах преобразований. — Их *«можно написать просто так»*. Это не только облегчает возможное использование в учебных целях, в быстрой прототипизации, но, а также облегчает преобразование и переписывание термов за счет прямого представления и анализа правил переписывания (см. [19],[87]). Аналогичные реализации на более реальном уровне являются, например, *«LLVM-биткод»* [253], GCC *«GIMPLE»* [171] или аннотированные объекты в качестве IR в проекте *«ROSE»* [228]. Во всех случаях, которые используют тетрады, необходимо предварительное IR входной программы преобразовать в синтаксическое дерево преобразуемое в тетрады. Синтаксический перебор в *«LLVM»* производится более гибко, чем в *«GCC»* с помощью представленного этапа и может быть совершен с помощью среды синтаксического анализа *«clang»* [251]. Дерево перебора может теоретически быть введено без *«clang»*, но на практике это совершенно немыслимо, потому, что, даже крайне простое дерево, всё равно может и будет представлено очень большим промежуточным представлением, если *«LLVM»* заставить вручную ограничиваться неэффективным и полным отсутствием всех дальнейших трансформаций IR. Несмотря на раздутые наименования и на первый взгляд *«ненужные»* синтаксические определения, гибкость и расширяемость сильно повышены в отличие от *«GIMPLE»*.

Ради ограничений и простоты, в отличие от *«биткод»*, реализация в Прологе исключает безусловные переходы к любому программному оператору. В реализации не стоит приоритет обеспечить максимальный объём программных операторов, если в будущем имеется возможность подключения любых иных программных операторов. Более важным вопросом является расширяемость и вариативность модели кучи: *«можно ли простым образом модифицировать кучу так, чтобы имитировать любую пошаговую манипуляцию кучи?»*, *«Можно ли добавлять всё новые фазы и правила логического вывода?»*.



Принципиально, нужно заметить, что выбранное промежуточное представление естественно может быть преобразовано из Пролога, например в биткод или «*GIMPLE*», но практическая реализация не стоит вопросом исследования. Прологовские термы представляют собой программные операторы (а также спецификацию и правила вывода) и как ранее в этой главе обсуждалось, могут быть представлены в качестве дерева. Да, можно выбрать тетрады инструкций (например, близки ассемблеру некоторой целевой машине), но, первым итогом синтаксического анализа всегда является синтаксическое дерево (см. рисунок 1.12). Выбрать другую модель означает, что одна и более фазы из упомянутых на рисунке 1.12 просто пропускаются. Это простое замечание, но, увы, этот принцип часто (не умышленно) нарушался и нарушается в истории проекта «*GCC*», «*LLVM*», а также в проекте «*jStar*» [193], где верификация проводится на уровне оптимизируемых триад, коротко до и во время генерации кода и многих других проектов в области статических анализаторов — совершенно независимо друг от друга. Хотя замечание простое, последствия могут приводить к необходимости определять точно, в какую фазу анализ псевдонимов всё-таки нужно включать и это обычно является очень непростым вопросом. Важность заключается в следующем: (1) иметь вообще возможность расширять и менять существующие этапы анализа динамической памяти, а (2) возможность адекватного представления для того, чтобы избегать раздутые и сложные семантические контексты и множество экстерных хранителей. Отсутствие возможности №2 является признаком тому, что описание модели сильно усложняется. Простота модели зависит от полного представления всех необходимых данных. Прежде всего, это касается термовых представлений входных программ и при необходимости семантических полей содержавшие данные из рисунка 1.12. Дополнительные данные не упомянуты на рисунке 1.12, но всё равно могут потребоваться на локальных фазах, а следовательно, не вовлекают за собой модификацию общей модели вычислений для работы с динамической памятью (см. рисунок 4.12, см. набл.1.14).

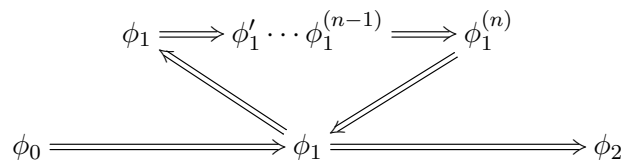


Рисунок 4.12: Архитектура конвейера верификатора и статических анализаторов

Преимуществом термов является их явное представление и явная манипуляция ими. Отсутствие их явного представления приводит к конструкции вспомогательных подтермов и к введению семантических полей (например, для аппроксимации какого бы ни было лимита).

### 4.3 Совместимость языков

В введении было описано, как замечают критики верификации, что часто дисциплина характеризуется «*чисто академической, без практического применения*». Основными причинами тому, являются: перечень конвенций к отдельным моделям, которые часто имеют резкие ограничения при сильно раздутом формализме. Из предыдущего раздела, особенно из тез.4.11, можно заметить сильные обобщения относительно языков спецификации и верификации для проблем динамической памяти (см. главу 2, см. набл.4.12):

**Наблюдение 4.12** (Сравнение декларативных парадигм). *Декларативная парадигма, в первую очередь логическая, для верификации куч лучше приспособлена, чем функциональная и императивная парадигмы в связи с представлением формул и логических выводов.*

Наблюдение на первый взгляд может казаться малозначимым. Однако, оно означает, что для верификации троек Хора удобнее использовать логический язык программирования, спецификации и верификации троек, что на первый взгляд является очевидным. В реальности немногие системы верификации были построены на основе логической парадигмы, поэтому они, слишком ограничены или замкнуты (см. главу 1).

Логические предикаты описывают состояния вычислений, именно поэтому необходимо проверять. Например, предикаты в первую очередь не нуждаются в побочных эффектах, чтобы выразить состояние, а в императивных языках они являются основными. Замысел предикатов заложен в том, что состояние можно было бы определить непосредственно без манипуляции каких бы то ни было глобальных переменных. То есть, диапазон видимости зависит от символьных переменных предиката, но не от экстерных хранителей памяти.

С другой стороны, на примере модификации Пролога, единицы логики могут быть сопоставлены один к одному элементами логического языка программирования. С помощью программирования можно решить вопросы верификации куч.

Не связанное с динамической памятью, но похожее применение, наблюдается в рукописи Леммера [149], в которой предлагается логический аппарат для верификации сходимости программных компонентов [93]. Идея его работы заключается в предложении перехода на логические утверждения для спецификации и верификации, которая опирается на (различную) систему логического вывода из-за целого ряда проблем в связи с объектно-ориентированным программированием, точнее его спецификации.

**Наблюдение 4.13** (Упрощение утверждений равно обобщению). *Если простые определения и утверждения можно задавать более простым образом, то и нахождение решений может оказаться более обобщённым.*

Упрощение определений какой бы то ни было математико-логической проблемы, иногда означа-

ет упрощение или решение проблемы, а, как правило, бывает — наоборот. Часто бывает именно так: чем проще определение, тем меньше существует частных случаев, для которых необходимо вводить отдельные определения. Следовательно, объём годных единиц растёт. Например, сопоставление символам для данного случая  $\forall a$  просто, а соблюдение всё новых конвенций усложняет в принципе. Ограничивая  $a$ , требуется хотя бы один дополнительный предикат. Это означает, использование квантифицированных переменных может резко увеличить выразимость даже маленьких формул. Аналогичное действует обратно: при инвариантной длине формулы несоблюдение утвержденного, приводит к спаду выразимости — это как раз те проблемы, о которых говорилось в этой и предыдущих главах.

Также можно заметить: если выводить предикаты, которые содержат символьные переменные вместо конкретного атома, то следовательно, решение будет более общим (в Прологе это происходит автоматически унификацией термов, а на уровне вывода, как метод нормализованной и финитной резолюции [83]). Задача верификации заключается в проверке программы генерируемой структуры данных. Как было обсуждено в предыдущем разделе, проверка ограничивает сильнее, чем построение структуры. Оба процесса обычно различаются. Сложность проверки заключается в выразимости.

**Заключение 4.14** (Минимизация разницы между языками). *Проверку куч можно упростить и расширить тогда, когда выражения описываются на одном и том же языке во время (1) спецификации, (2) верификации и (3) во входном языке.*

Использование одного и того же языка является экстремумом по задаче минимизации разницы между языками, которое рассматривается до тех пор, пока не будет обнаружен аргумент нарушавший гипотезу, если таковой имеется. Когда ликвидируется разница между (1) и (3), либо будь-то входной язык логический, либо полученное IR в качестве термов из входной программы, то утверждения записываются в качестве подцелей, а входная программа как термы. Утверждения о программе ссылаются на термы входной программы (см. главы 6, 5), обратное не допускается. Так как верное предложение является конгруэнтностью, достаточно ликвидировать разницу между (1) и (2). Формулы и любые необходимые им индуктивные определения задаются прологовскими фактами и правилами. Верификация полностью упирается на факты и правила, которые заданы спецификацией. Кроме того, в качестве тактик, правила спецификации и иных вспомогательных предикатов, предусмотрена возможность включать новые правила в качестве прологовской теории.

Задачи (2) и (3) соперничают следующим образом: (2) устанавливает, как должен выглядеть процесс построения графа куч, а (3) конкретно, исходя только из данных утверждений, пытается доказать верность. Для этого описание должно быть сфокусировано на граф кучи, где процесс верификации является процессом «понимания» и анализа. В главе 6 анализ опирается на синтаксическое определение. Предпосылкой тому являются граф кучи и унификация синтаксических, семантиче-

ских и прагматических определений куч и их проверок.

Ли [184] справедливо замечает, что логики высшего порядка очень важны как критерий применимости на практике. Как было обсуждено в начале этой главы, Пролог поддерживает такую возможность. Правила могут даже меняться, но нет необходимости менять правила во время интерпретации правил (см. главу 1). Поэтому, определённые правила доступны в обоих процессах (2) и (3).

**Наблюдение 4.15** (Сходство языков при верификации). *Рассматриваемые для кучи языки программирования  $P$ , спецификации  $S$  и верификации  $V$  имеют между собой общую взаимосвязь аналогично мета-паттерну «Model-View-Controller» по Рееенскаугу (см. рисунок 4.13).*

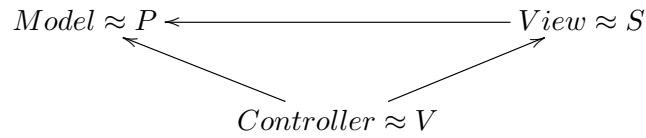


Рисунок 4.13: Мета-паттерн MVC применена к процессу верификации

Модель кучи описывается спецификацией  $S$  и обрабатывается языком верификации  $V$ . Пользователь использует  $V$  для изменения состояния и следит за результатами через  $S$  (см. опр.1.3). Однако, в классическом паттерне по Рееенскаугу  $V$  непосредственно манипулирует  $P$ , что здесь варьирует.  $P$  может иметь различные графовые представления  $S$  с обсуждёнными ранее свойствами. Сближение приводит также к тому, что основные интерфейсы коммуницируют на одном языке.

## 4.4 Представление знаний

Правила могут в декларативной парадигме быть представлены, в функциональном либо логическом виде. На примерах языков XLS-T и Пролог в [292] проводился количественный анализ по обработке слабо структурируемых данных (также как и термы в общем, см. предыдущий раздел). Для качественного сравнения эквивалентных программ проводилось множество проверок более 80 выбранных типичных и образцовых примеров, аккуратно вручную подобраны из множества учебников, монографий и онлайн ресурсов. Прямое сравнение показало (см. рисунок 4.14):

1. Пролог во всех примерах (кроме одного) превосходит в среднем на более чем 30% XSL-T, что можно вывести из соотношения  $N_T : N$  и  $\eta_1 : \eta_2$ .
2. В среднем описание той же самой функции в Прологе на 50% короче, а часто даже ещё короче. Обосновано такое решение на  $N$ ,  $\lambda$  и  $\Delta_N = \|N_T - N\|$ .

$N$	..	количество программных строк (длина программы)
$N_T$	..	теоретическая длина программы
$L$	..	интеллектуальный уровень (зависящий от языка)
$\lambda$	..	уровень языковой абстракции
$\eta_1$	..	количество операторов
$\eta_2$	..	количество операндов

$$\eta = \eta_1 + \eta_2 \qquad V = Nld(\eta)$$

$$N_T = \eta_1 ld(\eta_1) + \eta_2 ld(\eta_2) \quad \lambda = VL$$

Рисунок 4.14: Количественный анализ при использовании метрик Холстедта [112]

3. Функциональный язык страдает от замкнутости, т.к. могут быть использованы только встроенные операторы. В отличие от этого, когда логический язык предусматривает определять термовое IR, любые иные операторы и правила.

Более того, качественный анализ показывает и объясняет, почему выражения и правила в логическом представлении можно выразить намного проще. Это в основном из-за логического представления термов и реляций. В отличие от функциональных языков (имеющие компактную денотационную семантику), логически основаны на атомах, термах и правил с приоритетами (см. тез.4.10). Денотационная семантика является декларативной, однако, полный перебор логических взаимосвязей редуцируется только на одно более оптимальное отображение, а результат высчитывается на основе входного вектора. Переменные не являются символьными, а лишь переменными, которые даже в  $\lambda$ -абстракциях используются только в одностороннем порядке: присваивание значения (даже если по-ленивому) при использовании и вычислении функции. Подставкой параметризованных функций взамен реляций проблему не решить, как это наблюдается, например, в [38]. Когда необходимо выразить атомы, выражения и термы в общем, а также определить потенциально любые соотношения между ними, тогда разумнее ориентироваться на аксиоматическую семантику или семантику, основанную на соотношениях. То есть, успех представления знаний трансформаций и сравнения с существенной частью зависят от выбранной парадигмы, как было продемонстрировано в работах [292],[293].

Кроме ранее упомянутых особенностей, имеются следующие, которые необходимо учесть при верификации динамической памяти:

- **Правила компактны** и могут быть вызваны из любой позиции интерпретатора. Цели и подцели могут быть любыми. Нетерминация в общем неизбежна и искусственно не ограничивается. Однако, ради более эффективной обработки, в главу 6 может вводиться не значимое ограничение.

- Используется **строгое вычисление термов**, ленивое вычисление исключается. Это приводит к тому, что некоторые ситуации, например, передача рекурсивных данных Аккерманна (см. рисунок 4.1) не может осуществляться до тех пор, пока не будут вычислены все аргументы. Это ограничение практически не является значимым, т.к. всегда возможно написать программу без употребления незавершённых данных таким образом, что контроль записывается в тело правила. Более того, термовая унификация уже приводит к тому, что неопределённые частицы термов, т.е. символьные переменные присваиваются и при определении нет необходимости вычислять всё заново потому, что присваиваются лишь ранее неизвестные подтермы, а сам терм не меняется. Это возможно согласно принципу из рисунка 4.10.
- **Базовый тип** верификации является термом. Далее он может быть рассмотрен как параметризованный тип  $\lambda$ -вычисления третьей степени, т.е. тип из  $\Lambda_{T\lambda 3}$ , который допускает квантифицированные переменные. Тип, который построен из других типов, это зависимый тип (см. опр.1.9). Построение проводится согласно опр.4.1 с помощью функтора. Таким образом, сравнение термов может быть формализовано как проверка термов, однако, термы могут теоретически содержать сами себя в Прологе, благодаря символьным переменным. Практически этого можно избежать, если использовать проверку на самосодержащие термы (см. предикат `unify_with_check` в начале главы). Чем выше уровень  $\lambda$ -вычисления, тем сильнее действуют ограничения, а это означает, что меньше ожидается парадоксов. Функторы могут быть использованы для моделирования любых сложных структур данных, в том числе списков, деревьев и объектных экземпляров.
- **При успехе подцелей**, выдается подходящее множество сопоставлений, например:

$$?-H=pointsto(a,2), VC=pointsto(a,X), H=VC.$$

приводит к результату:  $H = pointsto(a,2)$ ,  $VC = pointsto(a,2)$  и  $X = 2$ . **Генерацию контр-примера** можно встроить в процесс унификации, например `unify_with_check`, если в отрицательных возможностях добавить разъяснение, используя `write` [248], т.к. прямое присвоение некоторых символьных значений невозможно и не имело бы смысла. Необходимо выявить сверху-вниз самый ближайший терм, который несопоставимый со сравниваемым термом. Принципиально, отслеживание модели вызовов можно совершить с помощью встроенной трассировки [83]. Если верификация проходит пошагово, разумно каждый шаг верификации записывать в виде «*DOT*»-файла для понятия и документации доказательства. Контр-пример предоставляет хотя бы один случай, когда верификация отклоняется. Унификация термов в Прологе приводит к наиболее общему сопоставлению, поэтому несовпадающие функторы по имени, арности или неунифицируемые переменные, являются сценариями отказа. В общем, частный пример означает терм, который может быть приведён в качестве отказа, всегда тогда, когда, атом не унифицируем с функтором, либо атомы или функторы различаются.

- **Снятие ограничений символьного характера** при определении куч [27] для запуска (см. главу 2), в том числе и для анализа правил верификации с обеих сторон при дедуктивном выводе ((*би-абдукция*)) [54], [211], можно осуществить, если правила вывода удобнее моделировать в качестве хорнских правил. Правила Хорна позволят, во-первых, квантифицировать  $\forall, \exists$  переменные в зависимости от того, где символьная переменная определяется и как она связана. Во-вторых, перебор правил и альтернатив может привести к успеху при поиске без дополнительных затрат. Однако, ради применимости, нужно объём альтернатив ограничивать. Если выбирается правило, то *абдукцию* можно имитировать следующим образом: дано правило Хорна « $b: -a_1, a_2, \dots, a_n.$ ». Если некоторые из  $a_j$  оставлять неопределёнными, т.е. имеют символьную зависимость, то, таким образом, могут быть выбраны различные  $b$ , если имеются альтернативы. Для этого строго требуется, чтобы начальные термы, аргументы головы  $b$  не исключались. Необходимо заметить, что Пролог не нуждается при выбранной модели кучи в дополнительных правилах, потому, что лексикографический порядок по указателю простых куч и возможностей уникального выбора (простых куч) встроенных предикатов по работе со списками, как например `concat`, уже дают широкий круг выбора и преобразований новых термов из старых подтермов. `concat` и иные сканирующие одним ходом предикаты эффективны благодаря свойству неповторимости. Поэтому, целый набор преобразовательных и вычислительных правил отпадает. Без ограничения общности, правила вывода обратимы, если между всеми подтермами предусловия и постусловиями, либо существует изоморфизм, либо необратимые встроенные предикаты декларативно в обратном случае не вызываются. В случае, когда изоморфизм нарушается, то, либо в отображении от входного вектора на выходной, либо в обратном случае производится расширение (см. абстрактная интерпретация из главы 1). Расширение является проблематичным при обратных отображениях, т.к. относительно кообласти происходит сужение домена, т.е. отображение становится прерывным.
- **Перегрузка значений** правил может осуществляться и пополняться различными телами, которые внутри могут реализоваться, например, на языке Ява, до тех пор, пока коммуникация осуществляется с помощью входящих, выходящих и комбинированных термов. Для анализа прологовских правил не достаточно использовать «*DCG*» (см. главу 1), потому, что необходимо изменить контроль и тактику выбора. Лучший пример может обсуждаться при принятии стратегий восходящих и нисходящих синтаксических анализаторов (см. главу 6).

## 4.5 Архитектура системы верификации

В [300] и [297] предлагается архитектура верификации с динамической памятью на основе Пролога. Архитектура представляется на рисунке 4.15. Архитектура следует ключевым принципам, которые были обсуждены в главе 1, это: (1) автоматизация, (2) открытость, (3) расширяемость и (4) обос-

нованность. (1) гласит от том, что доказательство находилось автоматически. В Прологе решение будет найдено, в зависимости от данных правил, если в правилах исключаются циклы и даны все необходимые правила, иначе, тогда доказательство не терминирует, либо завершает верификацию преждевременно. Далее, с помощью подхода, в главе 6 происходит автоматизация. (2) означает, что искусственные ограничения между термами, правилами и возможными реализациями должны отсутствовать. С одной стороны Пролог является открытым, т.е. могут быть добавлены все новые правила, а имеющиеся могут быть обновлены, если их определить ранее. С другой стороны, Пролог замкнут тем, что только то выводимо, что следует из данных правил согласно дедукции. Пункт (2) относится к архитектуре и используемым моделям представления памяти. (3) означает, что модель памяти может быть расширена и при необходимости изменены данные правила. Изменения всегда разрешаются благодаря переопределению правил Хорна. Расширяемость термов и правил также обсуждались детально в предыдущем разделе. (4) означает, что любой шаг верификации можно отслеживать и проверять, как обоснованные шаги логического вывода. Генерация «*DOT*»-файла визуализирует вычисление проверки, а при отказе генерация контр-примеров даёт более подробные результаты и предпосылки. Возможность интерактивно без соблюдения каких бы то ни было конвенций сильно упрощает и способствует проверке на обоснованность принятых решений.

На рисунке 4.15 на вход поступает данная программа, предпочтительно на языке Си (или другом императивном), который вместе с утверждениями о программе преобразуется в прологовские термы и правила. Синтаксический, а затем семантический анализ проверяет и исключает недопустимые типовые ошибки и основные ошибки нотаций. Термы всегда можно визуализировать в файловом формате «*DOT*». Утверждения могут ссылаться на леммы и теоремы, которые могут быть записаны непосредственно на языке Пролог и при необходимости могут быть использованы при верификации. При верификации включаются различные правила, которые задаются в теориях Пролога и загружаются динамически при интерпретации, например средой [82], [81]. Для решения отдельных теорий могут быть использованы, либо экстерные средства механизмом мульти-парадигмальной системы, либо подключением некоторых произвольно определённых SAT-решателей в самом Прологе. Напомним, Пролог широко используется в области решателей и «*Constraint Programming*». Новые этапы решения проблем динамической памяти могут быть подключены согласно принципу из рисунка 4.12. Переходы между маленькими и большими фазами совершаются, благодаря передаче состояний вычислений, т.е. процесс работает согласно потоку данных (см. рисунок 4.15) и может быть сравниваемым с общей архитектурой существующих конвейеров [135], [252]. Зависимость данных отличается от классического потока данных [137], из-за блока видимости динамически выделенных данных (ср. главу 3), который обычно не соответствует автоматически выделенным переменным на стеке. Однако, инфраструктура предложенного конвейера принципиально может быть использована при условии, если анализ псевдонимов установит отдельные интервалы видимости, см. набл.3.4.

Далее в [301] обсуждаются и предлагаются основные критерии для расширяемой и модифициру-



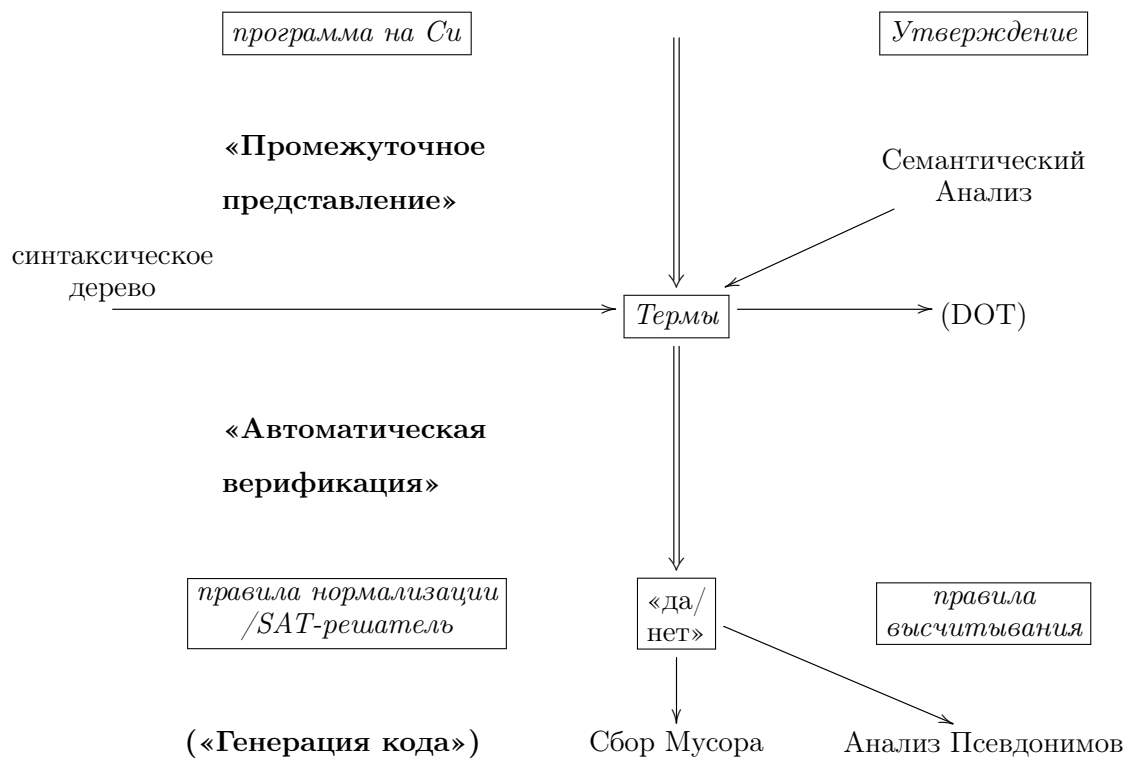


Рисунок 4.15: Архитектура верификатора динамической памяти

емой архитектуры. В первую очередь это относится к минимальности IR входного языка, который к сравнению с «PCF» [179] допускает присвоение для основы объектного вычисления по Абади-Карделли (см. раздел 1.2), неограниченный цикл и вызов процедур. Так как представленная модель вычисления, согласно классному виду вычисления, может быть типизирована по Абади-Карделли (обновления кода во время запуска исключаются, например, передача аргументов производится выборочно, либо по значению, либо по вызову с особенностью вставленной конструкции более широкого функторного объекта), то и свойства, согласно второй и третьей степени  $\lambda$ -вычисления, (см. опр.1.9 и опр.1.10) применяются непосредственно. Расширение может быть применено к входному языку программирования, к фазам статического анализа и правилам (включая имеющиеся).

В [302] вводятся фрейм, куча и их интерпретации, а также даётся предложение практического представления в Прологе. Так как куча предположительно представляется как терм, который принадлежит предикатной интерпретации в Прологе, тогда: либо верно и меняются все неопределённые символьные переменные подцелей, либо даётся отказ с предположительной причиной. Верификация похожа на такой же процесс сравнения, как представленный в [298],[307], т.е. автоматом сравнения по образцам и деревьям (с англ. «tree graph matcher» [66]). Хотя изначальная модель постановления процесса очень похожа, всё равно её сравнение тяжелее, например: за счёт вызовов процедур, абстрактных предикатов, пред- и постусловий и моделей графа кучи. Таким образом, из аналогии следует: если утверждения являются схемой/типом, а программа пошагово строит выражение, т.е.

граф кучи, тогда верификация является проверкой типов. — В том случае, если означает содержимое (см. рисунок 1.13)?

**Заключение 4.16** (Минимизация входной программы). *Результатом проверки содержимого является минимальная входная программа, которая манипулирует динамической памятью.*

Ответ кажется простым: входная программа. Однако, соотношение между «*типом*» и «*выражением*» не может быть однозначным. Возникает вопрос, а какая действительная программа генерируется? Суть в том, что программа генерируется пошагово только при необходимости в соответствии с «*утверждениями*». Программа строится минимальным образом, потому, что каждая грань графа куч соответствует всё новым программным операторам. Цикл в графе кучи не обязательно равен циклу в программных операторах, а, например, может быть равен двум операторам. Бесконечно много граней в графах исключается, поэтому только похожие цепочки могут соответствовать циклу в качестве программного оператора, условие которое определяется графом. При конструкции входной программы минимальность означает лишь то, что множество операций, которые не имеют прямого отношения к итоговому графу, удаляются и остаются только те операторы, которые действительно необходимы для построения финального графа кучи. Таким образом, проверка *содержимого выражения данного типа* может сгенерировать минимальную программу, которую можно сравнить с данной программой.

Архитектура в Прологе без дополнительных затрат разрешает следующее:

- **Любой входной язык** допускается, если термовое IR соблюдается, которое также может быть изменено и добавлено в правилах. Входная программа может быть даже пуста и процесс синтаксического анализа пропущен, если IR программы или её части вводятся вручную для соответствующей части верификации. Архитектура, представленная на рисунке 4.15 позволяет исключить синтаксические и семантические ошибки с помощью гибких фаз из рисунка 4.12.
- **Скромные спецификации** разрешают избегать полную спецификацию, поэтому специфицируются только те модули, которые нуждаются в верификации. Кроме полной спецификации, никаких альтернатив не было, что раньше приводило к большим затратам и трудно читаемым утверждениям. Этот подход называется «*footprint*» и наблюдается в области верификации куч с помощью ЛРП впервые в «*Smallfoot*». Принцип в Прологе прост, если имеется утверждение, то оно проверяется на верность, если нет, то верификация по умолчанию продолжается. Верификация применяется только для тех модулей, которые содержат утверждения. Чтобы избежать полную спецификацию кучи, надо использовать вспомогательные предикаты, такие как *true* или *false* (см. главу 5), а также предположить данные спецификации выборочно и не полностью.

Полиморфизм исключен ради простоты из корневых программных операторов (см. дискуссии в главе 1,3 и далее). Граф зависимостей заданных правил и лемм утверждений анализируются при запросе интерпретатором Пролога, а также во время синтаксического анализа, например, при построении компиляции (см. главу 6).

## 4.6 Объектные экземпляры

В разделе 1.2 подробно обсуждались два вида вычислений с объектами — по Абади-Карделли и Абади-Лейно. Из-за широкого распространения в императивных языках программирования с объектно-ориентированным расширением используется первый вид. Как было продемонстрировано и обсуждено в ранних главах, объектный вид тяжелее проследивать в связи с верификацией корректности и полноты. С теоретической точки зрения оба вида имеют одинаковую выразимость [155], [220] и полная абстракция может быть всегда найдена (см. ранее). Однако, написание программ может сильно отличаться и тогда доказательство полной абстракции, т.е. равенства операционной и денотационной семантики между обоими видами очень сильно расходится из-за трудной формализации относительно простых свойств объектного вида вычисления (см. раздел 1.2). Так как простота спецификации имеет наиболее важный эффект на общую простоту доказательств, тогда выбирается именно классный вид вычисления.

Объект — это, прежде всего, экземпляр некоторого класса. Ради простоты, полиморфизм исключается, т.к. он не имеет прямого отношения к корневой функциональности вычисления (настоящий полиморфизм переменных не рассматривается, т.к. в объектно-ориентированных языках полиморфизм выражается *спонтанным* полиморфизмом [55] исключительно с помощью подклассов [48]), и представляет собой только более удобный способ вызова подходящего метода во время запуска (см. раздел 1.2). В вычислении Хора полиморфизм выразим, но в наших целях предлагает лишь дополнительный эффект без увеличения выразимости (см. главу 1, [185]). Поэтому, объект, моделируемый кучу, является замкнутым регионом памяти без дыр, т.е. определённого типа, который содержит лишь поля. Методы не сохраняются вместе в динамической памяти, потому, что они статические и не меняются во время запуска. Изменение кода во время запуска также исключается из-за минимального успеха и огромных проблем свойств корректности и полноты (см. дискуссию из раздела 1.2). Так как объект присвоен данному типу класса, методы полностью определены. Наследственные поля и методы также полностью определены. Для наиболее легкого сравнения объектных экземпляров, соблюдается конвенция, что пары (наименование поля × содержимое) отсортированы по лексикографическому порядку. Таким образом, проверку и преобразование экземпляра на подкласс, можно реализовать двумя способами: (1) каждое поле проверяется согласно соотношению «>:» (см. опр.1.11), при этом, множество полей в верхних и нижних подклассных экземплярах расходятся или (2) все поля группируются согласно идентификатору наследованности. Таким об-

разом, в памяти необходимо эффективно сравнивать экземпляры нижних классов только с маленьким подмножеством, которое представляет экземпляр верхнего класса. Чтобы продемонстрировать (2), возьмем пример: «SubClass1 s1; SuperClass o1=(SuperClass1)s1;». Допустим, s1 содержит [o1,o2,o3], тогда вычисление o1, где верхний класс в SuperClass1 наследует только поля o1 и o2, может производиться преобразование с помощью копирования первых двух полей, т.е. начальным сегментом s1.

Поля объектных экземпляров записываются в список кортежом (*наименования, значение*). Ссылки на объектные экземпляры (в том числе циклические), выражаются символьными переменными. A=object(A,A) запрещается и может быть исключено с помощью unify\_with\_check.

A=object([(a,A),(b,A)]) разрешается. Предполагается использовать упрощенную форму:

A=object((a,A),(b,A)), т.к. функтор object внутри Пролога уже строит список головы, которая содержит object. Так, как кортеж содержит ровно два элемента, то сложный терм будет всегда четко определен и однозначен в отношении объектной сети.

Объектные поля доступны с помощью «.»-оператора и могут быть использованы в программных операторах и утверждениях. Исключение неверных доступов обнаруживается во время семантического анализа. Спецификация (всех) полей данного объекта производится на уровне абстрактного предиката, которые также могут быть определены, в том числе частично (см. главу 5 и далее).

Конвенции из главы 1, обсужденные в этой главе, а также конв.5.17 и конв.5.18 вводятся, во избежание парадоксов, с целью приближения к «UML».

## 5 Ужесточение выразимости куч

В этой главе рассматривается *ЛРП* и анализируются проблемы в связи с пространственными операторами. Получается, что один и тот же оператор приспособлен, согласно *графу куч*, разделять и связывать между собой кучи, в зависимости от состояния указателей и их содержания. Получается, оператор имеет различные аспекты в зависимости от контекста используемой формулы. Другими словами, единый пространственный оператор в классической *ЛРП* является *многозначимым* [296]. Многозначимость, это удобная запись, но влечёт за собой недостатки. Наиболее важными недостатками являются **контекст-зависимость**. Зависимость, прежде всего, означает, необходимость анализировать всю формулу, что может быть (не) связано с данной кучей и все её содержавшие кучи. Чтобы определить независимый граф кучи (т.е. семантически контекст-независимо), требуется использовать зависимую нотацию того же графа кучи (т.е. синтаксически контекст-зависимо). Это не является парадоксом, однако, желает иметь лучшее. Далее мы покажем, что синтаксически возможно определить граф кучи контекст-независимо без ограничения общности, исключить целый ряд проблем и улучшить процесс автоматизированного доказательства. Для автоматизации синтаксический анализ при интерпретации формул, которые описывают кучи, является накладным и избыточными, если пространственное отношение между кучами удастся явным образом выразить. Соотношение между кучами может быть связанное или не связанное. Переписывание многозначной формулы кучи в однозначную (единственную) формулу может быть не тривиально, т.к. необходимо рассматривать все переходы от одной кучи к другой, либо проверить отсутствие любых переходов из одной кучи в другую, на что может потребоваться значительное время. И наоборот, если имеется однозначная формула, то не ожидается сюрпризов в связи с соотношениями куч. (Не-)связанная куча с некоторой другой кучей сохраняется, при этом не имеет значения, какой предшествует контекст или следует иерархически определённой куче. Используя синтаксически контекст-независимую формулу для описания семантически контекст-независимой модели памяти графа кучи объединяет понятие о том, что такое куча.

Глава разбита на семь разделов. В первом разделе анализируются *ЛРП* и последствия многозначности. Во втором разделе проблема многозначности локализуется, и обсуждаются подходы к преодолению проблемы. В третьем разделе рассматривается ужесточение многозначности как решение проблемы. В четвёртом разделе в частности, рассматриваются объектные экземпляры классового вычисления. Объект рассматривается как комплексная единица *ЛРП*, на который распространяются

те же самые свойства ужесточения и для простых ссылок. В связи с ужесточением операторов в пятом разделе обсуждается возможность специфицировать лишь часть динамической кучи, благодаря свойствам строгого пространственного соотношения подкуч. В частности, обсуждается модульность спецификации и улучшения качества программного обеспечения в связи с объектами. В шестом разделе подробнее обсуждаются возможности применения формальных свойств ужесточенной модели памяти. В последнем разделе обсуждаются возможности и ограничения предложенной модели.

## 5.1 Мотивация

Возьмём следующее синтаксическое определение термовых выражений  $E$  над целыми числами в классической арифметике целых чисел в качестве рассматриваемой проблемы многозначности:

$$\langle E \rangle ::= \langle k \rangle \mid \langle E \rangle ' \otimes ' \langle E \rangle$$

Нетрудно убедиться в том, что синтаксис по Бэккуса-Науру представляет собой индуктивно-определяемые термы, где начальное определение любое, но определённое целое число  $k$ . Допустим,  $\otimes$  является некоторым бинарным оператором, который полностью определён для целых чисел, например сложение. Если мы имеем ситуацию, когда для выражения  $e_1, e_2, e_3: E_0 \cdots \otimes e_1 \otimes e_2 \otimes \cdots E_n$  и  $n \in \mathbb{N}_0$  один раз вычисляется как  $e_{1,2}$ , а при  $E'_0 \cdots \otimes e_1 \otimes e_2 \otimes \cdots E'_n$  вычисляется как  $e'_{1,2}$ , при этом  $e_{1,2} \neq e'_{1,2}$ , то либо правила вычисления не корректны (возможно, ошибка совершена в стадии разработки; далее исключается), либо вычисление зависит от контекста, т.е. зависит от  $E_0$  и  $E_n$ , либо  $E'_0$  и  $E'_n$ . Необходимо заметить, что если  $E_0 \equiv E'_0$  и т.д. до  $E_n \equiv E'_n$ , то проблема различия всё-таки совпадает с проблемой (не-)корректности вычисления. Исходя из стандартного случая, т.е.  $e_{1,2} \neq e'_{1,2}$  при  $E_0 \neq E'_0$ ,  $E_n \neq E'_n$ , можем утверждать, что обе  $E_0$  и  $E_n$  одновременно не пусты. Следовательно, зависимость означает при  $(e_1 \otimes e_2) \otimes e_3$ , что  $e_3$  содержит синтаксическую информацию, которая влияет на результат  $e_1 \otimes e_2$ . А поэтому, для каждого  $j$  умножения  $\otimes_{0 \leq j}^n e_j$  в худшем случае означает полный синтаксический перебор всех остальных факторов. Сложность ограничивается рангом полинома  $\binom{n}{2}$ . Какое отношение эта граница имеет к кучам?

**Наблюдение 5.1** (Перегрузка оператора). *Операция « $\star$ » является многозначной (см. позже) и она может быть использована для соединения, а также для разделения куч. Это усложняет логический анализ куч.*

Из набл.5.1 следует, что определение « $\star$ » из ЛРП очень близко к определению « $\otimes$ » вверху (см. опр.5.6). Поэтому при анализе каждую из куч необходимо внимательно проверять (что означает  $E$  в верхнем примере). Поэтому, имеется следующее предложение:

**Тезис 5.2** (Ужесточение выразимости). *Если ужесточить выразимость пространственного оператора ЛРП, то удастся исключить семантическую многозначность. Исключение контекст-зависимости операции позволит автоматизировать и упростить анализ куч.*

Доказательства этому и последующим тезисам будут следовать в этой главе.

**Тезис 5.3** (Упрощение с помощью высчитывания куч). *Соблюдая синтаксическое и семантическое единство, адекватное представление упростит сравнение и спецификацию данных и желаемых куч. Сравнение может производиться с помощью вычитания куч.*

Из этого тезиса следует, что контекст-независимость позволит определить формальные теории о равенствах и неравенствах куч, которые далее можно будет автоматизировать ради подключения SMT-решателя.

**Тезис 5.4** (Неполнота для улучшения полноты). *Формулировка неполных куч способствует решению проблемы о полноте специфицируемых правил верификации для куч.*

**Заключение 5.5** (Ужесточение в моделировании). *Ужесточение операторов не нарушает основные свойства локальности (объектных) куч. Ужесточение может послужить примером расширения для языка моделирования «UML/OCL», который на данный момент не поддерживает ссылок.*

*Доказательство.* Идея заключается в расширении пространственным соотношением, ссылаясь на трм.5.8 и лем.5.10, которое может связывать, либо разделять.

Язык моделирования «UML/OCL» основан на типизированном лямбда-вычислении второго порядка, следовательно, эквивалентен опр.1.9, следовательно может быть выражен в типизированном лямбда-вычислении третьего порядка, следовательно может быть представлен в прологовских терминах как описано в главе 4. □

Данные наблюдения и тезисы следуют анализам предыдущих глав и замечаний. Из предыдущих наблюдений и анализов можно заметить следующее:

1. Простая модель должна быть представлена простым способом. *Контр-примером* здесь может послужить [249]. Там, на первый взгляд неполное множество представляет на самом деле полное множество правил, которое может совершать очень сложные операции с указателями. Самые незначительные изменения могут легко привести к иному или не предсказуемому поведению. То есть, в сильно динамической системе минимальные изменения не должны менять весь характер поведения, особенно не должны менять *далекие* пространственные части куч.
2. Различные предыдущие модели памяти, точнее, их конвенции, не столь важны, как может показаться на первый взгляд. Показано, что ввод всё новых конвенций не расширяет, а наоборот, ограничивает выразимость дополнительными условиями. Предлагаемые новые возможности входных языков или языков спецификации настолько специфичны, что применение и метод верификации не устойчивы к малейшим модификациям и расширениям (см. главу 4). С практической точки зрения гораздо важнее описать точно и адекватно ту модель памяти, которая имеется, вводя как можно меньше искусственных ограничений и описывая только основное.

Для описания, включая все возможные ограничения, берётся непосредственно *граф кучи*. Это является *утилитаристским подходом*. Эпистемологическое определение термина «кучи» дается в главе 3.

В итоге реализации получается формальная грамматика операторов, с одним бинарным оператором для слияния и одним оператором для разделения, которая контекст-свободная (например, подграмматика  $E$ ). Предложенная ужесточённая модель предусмотрена для более эффективного проведения верификации, отделив правила теории куч от общих логических правил. Правила верификации представляются в качестве правил Хорна, и интерпретация правил совершается с помощью Пролога [297] (см. тез.4.11). На следующем этапе ужесточённые операторы заменяют оператор  $\star$  так, чтобы абстрактные предикаты могли быть автоматически распознаны при синтаксическом анализе (см. главу 6).

## 5.2 Многозначимость операторов

В качестве наиболее точного языка спецификации в вычислении Хора изначально предлагалось использовать математику, как наиболее точный формализм. Позже математика уточняется логикой предикатов в самом общем виде. В области верификации и спецификации куч можно использовать специальные логики, что довольно успешно применяется на практике (см. раздел 1.1). Однако, неограниченные формулы могут оказаться более удобными при автоматизации, которые увидим позже. Одно из таких «более приемлемых» условий автоматизации может оказаться *однозначное представление пространственных операторов*. Здесь необходимо пояснить возникающий парадокс: ужесточение операторов является условием расширения выразимости, что будет продемонстрировано позже. Ужесточение условий формул куч естественно приводит к ограничениям.

Проблема точности и выразимости является фундаментальной проблемой не только в области логики, но также в науке в самых различных областях, начиная от восприятия, до записи по согласованным конвенциям до выражений. Надо искать причину возникновения многозначности. Неудивительно, если язык выражений программных операторов и декларативный язык спецификации различны, то могут появляться разрывы в семиотике (см. главу 4) объектов и их взаимосвязи. Элементарный вопрос о равенстве двух различных представлений о куче (см. главу 3) может существенно усложняться, если не использовать единую, либо согласованную форму — это причина, которая лежит в определении куч.

Итак, вопрос об изоморфизме двух графов куч в обобщённом виде обсуждается в отображении на рисунке 5.1, и может быть оценён как тяжёлый. Решение изоморфизма может быть оценено, в общем, с экспоненциальной сложностью, для довольно плохих прогнозов. На практике имеются экспоненциальные алгоритмы, которые приближаются к полиному третьего ранга для небольшого объёма входных вершин. Если куча из рисунка 5.1 (а) содержит только сплошные линии, а при даль-



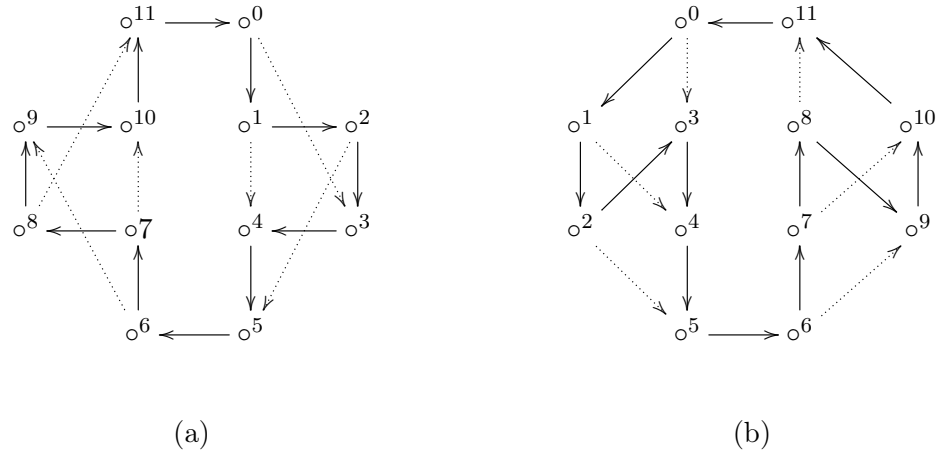


Рисунок 5.1: Изоморфизмы смежных куч с объектами

нейшем анализе выходит, что также имеется в частности соединение между вершинами №0 и №5, когда обе вершины кучи уже были специфицированы, то выявление изоморфизма сильно усложняется. Однако, сложность сужается ради типов и наименований, которые не меняются в итоге. Проблема сложности изоморфизма сохраняется лишь тогда, когда имеется набор указателей вместе с графом кучи, которые можно преобразовать в другой граф, который отличается от предыдущего только множеством наименований вершин. Для рисунка 5.1 это может быть совершено с помощью пермутации  $(0\ 11)(1\ 8\ 2\ 10\ 3\ 9)(4\ 7)(5\ 6)$ , исходя из графа на рисунке 5.1 б). С практической точки зрения вопрос изоморфизма стоит только тогда, когда необходимо проверить, может ли в принципе данная структура быть преобразована в другой граф, если допустить, что наименования могут меняться.

В главе 6 рассматриваются абстрактные предикаты более детально, однако, идея абстракции предикатов лежит в свёртывании и развёртывании графа. К примеру рассмотрим рисунок 5.2.

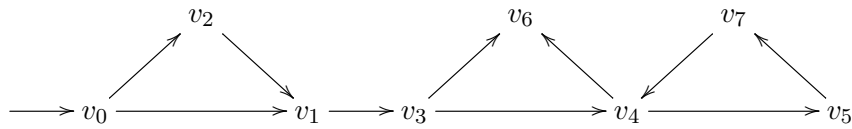


Рисунок 5.2: Пример связанного графа кучи

Данный граф согласно минимальной зависимости можно разбить на подграфы вдоль мостика  $v_1 \mapsto v_3$ , см. рисунок 5.3.

Граф можно будет описать отдельными предикатами  $\pi_0(v_0, v_1)$ ,  $\pi_1(v_3, v_4)$  и предикатом  $\pi_2(v_4, v_5)$ , либо более абстрактно как:  $\pi_0(v_0, v_1)$ ,  $\pi_{1,2}(v_3, v_5)$ , при этом, графы представленные предикатами связаны между собой и видимые вершины снаружи появляются в качестве аргументов неявным определением  $\pi_j$ , см. рисунок 5.4.

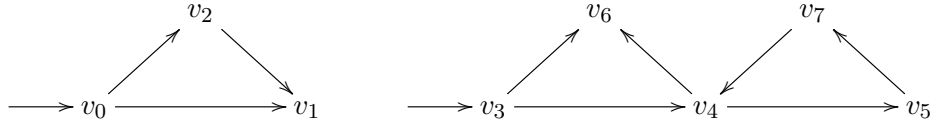


Рисунок 5.3: Пример разбитого на две части графа кучи

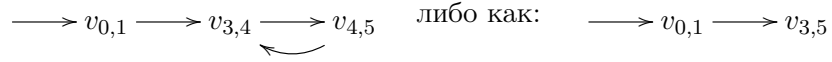


Рисунок 5.4: Пример возможного разбиения графа кучи на отдельные части

Развёртывание согласно определению  $\pi_j$  приводит к обратному.

Теперь можно вывести более обобщённые вопросы в связи с адекватным представлением кучи следующим образом:

1. Как специфицировать однозначные формулы и проводить максимально детерминированную верификацию?
2. Как решить вопросы об изоморфизме, локальности и абстракции графов простым образом (представление кучи)?
3. Как ограничить принудительную проверку объектов в программных операторах кода (представление объектов)?
4. Как эффективно решать равенства с кучами, если не все вершины (и грани) графа кучи полностью определены (*частичная спецификация*)?
5. Как можно избавиться от повторного анализа куч (пошаговая верификация)?

В главе 3 уже была представлена модель кучи, которая была предложена Рейнольдсом, Бурсталлом и другими. В этой главе итоги и эффекты определения Рейнольдса рассматриваются и проводятся дискуссии о выводимом графе, а также графах получившие от модификации различных параметров. Наблюдаются свойства и выразимость, что и является главным замыслом этой главы.

**Определение 5.6** (Выводимая куча по Рейнольдсу). *Куча определена как объединение  $\bigcup_{A \subseteq Addr} A \mapsto Val^n$  с  $n \geq 1$ , где  $A$  является некоторым адресным пространством и  $Val$  является некоторым доменом значений (например, целых чисел или опять же  $A$ ). Исходя из наблюдаемого поведения, можно вывести следующие свойства:*

Даны две кучи  $H_1$  и  $H_2$ , то  $H_1 \star H_2$ , где  $H_1$  описывает утверждение о куче  $H_1 = (V_1, E_1)$  (аналогичное происходит с  $H_2 = (V_2, E_2)$ ), где направленные грани графа  $E = V \times V$  такие, что соблюдается  $\forall v_1 \in V_1, v_2 \in V_2$  с  $v_1 \neq v_2$  и  $V_1, V_2 \subseteq V$  со следующими случаями:

- **Первый случай** (Разделение):  $(v_1, v_2) \notin E_1$ , и  $(v_1, v_2) \notin E_2$ .
- **Второй случай** (Слияние):  $\exists s \in V_1, \exists t \in V_2 : (s, t) \in E_1$  или  $(s, t) \in E_2$ , тогда  $H_1$  или  $H_2$  содержит  $\star$ -разделенные  $s \mapsto t$ .

Переменные, также как и указатели, хранятся в стеке, а содержимое указателей хранится в динамической памяти. Следующее доменное равенство согласно [26] действительно:  $Stack = Values \cup Locals$ . Утверждения меняются программными операторами и генерируются при верификации, при проверке куч, исходя из программных операторов. Утверждения о кучах, либо *верны*, либо *ложны*, в зависимости от конкретной кучи. Синтаксис утверждений определяется опр.3.10. Из опр.5.6 следует, что бинарный оператор « $\star$ » может быть использован двумя способами: для того, чтобы выразить две кучи не пересекаясь, а также, чтобы две кучи делили между собой один общий символ. Оператор « $\star$ » используется как логическая конъюнкция для связывания истины о кучах. Кроме того, он является, пространственным оператором, который выражает место нахождения и связанность. Она выражается тем, что связывающие формулы о кучах определяют, как две кучи расположены в некотором адресном пространстве касательно друг друга. Пространство подразумевает, что кучи занимают некоторые поля динамической памяти. Если определить связанность между двумя кучами как *двудольный граф* (биграф), то имеется левая сторона указателей и правая сторона множеств содержимого. Для того, чтобы связанный граф можно было полностью описать, вычитав *максимальное паросочетание* с целью уменьшения количества конъюнкций для формулы куч, соответствовала бы полностью *графу куч*. Такой подход на практике очень не практичен, т.к. нет необходимости и желания, со стороны разработчика, описывать *максимально сжатое представление графа куч* целиком (см. раздел 3.1). Но, если полученный граф кучи сильно отличается от ожидаемого графа, то неожиданные «*лишние части*» кучи являются показателем возможных очагов ошибок в программе. Важны и другие критерии, например, адекватное соотношение между синтаксическим представлением и графом, с целью нахождения вершин и граней, а также навигации по граням, и т.д. Компактное представление абсолютно не даёт преимуществ в данном случае, а наоборот, трудно читаемо. Необходимо сравнивать конкретные состояния ячеек в динамической памяти. По этой причине *регулярные выражения*, как удобный вариант отпадают. Регулярные выражения страдают проблемой *нелокальности*: как только граф куч локально меняется в одном месте, то может последовать изменение целого выражения. Желаемое поведение, должно быть таким, что добавление одной грани в граф кучи не должно менять всю формулу, а лишь ту часть подвыражения или части формулы, которая непосредственно связана с меняющейся частью. Не причастные (под-)кучи не должны меняться.

Сравнив с опр.5.6, а также определения и дискуссию из главы 3, можно заметить, что оно довольно трудное, а данные формулы, использующие это определение, могут быть многозначными, если всегда анализировать только часть от формулы. Для полного решения взаимосвязей всегда необходимо полностью анализировать все конъюнкты. Данное определение одной кучи, является результатом, если попытаться определить отдельную кучу. Напомним, что Рейнольдс определяет только множество куч, а отдельная куча у него, так и не определена. Увы, другие авторы (см. главу 1) также определяют только множественные кучи, но не определяют единственную форму кучи, если даже между строками авторы дают неполные и неформальные предпосылки на неё. Надо обратить внимание на то, что выведенное определение кучи в опр.5.6 является явным определением одной кучи, при этом, множеству куч не противоречит определению. Отсюда — берётся мотивация необходимости строже и явным образом решить многозначность « $\star$ ». Когда имеются однозначные операции, тогда можно обращаться к отдельной куче с помощью одного символа. Фактически модель Рейнольдса (и других) анализирует и подразумевает только смесь куч. Представление об отдельной куче отсутствует. Куча имеет только тогда значимое объяснение, когда оно задаётся вместе с чем-то. Ввод строгих операторов позволяет выразить семантику и замысел одной кучи, которая как таковой субъект естественно существует независимо от других куч. Следовательно, куча имеет идентичность. Таким образом, определение можно избежать лишь через значение нескольких куч, т.е. «структуралистские семантики» меняются на «не (строго) структуралистические семантики». Как только два оператора будут определены ( $\circ$ ,  $\parallel$ ), далее свойства и равенства могут быть исследованы, вследствие чего, термовые алгебры можно будет определить для решения прогрессивной сходимости верификации. Термовые алгебры разрешат установить всё новые и новые формальные теории над кучами, которые практически можно будет использовать на прямую, в качестве правил Хорна в рамках Пролог программы (см. тез.4.11).

### 5.3 Ужесточение операторов

Из-за многозначности, оператор  $\star$  может быть использован как конъюнкция (*слияния куч*), а также как дизъюнкция (*деление куч*). Более того, строгие различия (трм.5.2) в реализациях ЛРП часто  $\star$  используются равномерно логической конъюнкцией. Решение этих проблем является задачей данного раздела. Вводится формальное определение *конъюнкции кучи* и свойства этой операции, затем вводится дизъюнкция. Будет показано, что общность выразимости при обеих операциях не ограничивается.

**Определение 5.7** (Конъюнкция куч). *Конъюнкция куч  $H \circ \alpha \mapsto \beta$  определяется как граф кучи, где  $G = (V, E)$  является представлением графа кучи  $H$ , и где  $\alpha \mapsto \beta$  является обыкновенной кучей:*

$$\begin{cases} (V \cup \{\alpha, \beta\} \cup \beta', & \text{если } isFreeIn(\alpha, H) \\ E \cup \{(\alpha, \beta)\} \cup \{(\beta, b) | b \in \beta'\} & \text{если } H = \underline{emp} \\ & (V = E = \emptyset) \\ \underline{false} & \text{иначе} \end{cases}$$

Здесь  $\beta' = vertices(\beta) \subseteq V$  определяет все вершины графа кучи, которые указываются из  $\beta$ . В случае если  $\beta$  является объектным экземпляром, тогда также рассматриваются все поля объекта, которые являются указателями. Так как  $\alpha$  может ссылаться лишь на одну уникальную вершину графа (например, *путь доступа* к некоторому объектному экземпляру), тогда и существует не более одной совпадающей вершины в  $isFreeIn$  для данной кучи  $H$ . Общее предположение высшего определения заключается в том, что при пошаговом построении графа при использовании конъюнкции, всегда существует одна подходящая вершина, иначе, две кучи нельзя связать вместе.

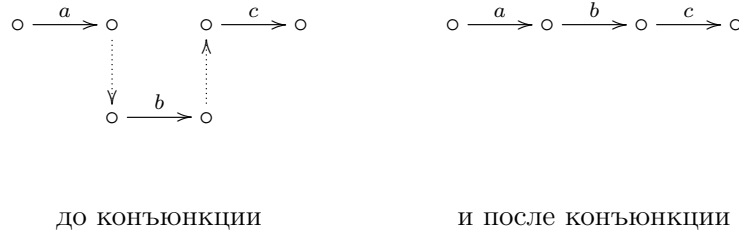


Рисунок 5.5: Граф кучи до и после конъюнкции

Например, нужно связать три пары указателей (указатель ссылается на некоторое содержимое)  $a, b, c$  (см. рисунок 5.5). Сначала необходимо выразить кучу  $a$ , которая может быть любой, либо как  $\underline{emp} \circ a$ . Только когда  $a$  существует,  $a$  ссылается на некоторое содержимое, которое эквивалентно началу кучи  $b$  и не связано (на данный момент мы подразумеваем именно такое состояние как начальное), только тогда обе кучи связываются. Если предположить обратное, т.е. в графе кучи имеются две одинаково содержимые, то по определению это исключено, т.к. допускается только одно по-настоящему уникальное содержимое (см. позже), но с любым количеством псевдонимов. Допустим, имеются одинаковые, но не по-настоящему одинаковые содержимые, тогда путь доступа должен отличаться, иначе, создаётся противоречие. В обоих случаях исключается возможность конъюнкции некорректного связывания. Итак, мы имеем связанную кучу  $a \circ b$ . Сейчас можно продолжить конъюнкцию под теми же условиями, как только что было изложено для  $c$ . При успехе мы получим граф кучи как на рисунке 5.5. Так как мы заинтересованы в конъюнкции любых графов, например двоичных деревьев, мы допускаем конъюнкцию в любых частях связанного графа. К примеру,  $a \mapsto 5$  является допустимой конъюнкцией графа кучи, однако,  $a \mapsto 5 \circ b \mapsto 5$  не является. Таким образом, мы сможем выразить псевдонимы, например, граф кучи  $x \circ \longrightarrow \circ^z \longleftarrow \circ y$  может быть выражен как терм кучи  $x \mapsto z \circ y \mapsto z$ .

Кучи могут быть связаны различными способами, когда вершина является объектом. Например,

можно договориться, что при присвоении объекта меняется только указатель, либо одно идентифицируемое поле. Можно также договориться о различных присвоениях. Например, когда все поля одновременно присваиваются некоторым входным вектором (например, массив или строка с определённым разделителем), либо присваивается только одно поле, а все остальные сбрасываются и т.д. (см. главу 3). Ради простоты, более популярный из языков программирования Си(++), далее рассматривается только присвоение «один-на-один» (см. раздел 5.4).

**Замечание:** Пусть  $\Phi_0$  некоторая куча, тогда следует  $\Phi = \Phi_0 \circ a_0 \mapsto b_0 \Leftrightarrow \exists (a_m \mapsto b_m) \in \Phi_0 \wedge (a_m = a_0, b_m \neq b_0 \vee a_m \neq a_0, b_m = b_0)$ .

**Теорема 5.8** (Конъюнкция обобщённых куч). *Если даны две кучи  $H_1$  и  $H_2$ , то конъюнкция  $\circ$  связывает данные кучи  $H_1$  и  $H_2$  вместе с одной кучей  $H_1 \circ H_2$ , если существует хотя бы одна общая вершина в обоих их представлениях графов куч, которая является общей. По умолчанию согласуется, что  $H_1 \circ \text{е\textit{м\textit{р}}} \equiv \text{е\textit{м\textit{р}}} \circ H_1 \equiv H_1$  в силе.*

*В отличие от опр.5.7, правая часть  $\circ$ -терма обыскивается в качестве подходящей первой вершины — это выбрано произвольно и не подлежит никакому обязательству, иначе, может быть изменено произвольно. Далее согласуется, что  $H_1 \circ H_2$  представляет собой единый объединённый граф кучи.*

*Доказательство.* Теорема является обобщением опр.5.7. Обе,  $H_1$  и  $H_2$  могут быть обыкновенными кучами видом  $a_1 \mapsto b_1 \circ a_2 \mapsto b_2 \circ \dots \circ a_n \mapsto b_n$ . Чтобы доказать корректность теоремы, сначала необходимо показать, что если не существует общий элемент в обоих графах, то согласно опр.5.7 получаем false, что совпадает с ожидаемым от конъюнкции. В противном случае, если имеется хотя бы один общий элемент, то согласно индукции, выбираем один элемент и тогда обе кучи связываются. Нужно отметить, что конъюнкция исходит лишь от возможности связывать графы и нас не интересует количество более одного. Все вершины кроме той, которая используется для слияния графов, могут также гипотетично быть использованы для слияния. В таком случае, полученный граф всё равно остаётся просто связанным. В противном случае начало, либо конец слитого графа куч находится, только в  $H_1$  или только в  $H_2$ , а также в обоих графах куч  $H_1$  и  $H_2$  одновременно, но это исключается. Из-за этого противоречия следует годность теоремы. По определению,  $a \circ a$  равно false, что необходимо фильтровать для всех конъюнктов. Обсуждение  $a \circ a$ -решателя будет проводиться позже и может быть реализован с помощью активного множества, которое содержит статически все успешно обработанные базисные кучи.  $\square$

**Соглашение 5.9** (Локация). *В абстрактных предикатах, локации могут быть символами. В целях увеличения применимости абстрактных предикатов для различных локаций и их разновидностей (прежде всего для локаций и полей объектных экземпляров) согласуется, что доступ к полю реализуется с помощью лево-ассоциативного бинарного оператора «.».*

Лево-ассоциативность означает, что терм  $object1.field1.field2.field3$  по умолчанию равен:

$$(((object1).field1).field2).field3$$

– таким образом, части выражения доступа к полю могут сопоставляться символьными переменными. Это повышает модульность, а в частности, повышает гибкость выражений.

**Лемма 5.10** (Моноид конъюнкции).  $G = (\Omega, \circ)$  является моноидом, где  $\Omega$  определяет множество графов кучи и  $\circ$  является конъюнкцией куч.

*Доказательство.* Чтобы доказать, что  $G$  является моноидом, необходимо доказать: (i)  $\Omega$  замкнут под  $\circ$ , (ii)  $\circ$  является ассоциативной операцией, а также (iii)  $\exists \varepsilon \in \Omega. \forall m \in \Omega : m \circ \varepsilon = \varepsilon \circ m = m$ .

$\omega \in \Omega$  связной граф кучи, полученный с помощью бинарного функтора « $\mapsto$ » в соответствии с опр.3.9. Согласно опр.5.7  $\forall \omega \in \Omega : \omega \circ \omega = \underline{false}$  в силе. В противном случае, для  $\omega_1, \omega_2 \in \Omega$  могут быть только два случая: если  $\omega_1$  и  $\omega_2$  имеют, хотя бы одну, объединяющую вершину, тогда согласно трм.5.8 соответствующий граф куч определён, иначе, результат  $\underline{false}$  (обозначив,  $\omega_1$  и  $\omega_2$  не пересекается). Таким образом, мы показали, что  $\Omega$  замкнуто над  $\circ$  и что граф кучи может быть получен в результате конъюнкции. В таком случае, соединение успешно установлено.

Далее, ассоциативность должна быть доказана, а именно, что:  $m_1 \circ (m_2 \circ m_3) = (m_1 \circ m_2) \circ m_3$  в силе.

Рассмотрев рисунок 5.5, можно сразу констатировать верность равенства с обеих сторон, так как не имеет значения  $a$  и  $b$  связаны первыми, либо  $a$  связывается с  $b \circ c$ , потому, что соединяющая вершина  $b$  остаётся инвариантом, когда порядок конъюнкций меняется.

$G$  создает полугруппу. Для этого остаётся доказать существование нейтрального элемента  $\varepsilon$  так, что (iii) остаётся в силе. Однако, это следует из обобщённой теоремы о кучах (трм.5.8).  $\square$

**Замечание:** Из (i) следует:  $c \not\leq b \wedge c \neq a : a \mapsto b \circ c \mapsto d \equiv \underline{false}$ , и  $a \mapsto b \circ a \mapsto d \equiv \underline{false}$  в силе. Очевидно, если имеется выбор, то безразлично какие две вершины связывать первыми – результат тот же самый, благодаря *сходимости* из-за (ii) свойства, которое доказывается позже в лем.5.11.

**Замечание:** Замкнутость (i) показывает на *свойство неповторимости* подструктурных логик (ЛРП рассматривается как такова), которое остаётся в силе и будет продемонстрировано позже.

**Теорема 5.11** (Абельская группа конъюнкции).  $G = (\Omega, \circ)$  является Абельской группой.

*Доказательство.* Из-за лем.5.10  $G$  является моноидом. Поэтому нам остаётся показать: (i) существование обратного к любому элементу из множества носителя графа кучи, так, чтобы соблюдалось:

$$\forall \omega \in \Omega. \exists \omega^{-1} \in \Omega : \omega \circ \omega^{-1} = \omega^{-1} \circ \omega = \varepsilon \quad (5.1)$$

и (ii)  $\circ$  является коммутирующим оператором.

Начнём доказательство с (ii): в базисном случае  $\langle loc_1 \mapsto var_1 \circ loc_2 \mapsto var_2 = loc_2 \mapsto var_2 \circ loc_1 \mapsto var_1 \rangle$  индуктивного опр.3.9 равенство, очевидно, соблюдается. Также соблюдается индуктивный случай до тех пор, пока условия от  $\circ$  соблюдаются. Условие индукции можно получить, рассмотрев рисунок 5.5, если на данный момент предположить, что для любых двух связанных куч оператор  $\circ$  коммутирует. Но, как только, речь идёт об абстрактных предикатах, понятие  $\circ$ -связанных термах может ограничиваться границами предикатов и не могут быть разбросанными как угодно, как подцель в абстрактных предикатах. С практической точки зрения, это не страшно, а наоборот, призывает к лучшей модульности, но с этим необходимо считаться при написании спецификации.

Доказательство продолжается показом свойства (i). Чтобы доказать обратимый элемент, это когда куча, существует, нужно задаться вопросом: а что с практической точки зрения означает «*обратимая куча*»? Когда речь идет о естественных числах, то обратимостью сложения будет вычитание на том же множестве носителя. То же самое происходит для поля комплексных чисел, которые являются расширением поля вещественных чисел. И хотя вещественные или комплексные числа непериодичны, тем не менее, на практике расширение арифметических полей приводит к значительному упрощению вычислительных задач. Хотя ответ отсутствует для множества вещественных чисел на вопрос: «*а что такое  $i$* »? При решении целого ряда задач всё равно  $i$  может быть полезным, зная о равенствах:  $i^2 = -1$  и  $e^{i\pi} = -1$ . В связи с этим, было бы справедливо поставить вопрос: почему бы не предположить, что на данный момент существуют кучи, и мы постулируем урав.5.1, хотя бы до тех пор, пока не будет доказано обратное?

Итак, что *интуитивно* подразумевается под *обратимой кучей* или «*кучей инверс*»? Если речь идёт об естественных и вещественных числах, то имеется модель числовой оси: числа возрастают/уменьшаются относительно нуля в зависимости от направления оси. Как быть с кучами, например, с обыкновенными, формой  $a \mapsto b$ ? Можно ли обратимой куче (инверсия) присвоить инверсию сопровождаемого предиката? — Это будет не точно. Может ли быть присвоено инверсии кучи пустое значение — возможно это будет не правильно, так как любая не пустая куча будет определяться как пустая. Как же будет определяться инверсия пустой кучи, и т.д.? Такое наивное определение тоже не целесообразно, так как не полностью определено для всех куч. Что, если стороны граней в графе куч просто поменяют стороны, например, из  $a \mapsto b$  становится  $b \mapsto a$ ? Это является лишь интересной идеей, но не практикуемо, потому, что конъюнкция не противоречит такому определению, а надо, чтобы при конъюнкции получалась пустая куча. При таком подходе не определена левая сторона в случае объекта. Можно представить, что слияние кучи удаляет «*положительную кучу*», если её соединить вместе с «*отрицательной кучей*». То есть, инверсия означает «*трансцендентную*» операцию удаления кучи из памяти, при этом оператор может быть применен к любой, в том числе и сложной куче. Можно обозначить  $(a \mapsto b)^{-1}$  как « *$a$  не указывает на  $b$* » или лучше, как « *$a$  инверсно указывает на  $b$* ». Первое объяснение неудачное потому, что «*не указывает*» по ошибке может расталкиваться, как  $a \mapsto c$ , при этом  $\exists c \in \Omega, c \neq b$  — это подразумевает, что вершина  $a$  всё-



таки существует и более того, между  $a$  и некоторым элементом, чьё содержимое отличается от  $b$  — всё это ложь, потому, что таких предположений не имеется. Поэтому, гипотетичный случай «*инверсно указывает*», точнее «*инверсно имеется ссылка на*», требует некоторое удаление, несмотря на странное значение, которое позволило бы чистое удаление всех «*лишних элементов*», которые требуют дальнейшую проверку. На первый взгляд это выглядит странно. Пока что, мы специфицировали только части кучи, которые реально существуют. Ввод инверсии теперь меняет ситуацию. Инверсию также нельзя путать с тем, что не должно быть в куче и это есть отрицание предиката, а инверсия кучи — это операция. В данный момент, мы допускаем и концентрируемся на урав.5.1.

Данное уравнение означает, что «*отрицаемое указывает на*»  $a \not\mapsto b$  — это прежде всего предикатное отношение между  $a$  и  $b$ , а в обобщённой форме отрицательная куча  $H^{-1}$ , для которой в силе по определению:  $a \mapsto b \circ a \not\mapsto b = \underline{emp}$  а также  $a \not\mapsto b \circ a \mapsto b = \underline{emp}$ , а более обобщённо  $H \circ H^{-1} = H^{-1} \circ H = \underline{emp}$ . Это означает, что  $\omega \circ \omega^{-1}$  «*чисто*» удаляет кучу, т.е. при необходимости также удаляется ненужная грань и вершина из графа кучи, если больше не остаётся входящих или выходящих граней касательно ещё существующих вершин графа. Таким объяснением данное равенство об обратимости ссылки становится понятным и сейчас не трудно проверить равенство  $H \circ H^{-1} \circ H \equiv H$ . В примере на рисунке 5.6 до применения инверсии состояние кучи является таким:  $d \mapsto a \circ a \mapsto b \circ c \mapsto b$ , но когда применяется инверсия  $\circ(a \mapsto b)^{-1}$ , то получаем  $d \mapsto a \circ a \mapsto b \circ c \mapsto b \circ (a \mapsto b)^{-1}$ , что равно к  $d \mapsto a \circ a \mapsto b \circ (a \mapsto b)^{-1} \circ c \mapsto b$  равно к  $d \mapsto a \circ c \mapsto b$ , что не полностью очевидно, т.к. оба указателя не пересекаются. То есть, такое состояние пока очищено не полностью и нуждается в дополнительных шагах для исправления ненужных элементов. Поэтому, следует пройти ещё два шага нормализации.

**Нормализация – Первый шаг:** Шаг является генеричным (общим). Если между рассматриваемыми графами куч существует мост, как единственная связь между ними, то оператор должен быть заменён на дизъюнкцию (см. позже).

Теперь, когда обнаружен *мост* между  $a$  и  $b$ ,  $\circ$  заменяется на  $\parallel$  в оставшихся терме. Результат можно снова считать обоснованным. Но, возможно, ради полноты вершины, должны быть полностью удалены из соответствующего графа кучи. Это требуется тогда, когда речь идёт о локациях объектных полей.

**Нормализация – Второй шаг:** Удаление вершины  $a$  может быть произведено полностью, когда больше не имеются ссылки на  $a$  в оставшихся графе куч.

Применив эти два шага нормализации, можно избежать проблему упомянутых исключений (ср. обобщённые кучи с набл.4.13). □

**Замечание:** Обобщённые кучи не были обсуждены. Для доказательства корректности необходимо показать  $H \circ H^{-1} \equiv \underline{emp}$ . Доказательство нужно проводить индуктивно над  $\circ$  при использовании  $(g_1 \circ g_2)^{-1} \equiv g_1^{-1} \circ g_2^{-1}$ , так, что существует гомоморфизм для « $\cdot^{-1}$ » касательно  $\circ$  (см. лем.5.13).

**Соглашение 5.12** (Обратимость кучи). Условие (i) подразумевает частичный случай  $\underline{etr} \circ \underline{etr}^{-1} \equiv \underline{etr}$ , так как мы согласуем  $\underline{etr}^{-1} \equiv \underline{etr}$ .

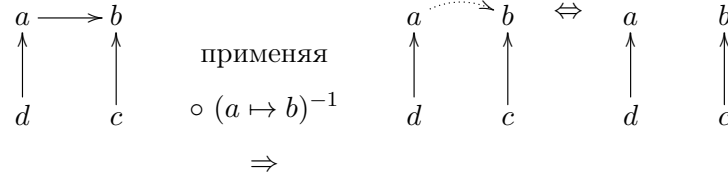


Рисунок 5.6: Граф кучи до и после инверсии

**Объяснение:**  $H \circ a \mapsto b \circ (a \mapsto b)^{-1}$  означает:

1. удалить грань между  $a$  и  $b$
2. удалить вершину  $a$ , если на неё в графе  $H$  не имеется больше ссылок
3. также удалить вершину  $b$ , если на неё в графе  $H$  не имеется больше ссылок

Свойства группы позволяют нам установить равенства над термами куч, например, для ускорения сходимости доказательства или для преобразования в нормальную форму (см. трм.5.3). Таким образом, можно будет проводить сокращение раздутых термовых представлений куч. Частичные спецификации разрешат сокращённое представление правил (см. раздел 5.5). Дальнейшая работа включает в себя подключение решателей для преобразования простейших термов куч. Необходимо рассмотреть случай, когда содержимым является указатель, например,  $a \mapsto o.f \circ (a \mapsto o.f)^{-1}$ , где  $o$  объект содержащий поле  $f$ . Очевидно,  $o.f$  не удаляется из динамической памяти, иначе, целостность объектных экземпляров нарушается — дальнейшее исследование приветствуется, но находится за пределами этой работы (см. дискуссию в разделе 4.6). Поэтому, по умолчанию уговаривается, что поле остается в памяти как единица целого объекта, но ссылается на `nil`. Таким образом, замеченные шаги остаются без изменения в силе.

**Лемма 5.13** (Гомоморфизм об обыкновенных кучах).  $(g_1 \circ g_2)^{-1} \equiv g_1^{-1} \circ g_2^{-1}$  действительно для любых не сложных куч  $g_1$  и  $g_2$ .

*Доказательство.* Если нам удастся доказать более обобщённую форму:  $G = g_1 \circ g_2 \circ \dots \circ g_n$ , то проблема будет решена. Для этого необходимо показать  $G \circ G^{-1} = \underline{etr}$ . Доказать это можно индуктивно, используя  $n$ . В базисном случае ( $n = 1$ ) получаем  $g_1 \circ g_1^{-1} \equiv \underline{etr}$ , что естественно в силе из-за необходимости существования обратного элемента. Для индуктивного случая предположим:

$$G = \underbrace{(g_1 \circ g_2 \circ \dots \circ g_k)}_{G_k} \circ g_{k+1}$$

тогда для

$$G \circ G^{-1} = (G_k \circ g_{k+1}) \circ (G_k \circ g_{k+1})^{-1}$$

обратная часть кучи является настоящим расширением кучи. Правая часть равенства равна

$$\underbrace{G_k \circ G_k^{-1}}_{\underline{emp}} \circ \underbrace{g_{k+1} \circ g_{k+1}^{-1}}_{\underline{emp}} \equiv \underline{emp}$$

(из-за индуктивного свойства обратимости, соблюдая конв.5.12).  $\square$

**Определение 5.14** (Дизъюнкция кучи). *Дизъюнкция кучи  $H \parallel a \mapsto b$  определяет кучу  $H$  и не сложную кучу  $a \mapsto b$ , которая не пересекается, тогда, если  $G_H$  является графом кучи  $H$ , тогда  $G_H = (V, E)$ , для всех граней  $(\_, a) \notin E$  и не существует пути от  $b$  до  $H$ , а также не существует обратного пути от  $H$  до  $a$ .*

Поэтому,  $x.b \parallel x.c$  не действительно для любого объекта  $x$  с полями  $b$  и  $c$ , если существует, хотя бы одна общая вершина для любого пути, начиная с  $x.b$  или  $x.c$ .

Допустим,  $\Sigma = X_0 \parallel X_1 \parallel \dots \parallel X_n$  с  $n > 0$  и  $X_j$  имеет форму  $x_j \mapsto y_j$ , тогда  $\Sigma = \Sigma_0 \parallel a_0 \mapsto b_0 \Leftrightarrow \forall (a_j \mapsto b_j) \in \Sigma_0 : a_j \neq a_0 \wedge b_j \neq b_0$ .

**Теорема 5.15** (Моноид дизъюнкция).  *$G = (\Omega, \parallel)$  является моноидом и группой, если  $\Omega$  является множеством графов куч и  $\parallel$  является дизъюнкцией кучи.*

*Доказательство.* В аналогии к предыдущей лемме, прежде всего,  $\forall m_1, m_2 \in \Omega : m_1 \parallel m_2$ , только тогда, когда  $m_1$  и  $m_2$  не имеют общую вершину. Это всегда так, когда нет пути от  $m_1$  до  $m_2$  и не существует граф окружающий оба,  $m_1$  и  $m_2$ . Если  $m_1$  и  $m_2$  различны, то  $m_1 \parallel m_2$  снова являются годной кучей в  $\Omega$ , потому, что  $m_1$  от другой части графа кучи, чем  $m_2$  и наоборот, поэтому следует замкнутость. Ассоциативность следует очевидно.  $\underline{emp}$  может послужить нейтральным элементом, тогда  $\underline{emp} \parallel m_1 = m_1 \parallel \underline{emp} = m_1$ . Пусть по определению  $\underline{emp} \parallel \underline{emp} = \underline{emp}$  будет в силе. Последним, уговаривается  $s \parallel s^{-1} = s^{-1} \parallel s = \underline{emp}$ . Это похоже на  $\circ$ . В общем, кучи следуют этому правилу.  $\square$

Конъюнкция и дизъюнкция частей кучи могут быть выражены правилами с помощью дуальных к нему правил следующим образом:

$$\circ_{[B,C]} \frac{U \circ B \parallel C}{U \circ B \circ C} \qquad \parallel_{[B,C]} \frac{U \circ B \circ C}{U \circ B \parallel C}$$

$$\parallel_{[B,C]}; \circ_{[B,C]}; \parallel_{[B,C]} \equiv \parallel_{[B,C]} \quad (5.2)$$

$$\circ_{[B,C]}; \parallel_{[B,C]}; \circ_{[B,C]} \equiv \circ_{[B,C]} \quad (5.3)$$

Операции  $\parallel$  и  $\circ$  — дуальные, и они могут быть преобразованы друг в друга, используя дуальную операцию, получив из урав.5.2 и урав.5.3, где «;» оператор последовательности.

Равенства в силе, из-за самообратимости операции и поставленного утверждения о существовании обеих вершин кучи  $B$  и  $C$ .

**Теорема 5.16** (Дистрибутивность). *Дистрибутивность в силе для  $\forall a, b, c \in \Omega$  для  $\circ$  и  $\parallel$ :*

$$(i) \quad a \circ (b \parallel c) = (a \circ b) \parallel (a \circ c)$$

$$(ii) \quad (b \parallel c) \circ a = (b \circ a) \parallel (c \circ a)$$

*Доказательство.* Доказывается только (i), т.к. (ii) следует непосредственно из дуальности  $\circ$  и  $\parallel$ . Равенство (i) доказывается двумя импликациями. « $\Rightarrow$ » обозначает, что из  $a \circ (b \parallel c)$  следует  $(a \circ b) \parallel (a \circ c)$ . « $\Leftarrow$ » означает импликацию в обратную сторону.

« $\Rightarrow$ »:  $b \parallel c$  подразумевает связи между  $b$  и  $c$  не существует. Следовательно,  $a \circ (b \parallel c)$  означает, либо между  $a$  и  $b \parallel c$  связь тоже отсутствует, что приведёт к false, либо одна соединяющая вершина существует и, по определению, она должна находиться, либо в  $b$ , либо в  $c$ . Без ограничения общности предполагается соединяющая вершина в  $b$ , тогда  $a \circ b$  в силе. Аналогичное может следовать из  $c$ . Как бы ни было, но один из графов расширяется:  $b$  или  $c$ , а другой не расширяется. Поэтому, второй граф всегда будет false.

« $\Leftarrow$ »: Либо  $a \circ b$ , либо  $a \circ c$  равны false из-за опр.5.7 потому, что  $a$  не может связываться одновременно с  $b$  и  $c$ , иначе имеется противоречие опр.5.14. Без ограничения общности можно утверждать, что если  $a \circ b$  равно true, то  $a \circ (b \parallel c)$  тоже.  $\square$

**Замечание:** Так как нейтральным элементом для операций  $\circ$  и  $\parallel$  является emp, то нельзя определить (алгебраическое) поле (например поле Галуа) над этими операциями из-за того, что лем.5.10, лем.5.11 и лем.5.16 в силе и несмотря на то, что множество носителем  $\Omega$  конечное, поэтому любая куча конечная и все операции над ними также производят конечную кучу.

**Замечание:** В аналогии к логическим конъюнктам  $\wedge$  и  $\vee$ , нормализованная форма с помощью  $\parallel$  всегда существует, применив ранее упомянутые равенства и лем.5.13 для того, чтобы произвести инверсию обобщённых куч.

Для оптимизации логического вывода с помощью уменьшения объёма графа, необходимо попытаться вывести операцию  $\parallel$  как можно дальше наружи в термах кучи, применив дистрибутивность, либо переставляя несложные кучи так, чтобы левые части ссылок были отсортированы по имени локации по лексикографическому порядку. Идея заключается в сокращении повторных поисков, например, для *пошаговой верификации*, так, что только меняющиеся части кучи принадлежат повторному вычислению.

Частично упорядоченное множество (ЧУМ) можно установить над графами для подмножества графов, операторам слияния ЧУМ служит  $\circ$ . Минимальный элемент является пустой кучей, а мак-

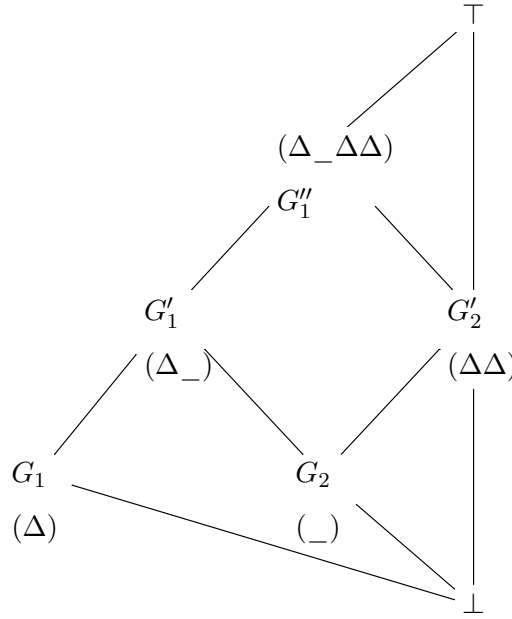


Рисунок 5.7: Упорядоченное множество над графами куч

симальный является полным графом кучи. Например, закон *абсорбции* не действительный для куч. ЧУМ не может быть полной решёткой. На рисунке 5.7 ЧУМ  $G$  содержит  $\{G_1, G_2, G'_1, G'_2, G''_1\}$ , которые соблюдают следующие неравенства по возрастающему порядку  $G_1 \sqsubseteq G'_1 \sqsubseteq G''_1$ ,  $G_2 \sqsubseteq G'_1$  и  $G_2 \sqsubseteq G'_2 \sqsubseteq G''_1$ . Кучи  $\perp$  и  $\top$  всегда присутствуют и при необходимости могут быть добавлены, поэтому каждый раз по умолчанию могут быть не указаны.  $G''_1$  является максимальным элементом, а  $\inf(G) = \underline{emp}$  минимальным элементом, где  $\sqsubseteq$  определяет подмножественное соотношение графа. Нетрудно убедиться в том, что две не соединённые кучи при конъюнкции (т.е. равны emp из-за опр.5.7) в соответствующей диаграмме Хассе всегда остаются несвязанными. Слияние всегда верное, потому, что: (первое противоречие)  $a \mapsto b \circ a \mapsto d$  не может следовать первой конъюнкции, либо другой сложной куче. Из-за (второго противоречия)  $a \mapsto b \parallel b \mapsto d$  противоречит определению  $\parallel$ . Однако, необходимо учесть, что порядок ЧУМ может пострадать после ввода инверсии кучи (ср. ранее), если использовать инверсию без соблюдения ограничений. На данный момент нас вполне устраивает использование инверсии для «более удобного» сравнения куч, в частности с проверяемой спецификацией, поэтому свойство локальности из главы 3 остаётся.

## 5.4 Классовый экземпляр как куча

Рассмотрим подробнее, как объектные экземпляры классов моделируются как граф куча согласно рисунку 5.8.

На рисунке 5.8 а) отдельное присвоение к существующему графу представлено (без подробностей), объект в прямоугольной рамке является некоторым объектным экземпляром. На рисунке 5.8

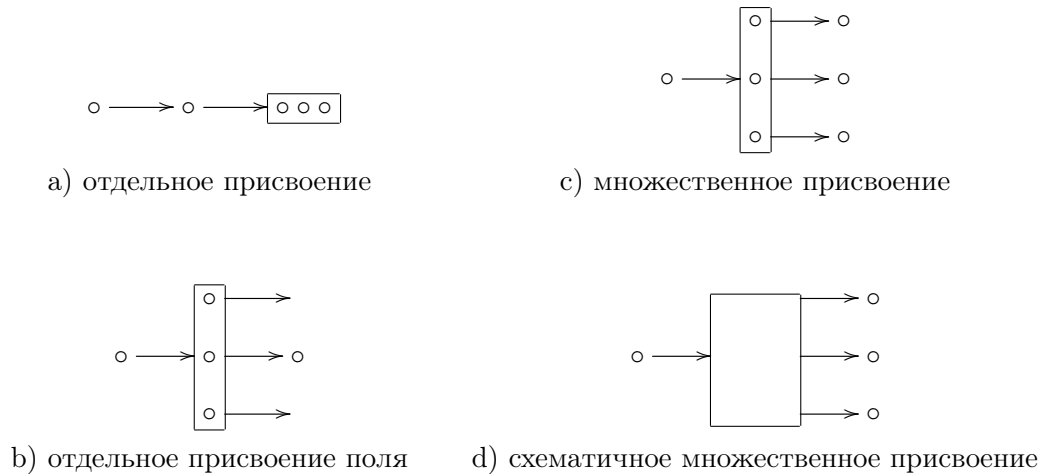


Рисунок 5.8: Формы присвоения объектных экземпляров

б) только второе поле объектного экземпляра присваивается некоторое значение сходимого типа данных, все остальные поля присвоены `nil`. Рисунки 5.8 с) и 5.8 д) имеют тоже самое значение как и с), только присвоение производится для всех полей, например по *лексикографическому порядку*. Далее вводим дополнительные ограничения, но без ограничения общности, которые будут полезными для, позже вводимых, операторов:

**Соглашение 5.17** (Моделирование объектных экземпляров). *Объектные экземпляры моделируются без ограничения общности как:*

- а) Нет внутренних объектов. Объекты внутри иного объектного экземпляра запрещаются. Внутренние объекты всегда могут быть смоделированы как отдельные, но ассоциированные объекты. Поэтому, разрешаются ссылки только к тем объектам, которые не пересекаются. Таким образом, структуры *union* исключаются в моделировании объектных экземпляров.
- б) Объектные поля различаются и могут быть использованы другими объектами и определены в наследованных классах. В случае конфликта с наименованием в иерархии наследования классов, конфликтующее наименование ближе к корню по иерархии наследования может быть переименовано так, чтобы все ссылки на это поле были также переименованы без ограничения общности.
- с) Из-за замкнутости, во время существования объекты не растут, кроме случаев, когда сам указатель меняется и ссылается на новый объект различного типа. Многочисленные объекты располагаются в динамической памяти. Проблема с фрагментацией памяти возникает из-за ограничения памяти. В целях избежания фрагментации, можно попытаться преобразовать динамически выделенные объекты в автоматически выделенные переменные, т.е. разместить их в стек. Если предположить, что в подклассах поля могут оставаться, либо

могут исчезать, то объектные экземпляры могут только расти, а следовательно и регионы памяти тоже растут. Увеличение может привести к серьёзным проблемам, решением которых, может послужить утилизация и заново выделенная новая динамическая память для её перемещения. В общем, проблема не решаема из-за неразрешимости проблемы приостановки. Однако, если статически в частных случаях удастся решить вопрос о максимальных лимитах, то можно выбрать более оптимальный вариант. В реальности необходимо учитывать, что поля объектов также могут исчезать в связи с наследованием классов, поэтому отсутствует монотонность количества полей объекта, но имеется статический размер занимаемой памяти, если исключить позднее присвоение. Присвоения любых объектов одного класса, либо подклассов запрещается. Из-за того, что разделяющие кучи, представленные Бурстоллом [51] следуют не сжимаемости, объектные поля не могут повторяться более одного раза в одной и той же конъюнкции для одного выражения кучи в качестве локации.

- d) Массивы в качестве базисного типа исключаются. Также исключаются более одной грани между двумя вершинами графа куч, различные поля объектов могут ссылаться на этот же объект.
- e) Общими вершинами графа куч могут быть не только простые вершины, но также сложные объекты. То есть, атрибуты абстрактных предикатов могут быть общими.

Чтобы не противоречить дальнейшим определениям с одной стороны, необходимо иметь возможность быстро проверять соотношения между двумя вершинами. Данная не пустая куча состоит из одной или более того простых куч. Для проверки, связаны ли две вершины графа кучи, необходимо проверить в худшем случае все левые стороны простых конъюнктов, т.е. необходим перебор всех граней графа куч.

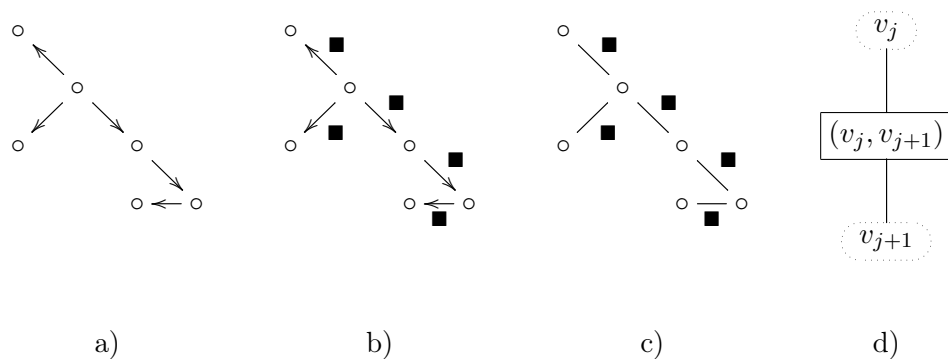


Рисунок 5.9: Преобразование схематического графа кучи

С другой стороны, чтобы получить для одной вершины все соседние вершины, нам выгоднее иметь модель, которую можно было бы итерировать по вершинам, а не по граням. Для быстрого определения соотношения (например соседства), предлагается например, `reaches(x,y)`, `reaches(x,Y)`,

$\text{reaches}(X, y)$ ,  $\text{reaches}(X, Y)$ , где  $x$  является определённой вершиной, а  $X$  является (под-)множеством вершин (аналогичное для  $y$  и  $Y$ ). Модель, основанная на вершинах, отражена на рисунке 5.9 маленькими закрашенными квадратиками, соответственно, позволит получить более эффективную итерацию. Обе модели а) и б), основанные на вершинах и гранях, являются дуальными и они могут быть преобразованы друг в друга: вершины по серединам граней кодируют начало и конец (см. d)), как одна вершина с объединяющим именем и связываются с соседними вершинами (см. рисунок 5.9 а) и б)). Очевидно, преобразование из одной модели в другую и обратно, согласно схеме преобразования на рисунке 5.9 с) возможно без потерь, т.е. отображение между обеими структурами является биективным независимо от направления граней (см. с)).

Поэтому, нам разрешается преобразовать графовое представление с конъюнкциями или дизъюнкциями, так нам удобнее для решения.

**Соглашение 5.18** (Объектные поля). *Объектные поля не пересекаются, поэтому указатели содержимого полей имеют различные адреса. Однако, указатели могут иметь псевдонимы. Объектное содержимое может быть выражено как  $x \mapsto \text{object}(fld_1, fld_2, \dots)$ . Без ограничения общности по умолчанию согласуется, что объектные поля не могут быть использованы в качестве содержимого через некоторый указатель в произвольных арифметических выражениях, но только путями, используя локацию согласно набл.5.9. Позднее связывание не рассматривается. Отсутствие позднего связывания означает, что полиморфизм с помощью решения нужной инстанции во время запуска программы отсутствует. Это является ограничением. Однако, все поля доступны, включая типы полей, могут быть полностью проанализированы статически. Следовательно, при анализе может быть использован только самый обобщённый класс по иерархии наследования. Решение, какой класс использовать при генерации объектного экземпляра — не решимо в общем, программный оператор ответственный за выполнение генерации может быть доступен или нет. Предсказывать это заранее в общем случае не возможно. Это означает, что при верификации может быть построена только наиболее обобщённая куча.*

## 5.5 Частичная спецификация куч

Как было сказано ранее, в опр.3.9 объектные экземпляры можно рассматривать, как хранители полей данных  $\text{obj}.f_1 \mapsto \dots \circ \text{obj}.f_2 \mapsto \dots \circ \text{obj}.f_n \mapsto \dots$ . Все поля создают некоторую кучу по отрывкам, но в отличие от абстракции, это преобразование можно охарактеризовать как *конкретизацию*. Поля классовых объектов имеют ограничения в связи с наименованием и типами, которые должны совпадать с соответственным классом. Все поля одного объекта существуют независимо друг от друга, поэтому они являются простыми кучами, которые связаны между собой с помощью о-конъюнкции как это уже было упомянуто. Поля объектов не могут быть утилизированы по отдельности (см.



набл.3.5 и последующую дискуссию). Локальные переменные поля объектов также должны иметь возможность специфицировать части, т.е. подклассовые объекты. В аналогии к необъектным указателям можно определить константные функции над кучами, например,  $\underline{true}(obj)$  или  $\underline{false}(obj)$  из опр.3.10 и опр.3.9. В отличие от необъектных указателей,  $\underline{true}(obj)$  вводит объектный терм в качестве дополнительного параметра. Таким образом, абстрактные предикаты могут быть использованы для дальнейшей модуляризации спецификации, которая синтаксически и семантически не значительно отличается от необъектного случая.

В соответствии с предложенным методом распознавания решателя  $a \circ a$  на основе вталкивания и выталкивания в/из стека согласно данному уровню абстрактного предиката (см. главу 6), теперь входящие и выходящие термы абстрактных предикатов, возможно, отслеживать и пропускать при повторных или ненужных вычислениях. Сравнение может производиться стеком для одинакового уровня, либо отметками между любыми уровнями предикатов с помощью транслирующих правил.

**Определение 5.19** (Неполные предикаты).  $\underline{true}(obj)$  определяет кучу  $\circ$ -конъюнктов всех полей объекта  $obj$  (включая пустую кучу, которая представляет собой пустой объект). Все поля, которые явным образом специфицируются, высчитываются первыми из множества всех оставшихся полей. При верификации все поля подразумеваются, когда ссылаясь на константные функции, которые явно не были специфицированы в рассматриваемом абстрактном предикате. Высчитывание всех полей для данного объекта может быть реализовано с помощью стекового окна для каждого уровня абстрактного предиката, который соответственно связан с определённым объектом.

Подробности следуют в главе 6. Нетрудно убедиться в верности  $\wedge, \vee, \neg$ -конъюнкций для  $\underline{true}(obj)$  и  $\underline{false}(obj)$ . Эти константные функции могут принимать любое число объектных полей (см. трм.5.3) и их булево обозначение не зависит от конкретных полей. Однако, сложные кучи в комбинации с ними, могут иметь неожиданное поведение, которое исключается при анализе контекста. Частичные спецификации, которые используют константные функции, частично описывают поля, но могут покрывать гораздо больше. Таким образом, спецификации частичные, но охватывают больше куч (см. трм.5.4). Более того, высчитывание позволяет сравнивать имеющие кучи и выявить те кучи, которых не хватает. Это позволяет полностью определить все входные кучи в правилах.

**Пример 5.20** (Неполный предикат №.1). Дан объект  $a$ , который имеет три поля  $f_1, g_1$  и  $g_2$ .  $C[\cdot]$  означает семантическую (неявную) функцию над термами куч типа  $ET \rightarrow ET \rightarrow \mathbb{B}$ , где  $ET$  из опр.3.10,  $\mathbb{B}$  булево множество, где первая расширенная куча является ожидаемой, а вторая является полученной кучей, то тогда:

$$\begin{aligned} C[a.f_1 \mapsto x \circ \underline{true}(a)] &= C[a.f_1 \circ a.g_1 \circ a.g_2] \\ &= C[\underline{true}(a) \circ a.f_1 \mapsto x] \neq C[p(a) \circ a.f_1 \mapsto x] \end{aligned}$$

где,  $p$  абстрактный предикат означает  $\underline{true}(a)$  (см. опр.5.19).

Однако,  $C[a.f_1 \mapsto x \circ p(a)]$  означало бы равенство потому, что благодаря распознаванию, используя актуальное стековое окно, находит все оставшиеся поля, если даже ниже спрятано несколько уровней абстрактных вызовов.  $C[\cdot]$  является гомоморфизмом касательно обсуждённых константных функций и конъюнкции  $\circ$ .

**Пример 5.21** (Неполный предикат №.2).  $C[\underline{true}(a) \circ \underline{true}(a)] = C[a.f_1 \circ a.g_1 \circ a.g_2] \circ C[\underline{true}(a)] = C[a.f_1 \circ a.g_1 \circ a.g_2] \circ \underline{emp}(a)$ .

## 5.6 Обсуждения

Одна пространственная операция была заменена на две строгие. Изначальные основные свойства ЛРП не поменялись, кроме неограниченной инверсии кучи, которая тщательно обсуждалась ранее. Если в трм.3.6 заменить « $\star$ » на « $\circ$ », а в случае дизъюнкции « $\star$ » на « $\parallel$ », то верность аксиом следует непосредственно, ради исключения аксиомы №5 для конъюнкции как раз из-за ужесточения « $\star$ ».

Имея форму, которая позволяет нормализовать термы кучи, специфицируемые кучи можно теперь анализировать линейно (см. с разделом 5.5). Из-за требования о неповторимости простых куч, комплексные кучи могут быть эффективно исключены с помощью «*мемоизатора*» (с англ. «*memoizer*»). На практике, необходимо исключать и обнаруживать повторяющиеся локации (возможно с различными, но одинаковыми значениями), как это было упомянуто в разделе 5.3. Иначе, кучи и граф кучи становятся противоречивыми и свойства ЛРП нарушаются. Это приведёт к полной неповторимости теорем. Принцип должен соблюдаться: одна простая куча специфицируется один раз. Простая куча, специфицируемая в одном месте не должна специфицироваться заново, аналогично принципу локальности: одно изменение локально производится только в одном месте. Если принципы не верны, то и применение правил не верное, а следовательно, из ложного предположения могут выводиться любые результаты, поэтому повторные кучи должны исключаться.

Однако, эту проблему в общем удастся решить только динамически, но без запуска программы. «*Динамически*» при статическом анализе означает, что во время верификации, абстрактные предикаты естественно могут быть произвольными. Абстрактные предикаты могут интерпретироваться процедурально (см. главу 6). Следовательно, используется *стековая архитектура* для их анализа (ср. главу 4). Она очень близка к операционной семантике предложенной Уорреном [266], где стек имеет ссылки на предыдущие стековые поля. Эта семантика отличается от семантик классических языков программирования, которая ради исключения, при использовании параметров по вызову, более близка к реализации логического языка Пролог. Также представление и переход предикатов более похожи на правила Пролога. Незначительно модифицируемая семантика машины Уоррена может быть применена к интерпретации абстрактных предикатов, с помощью строгих операций конъюнкции и дизъюнкции, с помощью которой удастся распознавать, например,  $\forall a \in \Omega. a \circ a$  для

соблюдения неповторимости.

Мемоизатор может кэшировать только те вызовы абстрактных предикатов, которые не меняют глобальные состояния. Мемоизатор может запоминать, что имеются (i) входные или (ii) выходные или (iii) входные-выходные переменные термы. Если далее, неограниченные символы внутри одного абстрактного предиката поздним подвызовом ограничиваются, то это необходимо учесть при порядке вычисления подвызовов (определение должно следовать порядку слева направо, чтобы разрешить подобные конфликты, см. главу 6).

Пролог [248], как общий логический язык программирования может быть использован (см. тез.4.11), как платформа, основанная на рекурсивно-процедуральных правилах и термах для логического вывода, используя теоремы о расширенных термов куч и абстрактных предикатов. В Прологе обобщённая схема рекурсивной индукции Пиано может быть всегда определена как « $p(0).$ » для базисных случаев, а « $p(n) :- n1 \text{ is } n-1, p(n1).$ » для индуктивных случаев, используя вспомогательный предикат `is` для высчитывания `n1`. Не трудно убедиться в том, что в Прологе можно выразить любую обобщённую  $\mu$ -рекурсивную схему предикатов. В общем, предикаты могут быть не определены (например, когда процедурный вызов не приостанавливается). Преимущество правил Хорна Пролога, в отличие от классических разовых функций (как они были использованы, например в [193]), это способность рассматривать термы предикатов как (i),(ii) или даже (iii), объединившие таким образом экспоненциальное количество различных классических разовых функций. Не каждый аспект может быть определён, поэтому вопрос об обратимости функции требует дополнительного внимания. Арифметические вычисления, а также *зелёные* и *красные отсечения* [248] поисковых пространств, являются одной возможной причиной, почему предикат может быть не обратим. Арифметические выражения в Прологе вычисляются с помощью оператора `is`. Обратимость арифметических выражений можно частично восстановить, если заменить натуральные числа Чёрческими числами (см. [298]), а фундаментальные операции обосновать только, если использовать константу, *одинарный функтор* и унификацию, как универсальную *монаду* базовой операции. Говоря обобщённо, необходимо гарантировать сильную корреляцию между входным и выходным результатом, которая должна стать изоморфизмом отображением для полной обратимости. Более того, прологовские отсечения могут быть заменены без потери общности и выразимости, так как отсечения являются лишь синтаксическим сахаром (см. главу 3).

Язык «*Object Constraint Language (OCL)*» [1] является языком спецификации для объектных экземпляров. «*OCL*» является расширением языка «*UML*», и существует, как графическая нотация для утверждений, либо как формулы. Формулы «*OCL*» выражают часть предикатной логики. Имеется: — поддержка квантификации переменных, поддержка массивов и абстрактных типов данных/классов и полиморфизм с помощью подклассов. «*OCL*» разрешает описывать цикл жизни объектов и методов. Однако, «*OCL*» не знает об указателях или псевдонимах. Утверждения об указателях от-

существует, поэтому на этапе моделирования и быстрой прототипизации имеются ограничения, как было описано в главе 1 (см. [297], закл.5.5).

Предлагается, чтобы указатели записывались в качестве множества наименований объектного экземпляра. Таким образом, псевдонимы записаны в одном месте, альтернативно вводится множество указателей, которое существует независимо от существующих экземпляров объектов. Состояние объектов совпадает с состоянием вычисления стека/кучи. Сложные объекты имеют поля, которые просты или ссылаются на любые другие объектные экземпляры. Абстрактный предикат предлагается логическим предикатом (см. главу 4), возможно, с символами. Нет обязательства, что в одном предикате описывается только один объект. Но рекомендуется определять одним абстрактным предикатом один объект целиком зависевшие объекты рекомендуется описывать отдельными предикатами. Ситуация, когда одним предикатом описываются два или более того объектных экземпляров не исключена, но не приветствуется ради модульности. Пространственное соотношение между объектами описывается операторами  $\circ$  и  $\parallel$ . Соотношения не требуют обязательного обозначения пространственности, если из контекста ясно, что речь идёт расширении представления «*OCL*».

Будущая работа может заключаться в расширении с абстрактными предикатами [295]. Начатое предложение следует расследовать дальше, особенно, учитывая постановления из конв.5.17 и опр.3.10. Ожидается повышенная выразимость, модульность и абстракция. Хороший обзор нынешних попыток расширить существующие вычисления с указателями, можно найти в главе 1, а также в [193].

## 6 Автоматическая верификация с предикатами

Рассмотрим простой пример из области вычислительной геометрии [29]. *Двусвязный список* содержащий грань данного многогранника, послужит примером, в котором каждая грань связывает две вершины некоторого 2 или 3-мерного пространства. Сеть многогранника после триангуляции распадается на треугольники. Каждая вершина дважды связана с двумя соседними вершинами. Каждая грань начинается и заканчивается определёнными гранями. Для определения нормального вектора, достаточно иметь треугольник. В простой куче локализатор указывает на содержимое. Для данного примера это могут быть вершины или грани. Согласно упомянутому двусвязному списку, каждый элемент имеет ссылку вперёд к следующей грани и назад к предыдущей грани. Простая куча содержит связанные между собой грани. Не связанные между собой грани по определению не указываются. Уговаривается, что принудительное отсутствие кучи (см. главу 5) исключается, либо отдельно не рассматривается в связи с Абельской группой из-за ранее упомянутых ограничений. Представим себе, что каждый раз как ссылаться на указатели, копии всех трёх вершин заносятся в память. Нетрудно убедиться в том, что такой подход малоэффективен. Если работать с указателями, то эта проблема отпадает, особенно когда вводится дополнительный уровень абстракции. Эти абстрактные предикаты позволяют более интуитивно выражать сложные кучи. Например, прологовская подцель  $\text{face}(p1, p2, p3)$  может означать, что три вершины  $p1, p2, p3$  связаны между собой в одном треугольнике вместо того, чтобы каждый раз полностью специфицировать  $\exists v1.v2.v3$ , при  $p1.data \mapsto v1 \star p2.data \mapsto v2 \star p3.data \mapsto v3 \star p1.next \mapsto p2 \star p2.next \mapsto p3 \star p3.next \mapsto p1 \star p1.prev \mapsto p3 \star p3.prev \mapsto p2 \star p2.prev \mapsto p1$ .

Прежде всего, абстракция означает обобщённость, вводя дополнительные параметры. Под абстрактным предикатом подразумевается правило Хорна с произвольным количеством параметров. Хотя некоторые авторы стремятся использовать «*Абстрактный Предикат*» как новый термин [195], нужно заметить, что абстракция не нуждается в дополнительном определении (см. главу 3), как новой концепции. Аналогично распространяется и на предикаты. Ясны концепции абстракции и предикатов, поэтому считается не целесообразно заново обозначать термины, тем более, имеются случаи, когда к предикатам добавляются параметры, например, в логике предикатов первого порядка. Это и является причиной, почему предикаты по-прежнему рассматриваются как классиче-

ские, а «абстрактные» как прилагательные к предикату. «Абстрактный Предикат» не отличается от термина предиката, поэтому отдельно взятое семантическое определение просто не существует. Абстрактный предикат имеет любое (включая ноль) количество термовых параметров и может содержать любую последовательность (включая ноль) подцелей ранее декларированных предикатов. В данном примере  $\text{face}(p1, p2, p3)$  равен той самой развёрнутой  $\star$ -формуле подцелей, которая была указана ранее. В зависимости от того, в каком состоянии находится вычисление подцели  $\text{face}$ , либо свёртывание, либо развёртывание, может быть целесообразным. Предикат  $\text{face}$  может зависеть от других предикатов. Однако, на данный момент просто не достаточно известно о том, когда необходимо свёртывать или развёртывать определение абстрактного предиката. Если развёртывание проваливается, то это может быть потому, что это в принципе не возможно, а также потому, что развёртывание и свёртывание были совершены в не правильный момент, либо в неправильной последовательности. Далее, будет рассматриваться новый подход, который позволит решить проблему автоматизации с кучами.

Уоррен [266] использует термин «программирования через доказательство» для того, чтобы выразить, что Пролог может быть использован как язык программирования, который используется для нахождения решения формулированного запроса правил Хорна. Изоморфизм *Карри-Хауарда* [179] гласит о взаимосвязи между доказательством и программированием. Касательно куч, философский лозунг данного подхода можно охарактеризовать как «доказательство является проблемой синтаксиса», это означает, что с помощью синтаксического перебора можно доказать корректность специфицируемой кучи и представительство кучи близко к моделям по программированию, т.е. к прологовским правилам. Главным наблюдением этой главы является: абстрактные предикаты описывают на самом деле формальный язык (см. тез.4.11). Позже мы убедимся в том, что формальный язык является логическим языком программирования. Следовательно, проблемы свойств куч, которые являются семантическими, можно решить синтаксическим распознаванием.

## 6.1 Сжатие и развёртывание

Представленный в этой главе подход сильно отличается от существующих традиционных.

По Рейнольдсу [224, 193] пространственный оператор  $\star$  связывает две отдельные кучи, при котором основы *подструктурной логики* остаются в силе, а *правило сужения* не в силе. Как было изложено в главе 3, классический оператор  $\star$  многозначен, поэтому, далее используется только строго соединяющий оператор « $\circ$ » в качестве пространственной конъюнкции.

Абстрактные предикаты задаются пользователем, как это было предложено в «*Verifast*» [124]. Вывод может осуществляться быстрее с помощью «*тактик*», которые могут определяться индуктивно в системе «*Coq*» [32] и выводиться в *полуручном режиме*. Системы основаны на принципе «сжать/распаковать» (fold/unfold) [122], могут при полной подсказке полностью независимо от на-

чала до конца вывод осуществить логически. Подсказки доказательства показывают на тот *редекс*, который необходимо предпринять для выхода из неопределённого состояния и для продвижения верификации в целом.

Пролог здесь используется как язык утверждения. [144] наглядно демонстрирует пригодность к доказательству формул в предикатной логике с помощью правил Хорна. [144] в прошлом предлагал использовать Пролог в качестве языка программирования, что, увы, не всегда возможно из-за вычислимости (см. главу 4). [266] содержит определение реализации логического вывода, опираясь на операционную семантику. Каллмайер [131] демонстрирует использование Пролога для распознавания *морфем* и *мутаций грамматики* в естественных языках. В частности, «*сопряжённые деревья*» предлагаются как механизм обработки мутаций в естественных языках на основе явных определённых  $\lambda$ -термов, которые исключаются в формальных языках, в частности, языках программирования во избежание многозначности. Примеры показывают многозначную перегруженность и трудность анализа из-за экспоненциального роста необходимых проверок посторонних условий. [168] на примерах поэтапно излагает, как Пролог может способствовать к решению проблем многозначности синтаксического перебора. Мэттьюс широко использует рекурсивные распознаватели, которые работают с деревьями и в Прологе реализованы эффективно и просто. Реализации являются стековыми автоматами, распознающие LL(k)-грамматики с модификациями: конечные состояния выделяются явным образом, а рекурсивные прологовские правила имитируются стеком. Мэттьюс использует «*списки разниц*» для реализации распознавателя регулярных языков, который на самом деле основан на «*автомате частичных производимых регулярных выражений*» [50]. Бжозовский предлагает «*прологовские формальные грамматики DCG*» и встроенные команды Пролога для изменения баз знаний во время запуска для увеличения гибкости, которую он считает необходимо расширять. Однако, подход Бжозовского имеет недостаток в том, что выразимость ограничена регулярностью (см. главу 3). Работу Перейры [201] можно оценивать как классическую монографию по Прологу и обработку естественных языков. Однако, подход Перейры имеет фундаментальные ограничения, которые всё-таки легко можно устранить среди множества образцовых и отличающихся примеров. Невозможность разрешить лево-рекурсию правил Хорна, хотя решение в принципе существует, т.к. LL(1)-распознаватель не в состоянии опознать всех предшественников для решения принадлежности правила. Далее, Перейра вводит  $\lambda$ -вычисления над деревьями для распознавания естественных языков. Таким образом, морфемы и лексемы связываются и получают зависимое значение от введённых параметров. Далее, введём первое прототипное определение кучи, учитывая опр.1.8 и главу 3.

## 6.2 Предикатное расширение

**Определение 6.1** (Утверждение о куче). *Утверждение о куче  $H$  индуктивно определено как:*

$$H ::= \text{emp} \mid \text{true} \mid \text{false} \mid x \mapsto E \mid H \star H \\ \mid H \wedge H \mid H \vee H \mid \neg H \mid \exists x. H \mid a(\vec{\alpha})$$

Утверждение **emp** означает пустую кучу, которая верна, если данная куча пуста. Пустая куча является нейтральным элементом относительно пространственным связям между кучами (см. главы 3,5). « $\star$ » разделяет две кучи на две независимые кучи. В этой главе мы не будем различать ужесточение между « $\circ$ » и « $\star$ » (см. главу 5) — ради общности мы исходим из конъюнкции куч. Утверждение **true** означает, любая куча (в том числе пустая) разрешается, а **false** означает, любая куча не разрешается. Эти определения близки к определению по Рейнольдсу [224]. Основой всех определений по Рейнольдсу является *обыкновенная куча*:  $x \mapsto E$ , где  $x$  некоторый *локализатор* (например доступ к объектному полю  $o1.field1$ ), а  $E$  является некоторым допустимым выражением, которое присваивается ячейке памяти обозначаемой локализатором  $x$ . Проверка совместимости типов проводится на более раннем этапе [297]. На данный момент безразлично, явное значение, либо ссылка на ячейку в памяти содержится в качестве содержимого по выражению (см. [51]). Рассмотрим две любые сложные кучи на Прологе в рисунке 6.1.

```
p2(X,Y):-pointsto(loc2,X),pointsto(loc3,Y).
p1(X,Y):-pointsto(loc1,val1),p2(X,Y).
```

Рисунок 6.1: Пример сложных куч

Здесь **p2** означает некоторый предикат с двумя символами **X** и **Y**, которые представляют собой некоторые значения, которые указываются локализаторами **loc2** и **loc3**. В отличие от этого, **p1** определяется через предикат **p2**. Как только мы вызываем **p2** с двумя синтаксически корректными аргументами, так мы имеем одну форму  $a(\vec{\alpha})$ . Вспомним, Пролог не может найти решения для синтаксически корректных, но семантически некорректных термов потому, что «*семантически некорректно*» означает, нахождение не выводимых термов для данных прологовских правил.

Интерпретация формулы  $H$  для данной кучи означает отображение от двух куч, т.е. данной и сравниваемой кучи, в булеву ко-область. Это означает, что если две данные кучи совпадают, то интерпретация успешна, в противном случае, наоборот. Для данных интерпретаций рассматривается только дедуктивный вывод (см. разделы 1.1.1 и 1.1.2). Поиск вывода завершается успехом тогда, когда запрос успешен, во всех остальных случаях завершается провалом. Несомненно, это точно то, что ожидается получить от предлагаемого поведения (см. тез.4.5).

Ради простоты согласуем, что формулы куч должны быть нормализованы в форму

$$a_0 \star a_1 \star \dots \star a_n \equiv \prod_{\forall j}^n a_j, n \geq 0$$

Далее,  $\wedge$  и  $\vee$ -связанные графы куч задаются в Прологе в виде запросов формой

$s_j, s_{j+1}, \dots, s_{j+k}$ . Альтернативно можно дизъюнкцию прологовских целей разбить далее на неко-



торые альтернативные правила, головы чьи различаются с помощью оператора «;». Отрицание утверждения рассматривается как отрицание предиката. В общем, отрицание последовательности не означает отрицание предиката потому, что последовательность для всех может быть просто не определена, кроме некоторого домена, это надо учесть. Двойное отрицание в общем случае не действительно при вызовах подцелей потому, что предикат может быть не тотален. Экзистенциальные переменные могут быть введены в любом месте правила Пролога, однако ожидается, что все введённые переменные когда-то присваиваются и используются.

Константные функции, как например, **true** и **false**, являются «синтаксическим сахаром», т.к. они могут быть заменены на любые другие кучи, которые квалифицируются в качестве вставных куч. Использование константных функций упрощает в спецификациях все возможные кучи. **true** может быть сопоставлена булевой истине, **false** наоборот противоречием.

**Заключение 6.2** (Корректность кучи). *Любая синтаксически корректная формула кучи описывает соответствующий граф кучи. Более того, любой граф кучи может быть представлен соответствующей формулой кучи. В общности обе стороны действительны ради исключения бесконечных куч.*

**Определение 6.3** (Неформальный граф кучи). *Граф кучи является связанным графом, который направлен и располагается в динамической части памяти. Динамическая часть выделяется при создании процесса операционной системой. Вершины графа указываются хотя бы одной локальной переменной, либо доступны локатором. Каждая вершина графа связана с адресом в динамической части операционной памяти. Ширина вершины может меняться с каждым указателем и зависит, прежде всего, от типа переменной. Когда одна вершина ссылается на другую, то обе вершины соседствуют в соответствующем графе. Если вершина имеет два указателя, то один становится, безусловно, псевдонимом другого.*

Опр.6.3 является уточнением впервые введённого графа кучи из набл.3.5.

## 6.3 Предикаты как логические правила

Абстрактные предикаты позволяют абстрагировать от обыкновенных куч к более сложным кучам, но более интуитивным человеку, используя выражения и формулы. Например, [193] вводит абстрактные предикаты, которые аннотируют данную входную программу и переводятся вместе с программными операторами до уровня ассемблера. Представленный подход автоматизации является попыткой преодолеть разрыв между спецификацией и логическим выводом. Пролог используется в этой главе как язык программирования, в котором утверждения и абстрактные предикаты о кучах специфицируются. Однако, между программой и языком утверждений существует семантический разрыв: одновременно имеются два параллельных формализма и реализации. Они приводят к всё

более различающимся нотациям и представлениям. Естественно, язык программирования может и должен различаться, когда речь идёт о возможном императивном языке программирования. В согласии с вычислением Хора, логические формулы описываются *логически*. Увы, иногда это нарушается (см. главу 1), а также имеются ограничения вычислимости (см. главу 4). Так, например, переменные (объекты) используются как локальные переменные вместо термов, вследствие чего, имеется целый ряд ограничений. Таким образом, язык спецификации, точнее ее частицы, «*деградируются*» в последовательность команд и больше не имеется ничего общего с изначальным замыслом вычисления Хора. Частицы не имели бы ничего общего с «*декларативно-логической парадигмой*» (см. набл.4.7 и набл.4.9): необходимо описать состояние вычисления, используя символы и предикаты. Символ и его диапазон годности всё-таки отличается от локальных переменных. Если символы «*вдруг переписываются*», то вряд ли это можно считать символом. По определению символам не приписываются новые значения заново, а переменным приписываются. Разница может казаться не слишком большой, но для определения и описания это имеет очень серьезные последствия. Вычисления часто (но не всегда) описываются символами, без переменных. При описании центральной концепцией являются термы, предикаты и рекурсии, а не цикл или условный переход, это нужно учесть. В обеих моделях можно установить минимальный набор Тьюринг-вычисляемых программ. Для решения верификации необходимо сравнивать состояния вычислений для того, чтобы определить, вычисление было совершенно правильно или нет. Проблема сравнения не исключает возможность и необходимость вычислять арифметические или алгебраические выражения, но при этом, подход и замысел вычисления Хора декларативен.

Подход представленный в [27] вводит символы в кучах, но с ограничениями. Например, отсутствует возможность описывать целые кучи, как например  $X \star Y$ . В отличие от этого, мы допускаем символы без ограничений полностью как они допускаются в Прологе (см. главу 4). В различных вариантах мы вынуждены будем выбирать только между теми правилом, которое имеет более длинное совпадающее предусловие – как это было реализовано, например, в [27]. Мы принципиально следуем поиску Пролога, что в обобщённом случае может быть слишком много, но «*метод ветвей и границ*» [248],[46] позволяет нам произвольно сужать поисковое пространство. Таким образом, мы не ограничиваем себя в некоторой методологии, либо *эвристике*, либо *тактике*. Любая методология может меняться частично, либо полностью — мы в состоянии всё это учитывать.

Пролог используется для того, чтобы определить, какие существуют подлежащие в кучах и какие связи между ними, для этого мы используем предикаты. Поэтому, можно считать, например [266] более близким подходом, чем функциональный или императивный. Когда речь идёт о проверке состояния куч, в общем легче описать, опираясь на факты и правила, чем на последовательность инструкций. С помощью абстрактных предикатов описываются кучи. За компактность, представление фактов и правил, ответственный — разработчик программы. Следующие формализмы помогут преобразовать абстрактные предикаты в формальную грамматику для дальнейшего представления.

$$\begin{aligned}
\langle predicate \rangle &::= \langle head \rangle [ \text{' :- ' } \langle body \rangle ] \text{' . ' } \\
\langle head \rangle &::= \langle atom \rangle [ \text{' ( ' } \langle arguments \rangle \text{' ) ' } ] \\
\langle body \rangle &::= \langle sub\_goal \rangle \{ \text{' , ' } \langle sub\_goal \rangle \}^* \\
\langle sub\_goal \rangle &::= \text{' ! ' } \mid \text{' fail ' } \mid \langle functor\_term \rangle \mid \langle term \rangle \langle rel \rangle \langle term \rangle \\
\langle rel \rangle &::= \text{' = ' } \mid \text{' \= ' } \mid \text{' < ' } \mid \text{' < = ' } \mid \text{' > ' } \mid \text{' > = ' } \\
\langle functor\_term \rangle &::= \langle atom \rangle \text{' ( ' } [ \langle arguments \rangle ] \text{' ) ' } \\
\langle arguments \rangle &::= \langle term \rangle \{ \text{' , ' } \langle term \rangle \}^* \\
\langle term \rangle &::= \langle atom \rangle \mid \langle var \rangle \mid \langle list \rangle \mid \langle number \rangle \mid \langle functor\_term \rangle \\
\langle list \rangle &::= \text{' [ ' } [ \langle term \rangle \text{' | ' } ] \langle arguments \rangle \text{' ] ' }
\end{aligned}$$

Рисунок 6.2: Расширенная форма РФБН прологовских правил

Все перечисленные преимущества характерны для Пролога и используются далее в предложенном методе.

**Определение 6.4** (Предикатное правило). *Предикатное правило определяется как  $\forall n. a : -q_{k \times n} \Leftrightarrow a : -q_{k,0}, q_{k,1}, \dots, q_{k,n}$  для  $k \in \mathbb{N}_0$ .*

Неслучайно опр.6.4 близко к опр.4.2. Первое определение используется для описания и вызова предикатов куч. Уговаривается по умолчанию, что  $a$  действительно всегда тогда, когда все подцели  $q_{k,j}$  в  $a$  в силе для  $0 \leq j \leq n$ . Синтаксис предикатного правила определяется расширенной формой Бэккуса-Наура как показано на рисунке 6.2.  $\langle number \rangle$  определяет любое прологовское число, а  $\langle atom \rangle \text{' ( ' } \langle arguments \rangle \text{' ) ' }$  определяет некоторый функтор с простым именем  $\langle atom \rangle$  и любым количеством аргументов.  $\langle var \rangle$  означает некоторый переменный символ, который начинается с большой буквы, например  $X$ . Рисунок 6.3 в главах 3 и 4 представляет пример полезного предиката функционала `map` на Прологе.

Допустим, имеется некоторый предикат  $a$ , тогда подцели  $q_{k,j}$  вычисляются слева направо для  $j \geq 0$ . Символьная среда  $\sigma$  внутри тела предикатного определения обновляется после каждого вызова, каждого из подцелей тела согласно рисунку 4.9. Не присвоенные символы остаются, но могут присваиваться после подцели. Термы результаты ранних целей не нуждаются в обновлении, так как символам присвоено неопределённое символьное значение. Семантика вызова предиката определяется следующим образом:

$$C(a) \llbracket a(\vec{y}) : -q(\vec{x}_{k,n})_{k \times n} \rrbracket \sigma = D \llbracket q_{k,n} \rrbracket \sigma(\vec{x}_{k,n}) \circ \dots \circ D \llbracket q_{k,1} \rrbracket \sigma(\vec{x}_{k,1})$$

По определению вектор термов  $\vec{y}$  может содержать общие элементы с вектором  $\vec{x}_{k,n}$ ,  $\forall k, n$  и  $C \llbracket . \rrbracket$

имеет тип  $\text{atom} \rightarrow \text{predicate} \rightarrow \sigma \rightarrow \sigma$ ,  $D[\cdot]$  имеет тип  $\text{subgoal} \rightarrow \sigma \rightarrow \sigma$ , и  $\sigma$  имеет тип  $\text{term}^* \rightarrow \text{term}$ , где  $\star$  означает *звезда Клини* (см. [79]).

Подцель  $q_{k,j'}$  не обязательно должна определять связанный граф изначально. Но, если это так, то это признак тому, что подмножество кучи определено целиком, по крайней мере, единая по интуиции структура данных. При разработке ПО модульность и «разделение забот» можно всегда считать хорошим признаком. Таким образом, абстрактный предикат вынуждает к описанию целой структуры данных. Следовательно, можно вывести следующий лозунг: «*один абстрактный предикат должен корреспондировать с одной кучей*», где под подкучей подразумевается не пустой граф кучи, который содержит настоящее подмножество вершин и представляет собой кучу. Далее, добавляя все более  $\star$ -конъюнктов, соответствующий граф кучи растёт непрерывно. Набор  $\star$ -конъюнктов образует кучи, возможно связанные между собой, которые соответствуют абстрактным предикатам. Когда речь идёт о «*сжатии или раскрытии*» абстрактных предикатов (похоже на вызов метода), существуют параметры, точнее вершины графа куч, которые стоят на обеих сторонах: со стороны вызова и со стороны вызванного предиката. Также могут существовать вершины, которые видны только изнутри предиката, которые не могут быть использованы извне предиката (по крайней мере, явным образом).

Без ограничения общности мы согласуем, что доступ к объектным полям с помощью функтора «.» разрешается, например  $a.b$  (см. рисунок 6.2) или  $oa(\text{object5}, \text{fld123})$  [297]. Ради примера и модульности, мы согласуем далее, что объекты, а также объектные поля, передаются в качестве параметров предикатам. Отличие между объектами и простыми стековыми локальными параметрами отсутствует, подробное объяснение будет дано позже.

**Определение 6.5** (Набор предиката). *Набор предиката  $\Gamma_a \subseteq \Gamma$  для некоторого предиката именем  $a \in T$  и  $\forall i.j.q_{i,j} \in (T \cup NT)$ , где  $T$  терминалы, а  $NT$  нетерминалы, определен как:*

$$\begin{aligned} \Gamma_a ::= & \quad a : -q_{m \times n} \\ & \quad a : - \quad q_{0,0} \quad , \quad q_{0,1} \quad , \quad \dots \quad , \quad q_{0,m} \\ \Leftrightarrow & \quad \vdots \quad \quad \quad \vdots \quad \quad \quad \ddots \quad \quad \quad \vdots \\ & \quad a : - \quad q_{m,0} \quad , \quad q_{m,1} \quad , \quad \dots \quad , \quad q_{m,n} \end{aligned}$$

Если  $m = 0$ , тогда  $a$  является фактом. К  $a$  могут быть ещё приписаны термы (содержавшие символы, например, когда  $m = 0$ ,  $n > 0$ ). Если  $t \in T$ , то  $t$  имеет вид  $loc \mapsto val$ , иначе  $t \in NT$  означает предикат под именем  $t$ , который имеется в  $\Gamma$ .

По умолчанию согласуется, что для последовательности  $q_{k,0}, q_{k,1}, \dots, q_{k,m}$  из  $q_{m \times n}$  любая строка находится в нормализованном виде, так, что для  $s \leq m$  нетривиальных элементов первые  $s$  подцели располагаются, а остальные  $m - s$  подцели являются тавтологиями в качестве подцелей, чей домен полностью определён как истина ( $\top$ ). Далее, согласуется, что:

$$\exists k.a : -q_k \preceq a : -q_{k+1}$$

в силе, означая, что предикат, появляющийся ранее, в  $\Gamma_a$  имеет выше приоритет, чем предикат, который определён позже.

**Заключение 6.6** (Предикатная среда). *Для предикатной среды  $\Gamma$  данной прологовской программы,  $\Gamma = \bigcup_{t \in T} \Gamma_t$  в силе. Все предикаты  $\Gamma_t$ , которые зависят друг от друга, обязательно находятся в одном замкнутом разделе предиката  $\bar{\Gamma}$ .  $\bar{\Gamma}_t \subseteq \bar{\Gamma}$  соблюдается.*

*Доказательство.* Идея заключается в показании следующего: все зависимые  $\forall t. \Gamma_t$  находятся в одном разделе предиката, а все независимые разделы, естественно, не зависят от зависимых предикатных сред. Все предикатные среды независимо лежат в  $\bar{\Gamma}$ , предикаты зависимы или независимы. Предикаты  $\Gamma_a$  и  $\Gamma_b$  от не соседних разделов из  $\bar{\Gamma}$  никогда не могут зависеть друг от друга.  $\square$

**Замечание:** Очевидно, из-за проблемы приостановки, вызов предиката из раздела в общем не решим. Далее рассматривается выразимость предикатов.

**Замечание:** Разрешение конфликтов с именами в  $\Gamma$  может быть разрешено, если закодировать локацию предиката в имя, как например класс, для которого предикат предусмотрен, тогда станет возможно различать предикаты. По определению предикаты с одинаковой локацией являются частью предикатного раздела, а следовательно не конфликтуют.

**Лемма 6.7** (Полнота предикатов). *Абстрактные предикаты покрывают все предикаты первого порядка для описания куч.*

*Доказательство.* В [144] можно ознакомиться с полнотой и выразимостью предикатов первого порядка на языке Пролога.  $\square$

**Лемма 6.8** (Предикаты высшего порядка). *Абстрактные предикаты могут выразить предикаты второго и высшего порядка.*

*Доказательство.* Пока мы только ограничивались вопросами выразимости в Прологе предикатов первого порядка. Далее, мы рассмотрим предикаты в Прологе реализующие высшие порядки выразимости. Высший порядок отличается от первого порядка тем, что далее предикат абстрагируется, т.е. предикат используется в логических выражениях как переменная, которая зависит от параметров. В Прологе, это происходит при вызовах, с помощью встроенного предиката `call`, который принимает список входных и выходных термов, как например `pred1(X) :- call(pred2,X)`.

Допустим, некоторый предикат  $P$  для списка входных термов  $[X|Xs]$ , список выходных термов  $[Y|Ys]$  и список термов, которые одновременно входные и выходные, пуст. Тогда определяется предикат `map` как указано на рисунке 6.3, который применяет предикат к каждому входному элементу последовательно слева направо. Предикаты высшего порядка могут оказаться полезными, особенно для модулей и структурированных данных. Например, для классных экземпляров объектов и потока выполнения меняется в *паттернов поведения*, как например «наблюдатель» [136],[146]. О паттернах в области верификации, в частности в связи с кучами [146],[145],[211], очень важно обсуждать,

```

map([], P, []).
map([X|Xs], P, [Y|Ys]) :-
    Goal =.. [P,X,Y],
    call(Goal), map(Xs, P, Ys).

```

Рисунок 6.3: Функционал map/3

т.к. паттерны неформальным образом могут сближать интуитивное понятие вместе с дедуктивным выводом со спецификацией.

Тип предиката `map/3` является  $list_a \rightarrow (list_a \rightarrow list_b) \rightarrow list_b$ , т.е. вторым входным типом назначается семейство предикатов, которое на входе ожидает список одного базисного типа, а на выходе базисный тип  $b$ . Не уточняется,  $a = b$  или  $a \neq b$ . Таким образом, рекурсия может быть сопоставлена с помощью предикатов третьего и высшего порядка, например, с помощью *левого свёртывания* (*foldl*), которое принимает некоторый предикат  $\oplus$  к данному списку входных термов к уже имеющемуся результату вычисления. Начиная данным нейтральным элементом:

$$\text{foldl}(\oplus :: a \rightarrow b \rightarrow a, \varepsilon :: a, X :: list_b) :: a$$

Правое свёртывание работает аналогично, начиная с правой стороны и продолжая вычисления справа-налево. `foldl` определяет *начальную алгебру* с начальным значением  $\varepsilon$  и множество носителя  $X$ , а также операцию  $\oplus$ , которая определена одинаковым типом, как и  $\varepsilon$  и применяется поэлементно для каждого элемента из  $X$ .  $\oplus$  вычисляет результат того же типа как и  $\varepsilon$ . Допустим,  $a$  равно  $b$  и оба переменные целые числа, а также  $X = [1, 2, 3]$  имеют вид «список целых чисел». Предположим, начальный счётчик  $\varepsilon$  равен 7, тогда `foldl` вычислит  $((\varepsilon + 1) + 2) + 3$ , что дает 13, а это явно целое число. Как мы увидим позже, предикаты высшего порядка не столь полезны для верификации куч, как для выражения индивидуальных ограничений и преобразований списков.  $\square$

Ради полноты синтаксического определения из рисунка 6.2 и перевода представленного в следующем разделе, необходимо задуматься о том, как правильно перевести прологовские операторы последовательности « $;$ » и отсечения « $!$ ». Если тело предиката содержит « $;$ », тогда всю последовательность после « $;$ » необходимо перенести в новый предикат с той же левой стороной. Так, например:

$$b : -a_0, a_1, \dots, a_m; a_{m+1}, \dots, a_n$$

для  $\exists m. 0 \leq m \leq n$  разбивается на:

$$b : -a_0, a_1, \dots, a_m. \quad b : -a_{m+1}, \dots, a_n.$$

Если аналогично встречается оператор отсечения « $!$ » в:

$$b : -a_0, a_1, \dots, a_m, !, a_{m+1}, \dots, a_n$$

, то  $a_0, a_1, \dots, a_m$  может содержать альтернативы, которые будут рассматриваться в случае провала. «!» утверждает, что если только одна подцель от  $a_{m+1}$  до  $a_n$  проваливается, то  $b$  полностью проваливается без дальнейшего поиска альтернатив. Все альтернативы могут быть факторизованы слева от «!» так, чтобы иные альтернативы исключались. В кратце, это является причиной, почему «;» и «!» могут быть исключены из Пролога в общности без потери выразимости (см. набл.4.13). Вопрос обобщения предикатов, безусловно, интересен, но в целях задач поставленных в этой работе далее не рассматривается. Подход Поулсона [198] задаётся вопросом достижения обобщения с помощью введения функционалов на уровне абстракции и логических правил [206],[170],[106],[80] для метода резолюции [286],[288],[103] — которые здесь далее не рассматриваются. Модули тактик в системе «*Coq*» [32] основаны на принципе, который также описывается Поулсоном.

Значение лем.6.7 и лем.6.8 таково, что можно выразить любые предикаты эквивалентности в Прологе беспрепятственно и без явной рекурсии. Мы не ограничиваем себя и допускаем  $\mu$ -рекурсивные предикаты ценой частичной корректности в связи с не определением приостановки интерпретации предикатов ради беспрепятственной выразимости.

**Определение 6.9** (Свёртывание предиката). *Развёртывание/Свёртывание предиката  $a(\vec{\alpha})$  из/в некоторый предикат  $a$  для данных предикатов  $\Gamma_a$  с настоящими значениями термов  $\vec{\alpha}$ /подцелей  $q_k$  определяется как: из-за лем.6.8 допустим,  $\Gamma_a$  равно без ограничения общности  $a(\vec{y}) : -q_k$  с  $q_k = q_{k,0}(\vec{x}_{k,0}), q_{k,1}(\vec{x}_{k,1}), \dots, q_{k,m}(\vec{x}_{k,m})$ . Если  $\vec{\alpha} = (\alpha_0, \alpha_1, \dots, \alpha_A)$  и  $\vec{y} = (y_0, y_1, \dots, y_A)$ , то*

$$a(\vec{\alpha}) \Leftrightarrow q_{k,0}(\vec{x}_{k,0}), q_{k,1}(\vec{x}_{k,1}), \dots, q_{k,m}(\vec{x}_{k,m}) \text{ с } \alpha_0 \approx y_0, \alpha_1 \approx y_1, \dots, \alpha_A \approx y_A.$$

В случае « $\Rightarrow$ » верхнего равенства  $a(\vec{\alpha})$  предикат развёрнут. В случае « $\Leftarrow$ » правая сторона определения предиката свёртывается в вызов предиката. « $\approx$ » означает *унификацию термов*.

## 6.4 Интерпретация предикатов над кучами

Предложенный в этом разделе универсальный, но нестандартный, подход зависит от следующих этапов:

1. Преобразование входной программы и аннотированные утверждения в прологовские термы, которые затем вписываются в логическую систему на основе Пролога (см. рисунок 4.15, [297]).
2. Определение абстрактных предикатов в предусмотренной части прологовской программы вместе с представлением входной императивной программы. Правила принадлежат интерпретации и следовательно нуждаются в синтаксической корректности. Также как и пункт 1, этот пункт подробнее рассматривается в разделе 4.5.
3. Определение формальной грамматики для данных абстрактных предикатов. Обработка грамматики языковым процессором, которая при успехе генерирует конкретный синтаксический анализатор.

4. Во время доказательства использование и подключение ранее преобразованных в синтаксический анализатор абстрактных предикатов при вычислении подделей.

**Наблюдение 6.10** (Сходство с формальными языками). *Глядя на структуру кучи, они напоминают об определениях формальных языков.*

Простое утверждение « $\mapsto$ » становится терминалом (см. опр.6.5). Абстрактный предикат становится нетерминалом, точнее его вызовом. Терминалы могут связываться последовательно с помощью бинарного оператора  $\star$ , который коммутативен (см. трм.3.6). Терминал получает значение единицы некоторой подграмматики — это эквивалентно грани некоторой данной куче. Утверждения « $\mapsto$ » могут эффективно связываться, когда левые локации сортируются по лексикографическому порядку. Когда происходит конфликт с одинаковыми именами, то  $\alpha$ -преобразование, учитывая местонахождения в данном модуле, может разрешиться, например, вводя префикс.

**Тезис 6.11** (Распознавание как доказательство). *Распознавание абстрактных предикатов осуществляется как доказательство.*

*Доказательство.* (см. далее в этой главе). В частности опр.6.20 и опр.6.21 разрешают определить соответствующий синтаксический анализатор, который тотален и приостанавливается согласно закл.6.17. Выводимая строка, которая содержит терминалы кучи и нетерминалы определены в опр.6.19. □

**Заключение 6.12** (Контекст-свободность выражений куч). *Раздел абстрактного предиката описывает систему правил, которая является контекст-свободной формальной грамматикой.*

*Доказательство.* Левая сторона предиката не может по определению содержать более одного нетерминала, терминалы очевидно также не допускаются. Следовательно, только один нетерминал разрешается на левой стороне. Отсутствие требования, где правая сторона должна быть строго право-рекурсивной, т.е. отсутствие требования регулярной грамматики с правилами формой « $S \rightarrow aA$ », приводит к контекст-свободности. Допускается распознавание грамматик, которые эквивалентны к «скобочным грамматикам» [110] (КС-грамматики, чьи правила могут иметь вид  $S \rightarrow (S)$ ), а именно, когда имеется  $n \in \mathbb{N}_0$  для некоторых терминалов  $a, b, x_0, x_1$  и  $x_2$ , которые могут быть « $\mapsto$ »-утверждением, так, чтобы  $x_0 a^n x_1 b^n x_2$  и  $x_0 \neq a$ ,  $x_0 \neq x_1$  и  $x_1 \neq b$ ,  $b \neq x_2$ . Если голова предиката содержит аргументы, то это всё равно не меняет статическую зависимость между определениями предикатов. Каждому разделу предиката можно приписывать один начальный нетерминал. □

**Наблюдение 6.13** (Редукция куч). *Генерация кучи абстрактными предикатами порождает вопрос об уровне абстракции и проверки, т.к. куча является генерированным элементом, который принадлежит проверке (см. [298]).*



Из этого следует: выведенная и ожидаемая куча, обе они могут содержать свёрнутые предикатные определения, которые должны быть развёрнуты для окончательного определения равенства. Не трудно увидеть, что этот процесс двунаправленный, так как свёрнутые подкучи могут содержаться в обеих кучах. Важно заметить, что этого рода проблема редуцируется к «*проблеме корреспонденции Поста*», которая в общем случае теоретически не является решимой, но только в частных случаях. Общая проблема для куч сформулированная Постом не рассматривается.

Набл.6.10 вместе с набл.6.13 может быть рассмотрено как предпосылка на формулировку: *дана  $\star$ -связанная куча. Вопрос: совпадает ли она или нет с данной спецификацией кучи?* А также можно задать вопрос — *какая куча самая близкая к корректной куче, так, чтобы спецификация соблюдалась?* Решение вопроса способствует к решению проблемы *контр-примера*.

**Лемма 6.14** (Куча как слово). *Проблему слова для раздела абстрактных предикатов  $P$  можно задать как: если даны  $\alpha_1, \alpha_2 \in L(G(P))$ , то следует ли из этого, что  $\alpha_1 \equiv \alpha_2$ ?  $G(P)$  означает формальную контекст-свободную грамматику, полученную из раздела предиката  $P$ .*

*Доказательство.* Здесь  $\alpha = (a + A)^*$ ,  $a \in T$ ,  $A \in NT$ ,  $\alpha$  является предложением.  $T$  означает множество терминалов, которые параметризованы и записываются как единое, которое содержит начало и конец « $\mapsto$ »-утверждения.  $NT$  означает нетерминалы, которые содержат все предикаты, а также все синтаксически корректные входные термы, которые могут быть параметризованы. Раздел предиката  $P$  определяет формальную грамматику  $G(P)$ . Из трм.6.11 следует контекст-свободность грамматики, а далее от приложения к теореме контекст-свободность генерируемого языка  $L(G(P))$ . Начальный нетерминал является вызовом предиката из  $\alpha_1$  или  $\alpha_2$ . Необходимо, изначально вычислить и определить при переборе множества последующих терминалов  $\sigma(\alpha)$ , а также множества началов нетерминалов  $\pi(\alpha)$  должны быть вычислены (см. позже).  $\sigma(\alpha)$  и  $\pi(\alpha)$  вычисляются один раз, если  $P$  не меняется. При этом необходимо заметить: можно задать для данного  $G(P)$  более одного начального нетерминала, в зависимости от подцелей в  $\alpha_1$  и  $\alpha_2$ . Более того, обыскивается не только один путь, начиная с  $\alpha_1 \vdash^* \alpha_2$ , но также начиная с  $\alpha_2 \vdash^* \alpha_1$ , где  $\vdash$  обозначает разовое применение правил. Только когда, не обнаруживается путь с обеих сторон, только, тогда можно утверждать, что  $\alpha_1$  не совпадает с  $\alpha_2$ , обратное не действительно. Чтобы проверить, совпадают ли два предложения куч, необходимо построить не только пути между предикатами, а также принимать начальные и промежуточные терминалы формой « $\mapsto$ ». Параметры в терминалах и нетерминалах в случае семантически корректных утверждений связаны, т.е. не связанных (не)терминалов нет, поэтому гарантированно, что существует параметр, который присваивает значение в обоих  $\alpha_1$  и  $\alpha_2$ . Предложенная платформа (см. главу 4) способствует тому, что при необходимости дополнительные проверки к предикатам могут подключаться произвольно, например, проверку на присутствие специфических терминалов внутри предикатного тела.  $\square$

## 6.5 Перевод правил Хорна

В этом разделе рассматривается перевод абстрактных предикатов, которые даны в качестве прологовских правил, в обобщённые правила контекст-свободной грамматики с атрибутами. Перед этим необходимо преобразовать обыкновенные утверждения формой  $loc \mapsto val$  в токены, как принудительный лексико-аналитический шаг интерпретации Пролога. Применение *мульти-парадигмального программирования* [81] разрешает интерпретировать прологовские правила во время запуска различными языковыми процессорами. Это позволяет достичь максимальную расширяемость, например, подключение написанных процедур на различных языках программирования и запуск в третьей загрузочной системе. Таким образом, процесс верификации может быть инициирован, контролирован и приостановлен входной программой. Процесс трансляции из правил Пролога в формальную грамматику удивительно прост, ради интерпретации правил Пролога. Однако, правила Пролога могут иметь аргументы с обеих сторон от знака определения правила «:-». Параметры и аргументы генерируемой грамматики можно моделировать как атрибуты формальной грамматики. Следовательно, процесс трансляции  $C[\llbracket \cdot \rrbracket]$  можно характеризовать как:

**Определение 6.15** (Преобразование в атрибутируемую грамматику).  $C[\llbracket \cdot \rrbracket]$  является семантической функцией преобразователя (с англ. «transducer») входных абстрактных предикатов в атрибутируемую грамматику и определяется следующим образом:

$$\begin{aligned} C[\llbracket \cdot \rrbracket] &= \emptyset \\ C[\llbracket C_1.C_2 \rrbracket] &= C[\llbracket C_1 \rrbracket] \dot{\cup} C[\llbracket C_2 \rrbracket] \\ C[\llbracket a(\vec{x}) : -q^0(\vec{x}), \dots, q^n(\vec{x}) \rrbracket] &= \{a_{\vec{x}} \rightarrow q_{\vec{x}}^0 \dots q_{\vec{x}}^n\} \end{aligned}$$

В отличие от ранее введённых нотаций далее вводятся подцели и теперь входной вектор  $\vec{x}$  содержит все переменные символы, ради удобной записи внутри каждого предиката. Если некоторая подцель  $q_j$  для  $j \geq 0$  не нуждается во всех компонентах  $\vec{x}$ , то подцель не нуждается в них.  $\dot{\cup}$  является множественным объединением, с учётом последовательности и сохранения дубликатов. Нетрудно заметить, что обе записи сопоставимы и сильно похожи друг на друга. Абстрактный предикат описывает кучу. Таким образом,  $C[\llbracket \cdot \rrbracket]$  преобразует кучу, т.е. является интерпретацией ассоциативной кучи, см. лем.6.14.  $C^{-1}[\llbracket \cdot \rrbracket]$  преобразует атрибутируемую грамматику обратно в Пролог:

**Определение 6.16** (Обратимость преобразования).  $C^{-1}[\llbracket \cdot \rrbracket]$  преобразователь входной атрибутируемой грамматики в набор абстрактных предикатов в качестве результата:

$$\begin{aligned} C^{-1}[\llbracket \cdot \rrbracket] &= \emptyset \\ C^{-1}[\llbracket C_1.C_2 \rrbracket] &= C^{-1}[\llbracket C_1 \rrbracket] . C^{-1}[\llbracket C_2 \rrbracket] \\ C^{-1}[\llbracket a_{\vec{x}} \rightarrow q_{\vec{x}}^0 \dots q_{\vec{x}}^n \rrbracket] &= \{a(\vec{x}) : -q_0(\vec{x}), \dots, q_n(\vec{x})\} \end{aligned}$$

**Заключение 6.17** (Приостановка преобразований).  $C[\llbracket \cdot \rrbracket]$  and  $C^{-1}[\llbracket \cdot \rrbracket]$  всегда приостанавливает работу для любого входного определённого вектора.

*Доказательство.* Доказательство простое, так как бесконечные циклы исключены. Преобразователи  $C$  и  $C^{-1}$  согласно опр.6.15 и опр.6.16 сканируют входные правила пошагово слева направо. Допустим, существует не завершающий цикл в данных правилах, то всё равно преобразователи завершают свою работу. Это потому, что цикл касается только разбора. Начало раздела предиката корреспондирует один к одному с начальным терминалом соответствующей *подграмматики*. Абстрактные предикаты могут иметь несколько стартовых точек, а следовательно, в соответствующей грамматике начальные нетерминалы могут меняться, но правила не меняются. Распознаватель может быть гибко использован, если все нетерминалы доступны снаружи. Это наблюдается в некоторых анализаторах, например «*ANTLR*». Распознавание в различных нетерминалах означает распознавание подвыражений.  $\square$

Ещё осталось расследовать  $C$  и  $C^{-1}$  касательно корректности и полноты.

**Заключение 6.18** (Корректность и полнота преобразований).  $C$  и  $C^{-1}$  полны и корректны.

*Доказательство.* Не трудно убедиться в том, что  $C \circ C^{-1} \circ C \equiv C$  и  $C^{-1} \circ C \circ C^{-1} \equiv C^{-1}$ , сопоставляя верные определения. Обсуждения из раздела 6.2, ни «!» ни «;» не влияют на выразимость. Если для любого элемента из  $C$  преобразование не приостанавливается, то кообласть также не полностью определена. Тоже самое распространяется на  $C^{-1}$ .  $\square$

Нетрудно заметить, что прологовские правила не единственная форма, но очень близка к формальной грамматике. Более обобщённо можно включить императивные языки программирования с процедурами на основе автоматического запуска со стеком.

## 6.6 Синтаксический перебор как верификация куч

В целях простого и интуитивно понятного алгоритма, константы из опр.6.1 далее пока рассматриваются. Касательно классовых объектов, предикат **true** может означать, например, принимать все « $\mapsto$ »-утверждения до отметки, которую необходимо согласовать, в зависимости от данного правила. В данной отметке, которая представляет собой *безопасный пункт синхронизации* в качестве «*правил генерации ошибок*» [110], может продолжаться синтаксический перебор, если перебор застрянет из-за ошибочной последовательности в потоке токенов, либо из-за не ожидаемого состояния на стеке при переборе. Предполагается, что входное слово, представляющее кучу, всегда конечное. Предположение основывается по той причине, что память — это линейно адресуемое конечное пространство. Чисто гипотетично, число совершённых развёртываний и свёртываний стремится к бесконечности. Позже мы покажем, что для  $\exists j$  функции  $\pi_j$  и  $\sigma_j$  ограничиваются полиномиальной сложностью.

Этот раздел предлагает и обсуждает основные конвенции, необходимые для реализации общего подхода синтаксического перебора, в целях верификации куч. Это рассматривается на примере

LL(k)-перебора. LL(k)-анализатор, является синтаксическим анализатором, который может смотреть вперёд любое количество токенов для разрешения многозначности одинаковых правил. LL(k)-анализатор выбирается образцово. Кроме LL(k)-анализатора, могут быть использованы другие анализаторы синтаксиса, как например, LALR, SLR, Эрли и другие. Журдан [129] с помощью «Coq» доказывает корректность LR(1)-анализатора — вопрос корректности анализатора, безусловно, важен, но в рамках этой работы отпадает, т.к. анализатор конструируется автоматически из данной формальной грамматики. Далее, первым определяется формальное предложение, как композиция отдельных « $\mapsto$ »-утверждений и подцелей как нетерминалы. Вторым и третьим, в аналогии к LL(k)-анализатору, где отдельные терминалы представляются как обыкновенные утверждения операции, вводятся «*first*» и «*follow*». Четвёртым, обе операции *сдвиг* (SHIFT) и *свёртка* (REDUCE) представляются, чтобы иметь представление для более обобщённых анализаторов.

**Определение 6.19** (Абстрактное предложение). *Абстрактное предложение  $\alpha$  является  $\star$ -конъюнкцией куч, которая записывается как  $a \mapsto b$ , где  $a$  локация,  $b$  объект, содержащий некоторое значение, либо значение отсутствует: *nil*.*

Например,  $\alpha ::= [\text{pointsto}(x, \text{nil}), \text{pointsto}(y, 1), \text{member}(x, [y])]$  описывает актуальное состояние кучи при верификации данной императивной программы.  $\star$  заменяется в предыдущем списке запятой. Спецификация правил может зависеть от  $[\text{pointsto}(Y, 1), \text{member}(X, [Y|_ ]), \text{pointsto}(X, _)]$ .

Поэтому, для проверки абстрактного предложения (спецификация) для данной программы (генерируемая куча), необходимо сравнить, выводима ли одна из сторон из другой или нет.

Абстрактное предложение может также содержать унификацию термов, как например,  $\text{pointsto}(X, 5), X=Y$ . Унификацию термов необходимо тщательно проверять и отделять от: « $\mapsto$ »-утверждений и от вызовов предикатов, т.е. нетерминалов. Надо учесть, что неограниченная рекурсия термов должна ограничиваться так, чтобы терм сам себя не содержал по определению. Поэтому, *самосодержащие термы* необходимо ограничить, так как они часто имеются по определению в Прологе (см. раздел 4.1), где проверка самосодержимости по умолчанию отсутствует. Анализаторы, которые вынуждены анализировать неограниченные термы, могут попадать в тупиковую ситуацию, если сам терм циклически определяется самим собой. Поэтому, во избежание проблемы принудительной работы, уговаривается, что проверка на присутствие циклов проводится отдельно от верификации, либо по умолчанию исключается.

Рассмотрим теперь «*наивный*» подход для сравнения равенства двух абстрактных предложений. Рассмотрим алгоритм № 2.  $\pi$  означает функцию начальных терминалов для данного правила и данного положения, как было определено ранее. Проблема в этом подходе (развёртывая фактически принудительно предикаты всё далее и далее, возможно бесконечно) заключается в неопределённости, когда и сколько раз применить точно развёртку и свёртку, тем более не известно в случае

нахождения одного решения, не является ли решение оптимальным, что, безусловно, зависит от данных правил. Предположим, имеется:

$$\alpha_1 = [\underbrace{a \mapsto b}_{i_0}, i_1, \dots, i_{m_1}, \underbrace{q_1(x)}_{p_0}], \quad \alpha_2 = [\dots, \underbrace{a \mapsto b}_{j_3}, \dots, \underbrace{q_1(x)}_{q_7}]$$

необходимо сравнить сходимость термов обоих выражений. Сдвиг термов (SHIFT-TERMs) приводит к унификации  $i_0$  и  $j_3$  и продолжению сравнения остальных термов. Сначала свёртка (REDUCE-PREDS) проверит, подходящие и применяемые ли предикаты для расширения. Поэтому, первый терминал предиката может быть запрошен первым. Развёртывание  $expand(p_k, \alpha_1)$  сопоставит подцель телом определения предиката  $p_k$  (см. опр.6.9 и рисунок 6.2) и преобразует новое абстрактное предложение  $\alpha'$ , которое можно описать в Прологе как  $concat(\alpha, [i_7, i_8, i_9], \alpha')$ , если  $q_1(x)$  развёртывается в список  $[i_7, i_8, i_9]$ .

**Определение 6.20** (Множество началов нетерминалов). *Множество началов нетерминалов (first set) определяется как кообласть полного отображения  $\pi$  со следующим типом  $(T \cup NT) \rightarrow 2^T$  для  $m \in \mathbb{N}$ , так, чтобы соблюдалось:*

$$\pi(a) ::= \begin{cases} a & \text{если } a \text{ является } X \mapsto Y \text{ или } \Gamma_a ::= a. \\ \bigcup_{0 \leq j \leq n} \pi(q_{j,0}) & \text{если } \Gamma_a ::= a : -q_{m \times n}, n \in \mathbb{N}. \end{cases}$$

Независимо от конкретных аргументов,  $\pi$  определяет все те терминалы, которые являются « $\mapsto$ »-утверждениями, либо являются первым терминалом предикатных подцелей. Мы подразумеваем, что подцели унификации и вызовы встроенных (т.е. «встроенных») подцелей фильтруются и не рассматриваются при вычислении  $\pi$  и  $\sigma$ .

**Определение 6.21** (Множество последующих терминалов). *Множество последующих терминалов (follow-set)  $\sigma(t) \subseteq T$  для  $t \in (T \cup NT)$  определено как:*

$$\sigma(t) ::= \begin{cases} \bigcup_{i,j} \pi(q_{i,j+1}) & \text{если } t \text{ находится на месте } (i, j < n) \text{ в } q_{m \times n} \\ & \wedge 0 \leq i \leq m \\ & \wedge q_{i,j+1} \neq \top \\ & \wedge \exists a. \Gamma_a ::= a : -q_{m \times n} \\ \bigcup_a \sigma(a) & \text{если } t \text{ находится на месте } (i, n) \text{ в } q_{m \times n} \\ & \wedge \Gamma_a ::= a : -q_{m \times n} \\ & \wedge \exists b. \Gamma_b ::= b : -q_{m_b \times n_b} \\ & \wedge a \text{ находится на месте } (i_b, j_b) \text{ в } q_{m_b \times n_b} \\ \emptyset & \text{иначе} \end{cases}$$

Неформальное объяснение таково: множество последующих терминалов определяет все те терминалы, которые могут следовать данному актуальному « $\mapsto$ » утверждению или данной подцели в случае, когда терминал находится в конце правила. Согласно [110] мы теперь обладаем возможностью определить LL(k)-распознаватель с помощью определений  $\pi$  и  $\sigma$ .

**Пример 6.22** (Многозначность правил). *Даны следующие правила нетерминалов  $q_1 \rightarrow a$ ,  $q_2 \rightarrow aq_2 \mid q_3b$ ,  $q_3 \rightarrow \varepsilon \mid q_3a$ . Нетрудно заметить, что эти правила многозначны, например из-за  $\pi(q_2) = \{a\}$ ,  $\sigma(a) = \{\varepsilon\} \cup \pi(q_2) \cup \pi(q_3) \cup \sigma(q_3)$ .*

**Пример 6.23** (Корректность). *Дана следующая конечная спецификация:*

$$[(loc1, v1), p1(loc1, loc2), (loc2, v2)]$$

*и подходящая к нему условная цепочка  $[(loc1, v1), (loc2, v2)]$ . Цепочка является корректным словом данной спецификации, лишь в том случае, когда  $p1$  генерирует только пустую кучу.*

В случае, если данное слово не является просто последовательностью терминалов, но окажется «абстрагируемым» предложением, т.е. содержит произвольную последовательность терминалов и нетерминалов, то проблема сравнения может возникнуть в различных местах последовательности. Одной из этих проблем может оказаться вычисление повторяющихся куч и абстрактных частиц предикатов. Поэтому выгодно, когда вызовы абстрактных предикатов запоминаются в кэш (мемоизатор, например в [268]), а затем подцели могут быть сравнены гораздо быстрее, тем более, из-за декларативного характера предикатов, где символьные значения не присваиваются заново. Поэтому, предикаты могут быть сравнены чисто процедурно, без «опасных» побочных эффектов и изменения состояния вычисления несколько раз подряд. При ускорении кэширования необходимо запоминать наименование предиката и все термовые аргументы при вызове подцели.

Отрицания предикатов далее не рассматриваются из-за подробностей представленных в разделах 5.3, 5.5, 5.6, 4.1 и 4.2.

## 6.7 Свойства

На рисунке 6.4 показан пример конфигураций одной кучи, в которой любая вершина  $v_j$  при  $j \neq 2, j \neq 5$  с отходящими гранями для классного экземпляра, более одного указателя в качестве атрибута. Конфигурация состоит из 8 треугольников, каждый из которых ограничивается сплошной, пунктирной или извилистой линией. Линия, начиная с середины между  $v_0$  и  $v_3$  доходящая до  $M_1$ , выделяет треугольник  $\Delta(v_0, M_1, v_1)$ . Таким образом, рисунок 6.4 демонстрирует разновидность одной и той же структуры в динамической памяти, используя различные описания. Предполагается, что два и более выходящих указателей подразумевает соответствующее количество различных полей в объединённом экземпляре (см. главу 3 и рисунок 5.1). То есть, вершины могут охватывать те же самые графы различными абстрактными предикатами, например, треугольниками с пунктирными линиями или извилистыми линиями. Однако, при различных представлениях, необходимо учитывать все вершины — это является обязательным критерием, но не достаточным. Вопрос равенства двух различных куч, эквивалентен к вопросу *изоморфизма* двух данных направленных графов.

Когда анализируется входной поток токенов, последовательное состояние анализатора можно определить с помощью проверки нескольких токенов вперёд. В худшем случае, придётся проверить все входные токены, но решение будет принято. Это в случае, когда куча описывает объектный экземпляр и необходимо определить границы объекта. Если ограничиться тем, что один объект может быть описан только в одном предикате, либо поля указатели описываются зависимыми предикатами, то такая конвенция позволит избежать описанную проблему и провести генерическую канонизацию по предикатам. Синтаксический перебор, соблюдая данную конвенцию, имеет полиномиальную сложность [110]. Проводится синтаксический перебор без дополнительных условий. Если объектные границы не соблюдаются, то перебор может повлечь за собой пересечение куч, которые могут относиться к другому объекту. На практике это может означать, *частичный и параллельный перебор* различных куч одновременно, что желательно избежать по различным соображениям. Во-первых, проблема плохо делима из-за множества взаимосвязей, определений и из-за иерархии вызовов и фреймов. Во-вторых, надо задаться вопросом о преимуществе такого подхода, который кажется довольно сомнительным на данный момент потому, что не предвидится никакого ощутимого выигрыша, но дополнительные затраты на параллелизацию поглощают все возможные преимущества.

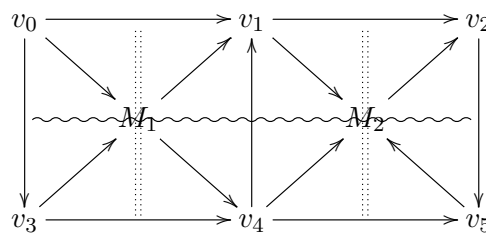


Рисунок 6.4: Пример конфигурации кучи

Поэтому, такой вопрос с теоретических и практических сторон пока не задаётся. С практической точки зрения необходимы параллельные кучи, которые наполнялись бы последовательно обработкой спецификации (см. *стек Трибера* [44]).

Если все разделы предикатов можно перебрать синтаксически, учитывая упомянутую конвенцию, то перебор « $\mapsto$ »-утверждений решим и завершается с ответом или с отказом. При отказе, синтаксическая ошибка содержит неверный токен и/или данный сегмент в нетерминале (т.е. состояние перебора). Таким образом, она показывает на состояние кучи, которая не совпадает с имеющейся кучей. Также необходимо заметить, что сходимость вычисления соблюдается при сборе LL-/LR-анализаторов потому, что состояние вычисления записывается как кортеж (состояние анализа, входная цепочка) и для каждого состояния переход детерминирован. В итоге, главная модель памяти не была кардинально изменена, а лишь расширена абстрактными предикатами.

## 6.8 Реализация

Система реализована на основе «*GNU Prolog*» [83] при поддержке «*ANTLR*» версии 4 [197]. Работа с абстрактными предикатами предусмотрена для работы [297], [302]. Изначально, обе работы, основанные на Прологе, были выбраны ради простоты пользования и в преподавательских целях. Для максимальной поддержки пакетов, написанные на Прологе, была использована дистрибуция «*GNU Prolog*», для максимального сходства с генеричной версией Пролога. Для расширения и возможности поддерживать различные библиотеки на различных языках, была использована *многоцелевая парадигма* [81], для динамического подключения загрузочного кода на разных языках программирования. Таким образом, программная библиотека Денти [81] разрешает подключать, например, Ява-библиотеку через соответствующую графическую оболочку. Процедуры на языке Ява или иных языках через соответствующий интерфейс при запуске прологовского запроса могут включить дальнейшие вызовы. Слой вызовов «*tuProlog*» выступает в качестве «*медиатора*» и «*адаптера*». При этом, вызов и результаты могут передаваться с обеих сторон с помощью «*прокси*», как будто процедура дальнего вызова библиотеки является локальным правилом Прологу. Таким образом, не только на языке Ява можно создать всё новые вспомогательные и встроенные предикаты, но также новосозданные предикаты могут снова вызывать предикаты. Так вызов при запуске Пролог программы согласно принципу ящика из рисунка 4.9, производится полностью динамически.

Реализация преобразует входную программу в IR, которое состоит из термов Пролога. Затем, утверждения копируются отдельно в теорию Пролога, а абстрактные предикаты преобразуются в грамматику «*ANTLR*», как это было изложено ранее. Затем, запрашивается перебор, синтаксический анализ с помощью того распознавателя подключается, который генерируется после определения «*ANTLR*» грамматики (часто на языке Ява). Затем, выход, контроль возвращается вызывающей стороне. При необходимости, абстрактные предикаты могут проверяться автоматически. В



случае синтаксической ошибки, выдаётся соответствующая ошибка.

«*ANTLR*» использует для разрешения синтаксической многозначности две технологии: «*синтаксические предикаты*» и «*семантические предикаты*» [197]. Термины очень похожи на проблемы данной работы, но их следует внимательно различать и не путать. Кроме этих двух технологий, всегда имеется возможность переписать правила, так, чтобы многозначность не возникала, например, с помощью *факторизации правил*. На практике «*ANTLR*», увы, не покрывает полный класс LL(k)-распознавателей. Предикаты ограничены и в случае конфликта часто нуждаются в *переписке (термовых) правил*. «*ANTLR*» часто не в состоянии различать самостоятельно, особенно регулярные, сложные выражения. Следовательно, приводит к полной генерации кода для необходимого распознавателя. Такая же проблема с ограниченностью LL(k) наблюдается во множестве других генераторов компиляции, как например, в «*yacc*» или «*bison*» [162]. С практической точки зрения это означает, что данная грамматика, хотя формально и корректна, не может быть перебрана из-за ограничений в реализациях распознавателей. К счастью, на практике переписка грамматики часто разрешает этот конфликт в практических целях. Также необходимо заметить, что более мощные распознаватели, по принципу, «*снизу-вверх*» имеют возможность избежать различные ограничения за счёт сложных реализаций и объёма, генерируемых распознавателем, например, «*bison*» работающий по принципу «*сдвиг-свёртка*». Обзор технологий синтаксического анализа можно найти в [291],[110].

В качестве примера, приведём необходимые трансформации на более подробном уровне для обработки лексемов и токенов. Сначала,  $bar \mapsto foo$  преобразуется в  $pt\_3bar\_3foo$ , где число «3» это длина наименований, либо сложное выражение целой локации, которая требуется для различения последовательных наименований. Если локация сложная, например  $b.f.g$ , то локация вместе с длиной определяют выражение утверждения в смежной записи (с англ. «*mangled*» [163]). Например,  $pointsto(X,2)$  преобразуется в прологовский атом  $p\_X\_2$ . Цель преобразований заключается в получении одного атомного символа, который представляет собой простое утверждение о куче, это терминал. При необходимости терминал без потерь может быть полностью восстановлен обратно в утверждение о куче, т.к. для этого все данные имеются и данные компоненты строго различаются в прологовском атоме. Процессы прямого и обратного преобразования могут быть реализованы в качестве встроенного предиката, например на языке Ява. Затем, используются в главных частях верификации, либо могут быть использованы другими встроенными предикатами на Яве, либо на другом подключенном языке к Прологу (см. предпосылки, [81]).

Далее, левая сторона (де-)канонизации (на Прологе)

« $p1(X, [X|Y]) :- \dots$  .» преобразуется в « $p1(X1,X2) :- X1=X, X2=[X|Y], \dots$  .».

Правило « $p(X,Y) :- \alpha$  .» из Пролога можно перевести в следующую «*ANTLR*»-грамматику:

$p[\text{String } X, \text{String } Y] : \alpha$ .

Таким образом, все *синтезируемые атрибуты* можно передавать сверху-вниз. *Порождаемые атрибуты* можно приписывать к соответствующему предикату *p*, добавляя в «*ANTLR*» ключевое слово **returns** вместе с наименованием атрибута перед двоеточием. Поэтому, необходимо определить, синтезируемы или порождаемы ли они и затем все атрибуты перевести и вписывать в соответствующее «*ANTLR*»-правило. Правила, конфликты и ограничения Пролога очень похожи на правила «*ANTLR*».

Когда абстрактные предикаты преобразуются в конкретную грамматику, например «*ANTLR*»-грамматику, то возникает проблема. Унифицированные термы, « $\mapsto$ »-утверждения (терминалы) и нетерминалы следует преобразовать согласно данному синтаксису в грамматику вместе с аннотациями. Также «*ANTLR*» разрешает *транслирующие правила*, которые определяют семантику самой программы и записываются в «*ANTLR*» в фигурные скобки. Отрицание предложений и их подпредложений можно сформулировать в «*ANTLR*» с помощью « $\sim$ » и скобок, которые означают, что данное регулярное выражение из « $\mapsto$ »-терминалов должно или не должно следовать. Далее, элементы перевода грамматик не обсуждаются, т.к. с помощью атрибутов и транслирующих правил, можно имитировать все остальные выражения и предложения, в том числе и несовпадение предикатов (см. [291], [290], [110], [197], [252], [179]).

## Графическая оболочка

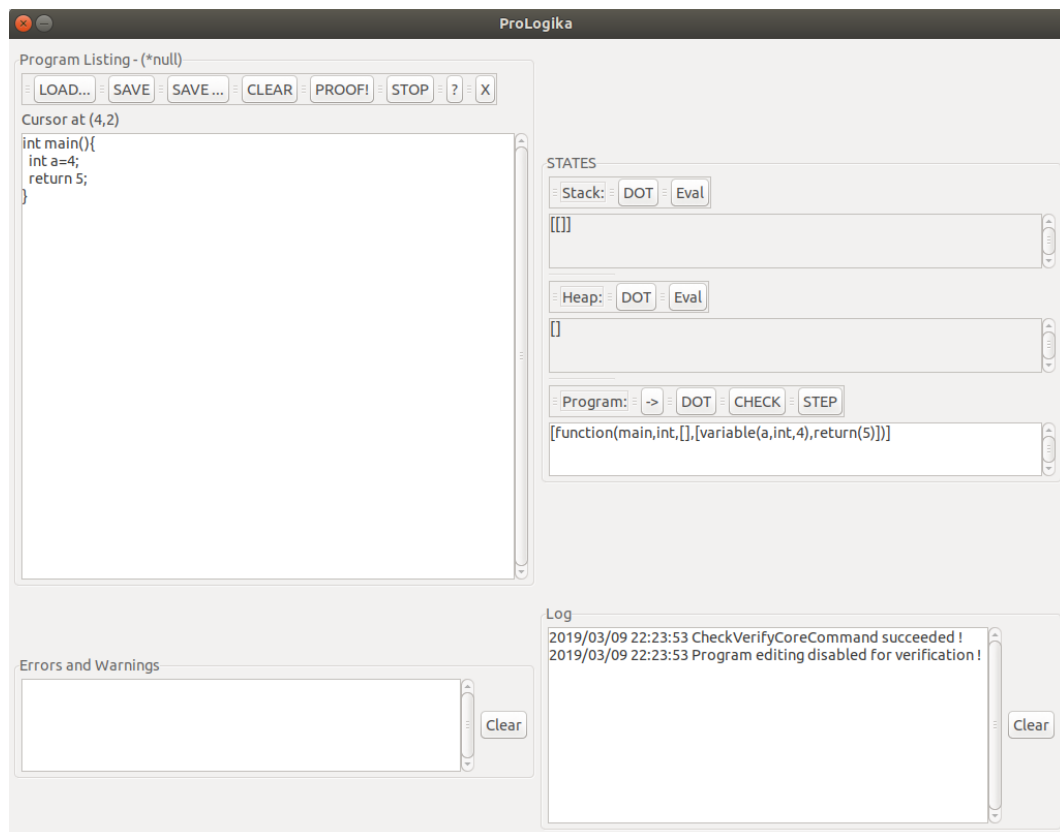


Рисунок 6.5: Графическая оболочка

Графическая оболочка (см. рисунок 6.5) состоит из трёх важных частей: (i) из левой части, которая содержит программу Си-диалекта, (ii) из правой верхней части, которая при выполнении программы содержит актуальное содержание динамической памяти и (iii) из правой нижней части содержащая программное представление в качестве термов Пролога. Содержание памяти и программное представление можно преобразовать в графическое представление. Кроме того, внизу имеются окна для ошибок и предупреждений, а также запись текущих операций верификации.

Разработка графической оболочки оказалась наиболее полезной в использовании, т.к. для автоматизации преобразования, отслеживания доказательств и представлений динамической памяти и правил верификации, вместе с программой, легче отслеживать, когда рядом имеются все необходимые данные.

## Use Cases

Программист в данной оболочке задаёт и редактирует программу (см. рисунок 6.6). Для этого имеются синтаксические проверки, а также программное представление может отобразить в графическом представлении в формате “DOT” в виде дерева представить.

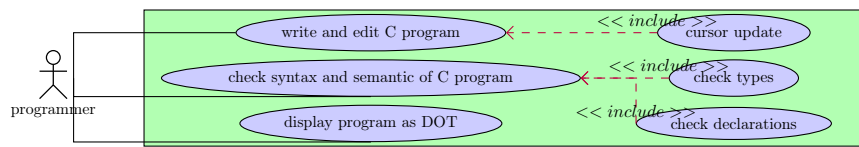


Рисунок 6.6: Перспектива программиста (UML Use Case)

Роль персоны, которая специфицирует программу (см. рисунок 6.7), отличается от программиста — формальными, точнее логическими, формулами описывается поведение программы. Спецификации можно в систему заносить, редактировать до и после загрузки программы. При запуске данные спецификации, которые размещены вместе с программой, проверяются. В случае возникновения ошибки, ошибка выделяется в правой нижней части. Проверка спецификации включает в себя при- и постусловия процедур, абстрактные предикаты, классные инварианты, но может также включать проверку полноты набора правил одного правила Пролога, касательно входной кучи.

Верификация кучи (см. рисунок 6.8) в отличие от редактирования программы заключается в вычислении и проверке имеющийся при выполнении программы состояния памяти с спецификацией. Для верификации необходимо проводить проверку сходимости куч, а также преобразование в нормализованную форму. Для упрощения и объяснения текущего доказательства имеются вспомогательные элементы, прежде всего, это визуализация дерева доказательства, генерация контр-примера, а также использование предыдущих доказательств для более быстрой сходимости.

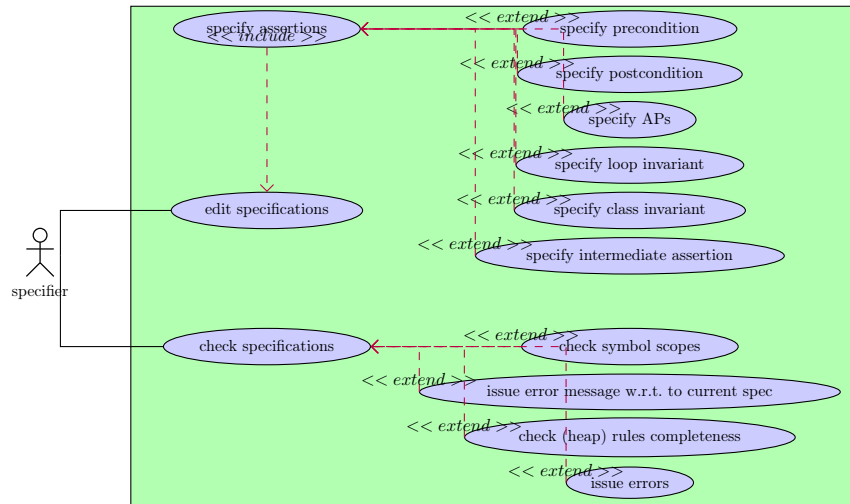


Рисунок 6.7: Перспектива спецификации программы (UML Use Case)

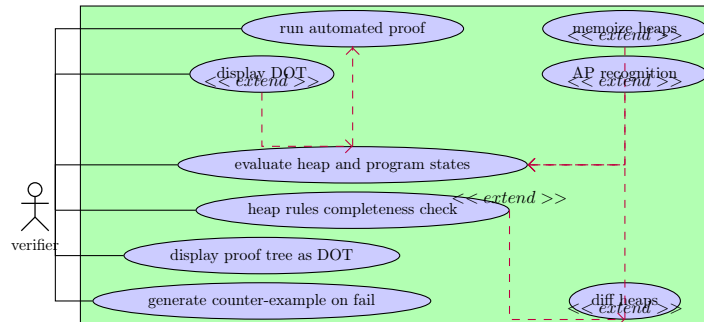


Рисунок 6.8: Перспектива верификации (UML Use Case)

### Пример) Реверс линейного списка

Дан список  $\{1,2,3\}$ , имеется указатель  $y$  на последний элемент линейного списка. Тогда реверс списка можно описать рекурсивно, отделив последний элемент и присоединив его к началу оставшегося списка. Таким образом, получается список как указано на рисунке 6.9.

Далее, получаем список как указано на рисунке 6.10.

### Классовые определения

Классовые экземпляры относятся в основном к классам Си++ или Ява и представляют упрощенный образ. Классы имеют разовый идентификатор, поля и методы. Поля и методы определяются в любом порядке и являются разовыми. Видимость для представления универсальности модели необязательна, но разрешается. Встроенные методы запрещаются, доступ к собственному классному экземпляру осуществляется с помощью `this`.

$\langle class\_definition \rangle ::= [ \langle class\_modifier \rangle ] 'class' \langle ID \rangle ' \{ ' \langle class\_fields\_methods \rangle ' \}$

$$x \mapsto 1 \circ x.n \mapsto 2 \circ \underbrace{x.n.n \mapsto \mathbf{nil}}_{\text{optional}} \parallel y \mapsto 3 \circ \underbrace{y.n \mapsto \mathbf{nil}}_{\text{optional}}$$

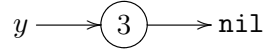
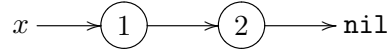


Рисунок 6.9: Пример линейного списка №1

$$x \mapsto 1 \circ x.n \mapsto 2 \circ \underbrace{x.n.n \mapsto \mathbf{nil}}_{\text{optional}} \circ y \mapsto 3 \circ y.n \mapsto x$$

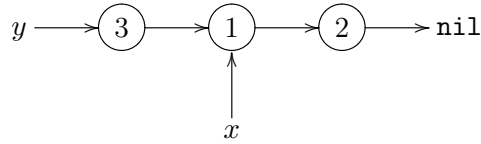


Рисунок 6.10: Пример линейного списка №2

$\langle \text{class\_fields\_methods} \rangle ::= \{ \langle \text{class\_field} \rangle \mid \langle \text{class\_method} \rangle \}^*$

$\langle \text{class\_field} \rangle ::= [ \langle \text{class\_modifier} \rangle ] \langle \text{variable\_declaration} \rangle \text{';'}$

$\langle \text{class\_method} \rangle ::= [ \langle \text{class\_modifier} \rangle ] \langle \text{function\_definition} \rangle$

$\mid [ \langle \text{class\_modifier} \rangle ] \langle \text{function\_declaration} \rangle \text{';'}$

$\langle \text{class\_modifier} \rangle ::= ( \text{'private'} \mid \text{'public'} \mid \text{'protected'} ) \text{'::'}$

Декларация типов следует в основном декларации по стандарту ISO-C++, которая разрешает неинициализированные и иницирированные переменные.

$\langle \text{variable\_declaration} \rangle ::= \langle \text{type} \rangle \langle \text{ID} \rangle \{ \text{' ,' } \langle \text{ID} \rangle \}^*$

$\mid \langle \text{type} \rangle \langle \text{ID} \rangle \{ \text{' ,' } \langle \text{ID} \rangle \}^* \text{'=' } \langle \text{expression} \rangle$

Определение функции осуществляется простым образом, т.е. неограниченные синтаксисом (variadic) параметры не допускаются. Конвенция вызовов подпроцедур следует схеме вызовов в языке Си. Тип возврата метода является либо **void**, либо базисным типом в данный момент. В ближайшее будущее объектные типы будут возвращаться, а до этого все изменения состояния вычисления передаются только с помощью параметров.

$\langle \text{function\_definition} \rangle ::= \langle \text{function\_declaration} \rangle \langle \text{block} \rangle$

$$\langle \text{function\_declaration} \rangle ::= \langle \text{type\_or\_void} \rangle \langle ID \rangle ' ( ( \langle \text{formal\_parameters} \rangle | ) ) '$$

$$\langle \text{type} \rangle ::= \text{'int'}$$

$$\langle \text{type\_or\_void} \rangle ::= \langle \text{type} \rangle | \text{'void'}$$

## Встроенные правила для автоматизированного логического вывода

**Нормализация куч** осуществляется с помощью встроенного предиката Пролога

`simplify/2`. Этот предикат принимает кучу в виде терма и возвращает терм кучи, в котором  $||$  оформляется крайней снаружи. Предполагается, для быстрой обработки, что входная куча уже  $||$ -нормализована, т.е. в виде  $q_0 || q_1 || \dots || q_k$ , где  $\forall j. q_i$  в виде  $X_i \circ X_{i+1} \circ \dots \circ X_k$ .

**Высчитывание** (“множественное сравнение”) осуществляется с помощью `subtract`. Этот встроенный предикат принимает список термов Пролога и генерирует список не хватающих термов кучи. Предикат может быть использован для установления полноты данного семейства предиката (например, важно для упрощения, трансформации и для обработки с тройками Хора в общем).

**Сравнение** производится покомпонентно с помощью двух термов: имеющейся и ожидаемой кучей.

## Слойная архитектура верификатора

Предложенная архитектура верификатора “ProLogika” [306] состоит из 6 главных слоёв (см. рисунок 6.11), которые на периферии сотрудничают с библиотекой логического вывода “*tuProlog*” (2p), а также с генератором синтаксических анализаторов ANTLR в версии 4. Система реализована на языке Ява, хотя одновременно она тесно связана с реализацией Пролога и максимально близка к GNU Prolog. ANTLR может генерировать выходной код не только Ява, но также разных других языков программирования. Таким образом, обеспечены расширяемость и вариабельность системы “ProLogika”. “*ANTLR*” принципиально может быть заменено на любой другой синтаксический анализатор эквивалентной мощностью, под условием, что не только стартовый нетерминал может быть распознан, но также подграмматики используемых нетерминалов.

Система верификатора динамической памяти “ProLogika” состоит из следующих пакетов: `internal`, `output`, `parsers`, `prolog`, `frontend` и `gui`. `gui` предоставляет графическую оболочку и управленческие к ней модули. Результаты, промежуточные представления вычисления и визуализации выявляются в самой оболочке или могут быть запущены с ней.

Пакеты `frontend` и `parsers` тесно взаимосвязаны. Пакет `frontend` содержит интерфейсы и модули различного уровня абстракции для лексического и синтаксических анализаторов. Параметризация синтаксического анализа является одной из основных верификации абстрактных предикатов. `parsers` содержит изначально генерированные лексические и синтаксические анализаторы, а также анализаторы, генерируемые при запуске верификации. Интерпретация абстрактных предикатов

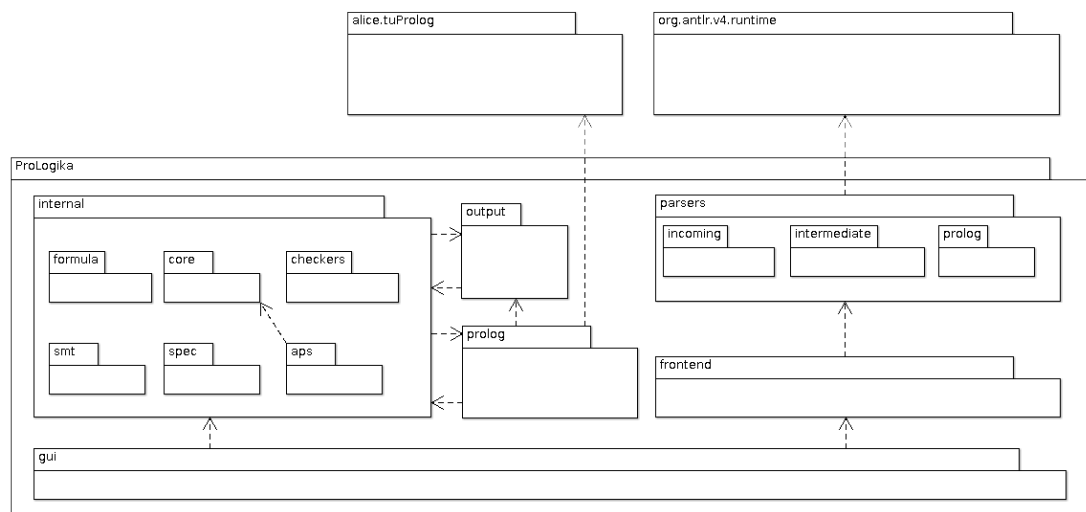


Рисунок 6.11: Архитектура верификатора

входит в пакет `ProLogika.internals.aps`.

Пакет `prolog` обогащает встроенную систему логического вывода с дополнительными предикатами Пролога, например, для обработки термов куч, анализ и выявление ошибки, а также для реализации дальнейших побочных эффектов. Реализация дополнительных встроенных предикатов осуществляется таким дизайном, который гибок и легко может быть расширен. Правила Пролога могут быть всегда добавлены и расширены во время запуска программы верификации.

Пакет `output` предоставляет вспомогательные функции для писания термовых представлений, например, в DOT-файл или на консоль. На данный момент предусмотренная генерация OCL-кода не реализована.

Пакет `internal` производит саму спецификацию и верификацию связанных функций. Пакет `core` содержит различные модули, они связаны с обработкой термовых представлений куч. Пакет `aps` содержит все необходимые модули, которые связаны с абстрактными предикатами, следовательно, он связан с пакетом `core`. `checkers` производит проверку куч до, после и во время запуска программного оператора. Остальные пакеты подлежат сильному исправлению и находятся в настоящее время на стадии разработки.

## Компоненты

В соответствии с разделом 6.8 можно выявить 6 главных компонентов верификатора (см. рисунок 6.12): (i) `ProLogika.GUI` отвечает за графическую оболочку, (ii) `ProLogika.parsers` отвечает за всё, что связано с анализатором синтаксиса, включая проверки условий синтаксиса и семантики, а также за обработку абстрактных предикатов (пакеты `parsers: incoming`, `intermediate` и

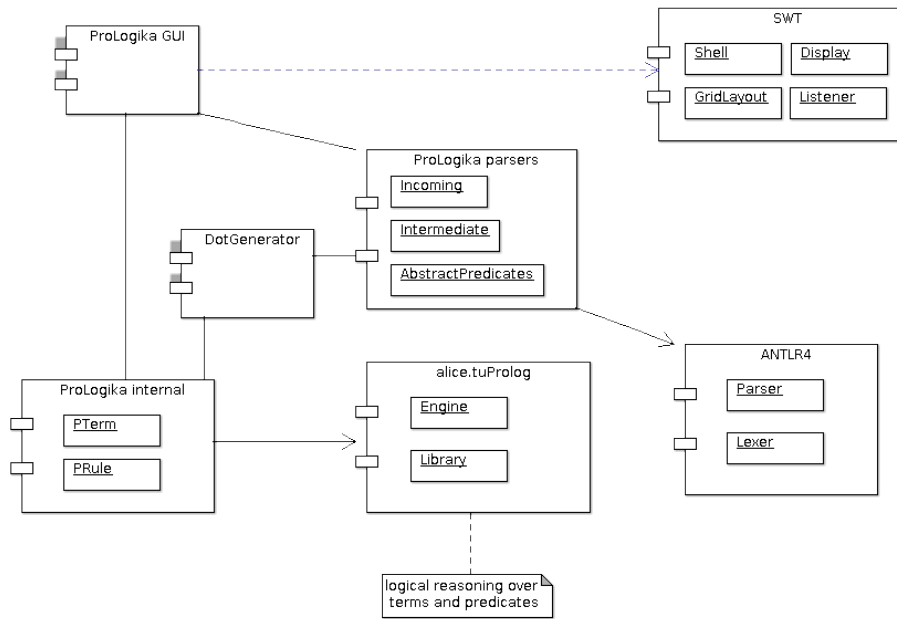
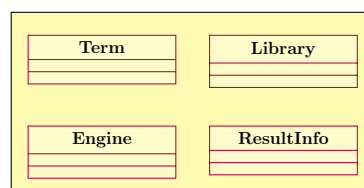


Рисунок 6.12: Компоненты верификатора

**AbstractPredicates**), (iii) библиотека Java **SWT** тесно связана с компонентом **ProLogika.GUI** и введён для поддержки платформы независимой от ОС, (iv) **ANTLR** компонент для построения языковых процессоров, (v) **alice.tuProlog** представляет библиотечный компонент для реализации генерического логического вывода, построен на основе Пролога, (vi) **ProLogika.internal** компонент представляющий Прологовские термы (**PTerm**) и правила (**PRule**, **PSubgoal**), которые используются другими компонентами.

## Пакеты

Рисунок 6.13: Пакет **alice.tuProlog**

Пакет **alice.tuProlog** (см. рисунок 6.13) содержит: (i) класс **Term**, который представляет собой общий Прологовский терм, который не является абстрактным, (ii) **Library** является Прологовской библиотекой, которая может содержать целую формальную теорию Прологовских правил. **BuiltinLibrary** является реализацией класса **Library**. **Engine** контейнер, который может производить логический вывод правил Хорна согласно данной теории. **ResultInfo** представляет совокупность любого количества результатов данному запросу (подцелей).



Пакет `parsers`:

Пакет `parsers` (см. рисунок 6.14) содержит пакеты: `incoming`, `intermediate`, `prolog`, а также два класса `MetaType` и `MetaTypeManager`. `MetaType` представляет определение типа во входной программе, например, класс или целое число (`integer`). `MetaTypeManager` является наблюдающей за определениями инстанций.

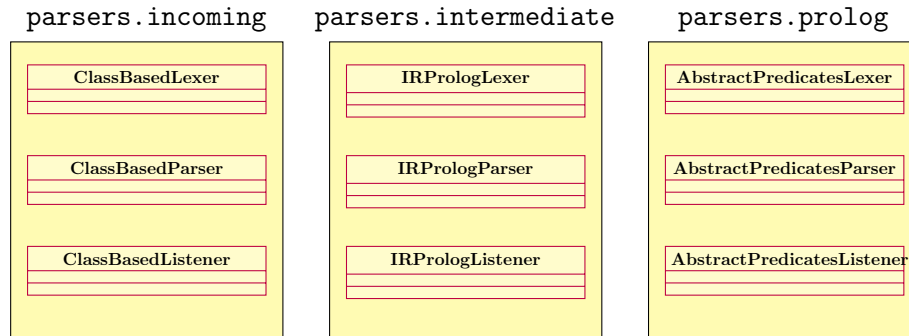


Рисунок 6.14: Пакет `parsers`

Пакет `internal`: Пакет содержит пакеты: `core`, `checkers`, `aps`, `spec`.

Пакет `internal.core`:

Пакет `core` (см. рисунок 6.15) содержит классы, которые представляют ядро модели логического вывода: термы, правила, символьные переменные, подцели, и т.д. Префикс `-P` ссылается на “*Prolog*”, классы строго соответствуют элементам ISO-Пролога.

Главными базисными элементами являются: `PTerm`, `PRule` и `PSubgoal`. Остальные `P`-классы соответствуют Прологу, ради исключения классу `Unparsed`, который частично вычисляемый терм, который полностью определён на более поздней стадии вычисления.

Пакет `internal.checkers`:

Проверки проводятся вместе с нормализацией и высчитыванием правил во время верификации, например во время анализа данного абстрактного предиката (`APCompletenessChecker`, `HeapCompletenessChecker`, `PrologAPsChecker`) (см. рисунок 6.16). Синтаксические анализы также проводятся во время анализа входной программы и проверки вставленных в программу спецификатором утверждения (например с помощью `ANTLRGrammarChecker`). Все проверки могут осуществляться, если реализовать интерфейс `CheckerInterface` конкретными методами.

Пакет `internal.aps`:

Пакет `aps` (см. рисунок 6.17) содержит `ANTLRBuilder`, `ANTLRLauncher`, `GrammarBuilder`, а также все остальные классы, которые обрабатывают абстрактные предикаты (APs) согласно формальной

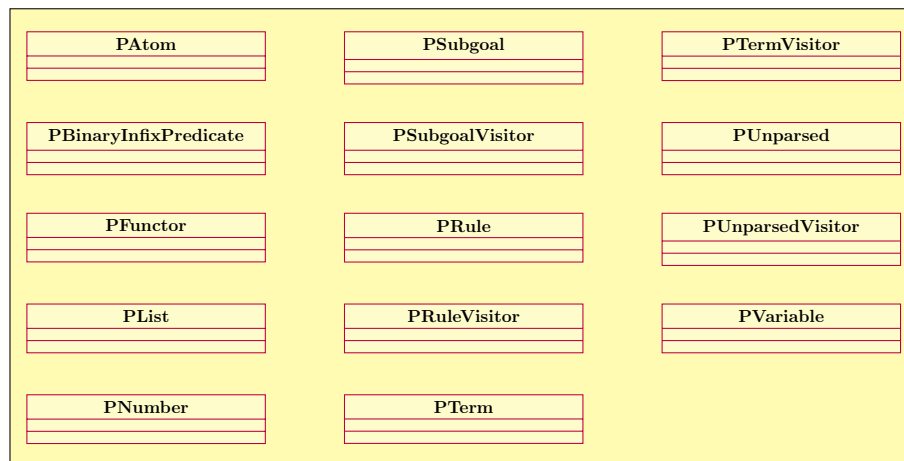


Рисунок 6.15: Пакет core

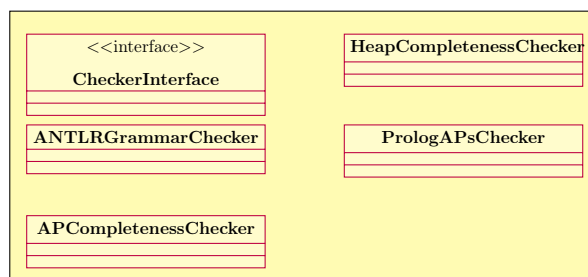


Рисунок 6.16: Пакет internal.checkers

атрибутируемой грамматике (как было введено Кнудом и Вегнером). `ANTLRBuilder` является специализацией класса `GrammarBuilder`. “*ANTLR*” используется изначально лишь как один пример генератора синтаксического анализа. Любой генератор или вручную написанный анализатор может послужить заменой, если предоставить необходимый интерфейс.

Аналогично выше сказанному, `ANTLRGrammar` является специализацией класса `FormalGrammar`. `ANTLRBuilder` способствует к построению анализатора анализаторов.

`ANTLRLauncher` производит вызов драйвера ANTLR-генератора (для производства различных функций ANTLR).

## Синтаксические анализаторы

Класс `Main` (см. рисунок 6.18, связан с `Parser` из рисунка 6.19 и `SyntaxAnalyzer` из рисунка 6.20) управляет тремя инстанциями синтаксического анализа согласно формальным грамматикам: (1) `ClassBased.g4`, (2) `IRProlog.g4`, и (3) `AbstractPredicates.g4`.

Пакет `frontend`: см. рисунок 6.21.

Замечания: `IRPrologCommand.execute()` возвращает `IRPrologDescription` object.

Имеется паттерн:

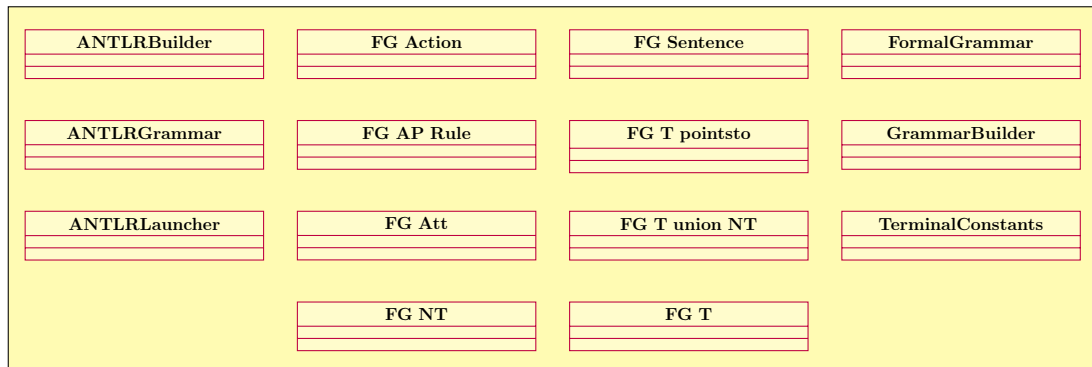


Рисунок 6.17: Пакет internal.aps

```
ADAPTER::(CLASS) ADAPTER:
```

```
    SyntaxAnalyzerCommand
```

```
ADAPTER::ADAPTEE:
```

```
    ClassBasedCommand, IRPrologCommand, AbstractPredicatesCommand
```

```
Package parsers.incoming:
```

Следующая структура применима не только для `parsers.incoming` (см. рисунок 6.22), а также для `parsers.intermediate` и `parsers.prolog`.

Чтобы модифицировать синтаксический анализ при входе и выходе из правила нетерминала (*срёртка*) были введены классы с суффиксом `Listener`.

Пакет `parsers`: см. рисунок 6.23.

Класс `MetaType` представляет встроенный или само-определяемый тип, который применяется при проверке типов над выражениями для входной программы. Каждый тип разово идентифицируется с помощью поля `typeID`. Однако, при построении типов `typeID` не присваивается изначально (только имя, но без полной сигнатуры, т.к. рекурсивные типы также разрешаются). Статически определёнными, встроенными типами являются: `INTTYPE` и `VOIDTYPE`.

Элементы пакета `parsers.intermediate` выполняют паттерн-роль “*директора*” для класса `MetaTypeManager`.

Пакет `frontend` см. рисунок 6.24.

Замечания:

```
STRATEGY::TEMPLATE:
```

```
    SyntaxAnalyzer.launchMainNT()
```

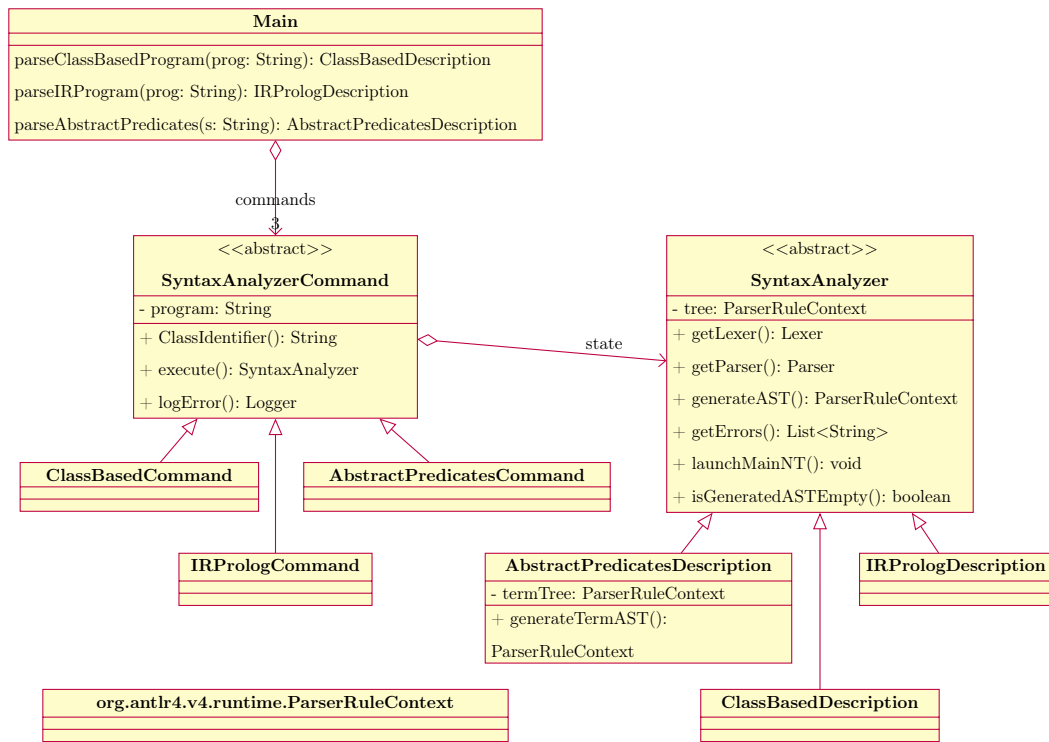


Рисунок 6.18: Класс Main

STRATEGY::CONCRETE STRATEGY:

ClassBasedDescription.generateAST()

IRPrologDescription.generateAST()

AbstractPredicates.generateAST()

Метод SyntaxAnalyzer.launchMainNT выглядит следующим образом:

```

prelaunch();
this.tree=generateAST();
postlaunch();

```

В классе IRPrologDescription get-методы означают доступ к синтезированным атрибутам соответствующей формальной грамматики.

Класс ParserRuleContext содержит всё то, что употребляется наследованным и синтезируемым атрибутами.

## Прологовские термины

Пакет internal.core см. рисунок 6.25.

Этот пакет содержит все представления всвязи с терминами, подделями и правилами.

COMPOSITE::COMPONENT:

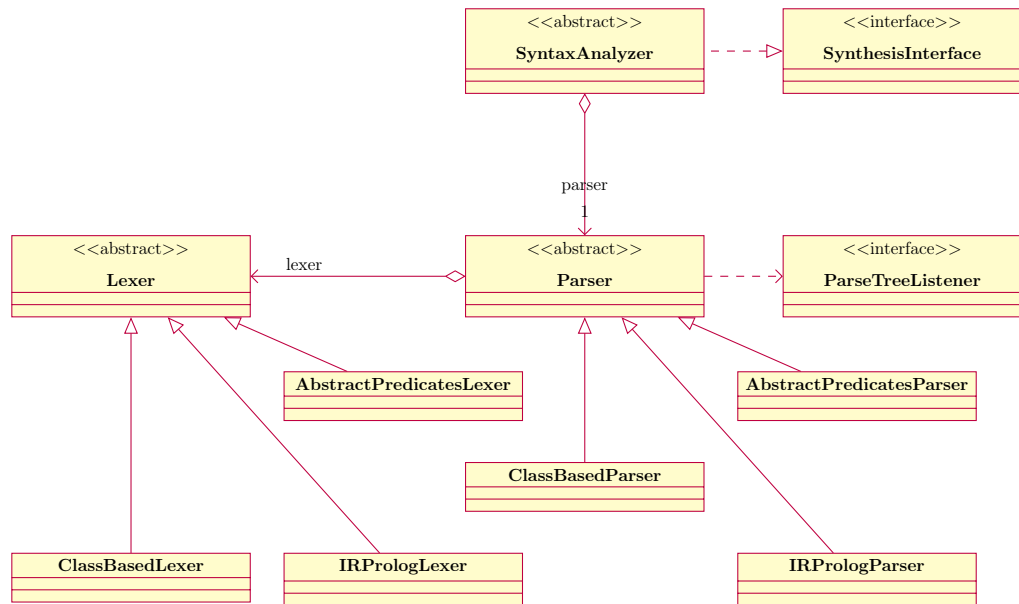


Рисунок 6.19: Класс Parser

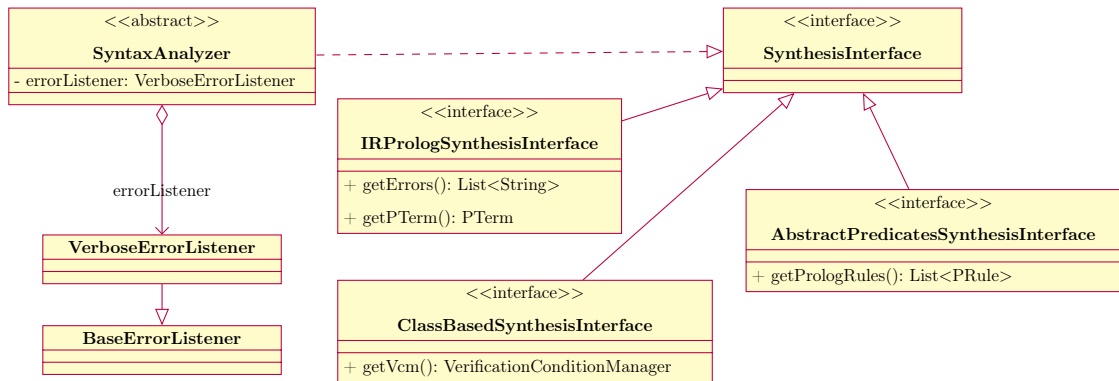


Рисунок 6.20: Класс SyntaxAnalyzer

PTerm

COMPOSITE::COMPOSITE:

PTerm

COMPOSITE::CONCRETE COMPOSITE:

PAtom, PVariable, PFuncutor, PNumber, PList

PAtom.getChildren возвращает null. PAtom.setChildren игнорирует множество и при необходимости выделяет предупреждение на консоль. PSubgoal представляет подцель, которая далее может содержать PTerm как составляющее (не подцелей).

Замечания:

В пакете internal.core (см. рисунок 6.26) нет необходимости в PSubgoal, т.к. оно уже содержится в PRule, а также потому, что подцели являются частями запросов (которые не являются частью грамматики построенных абстрактных предикатов).

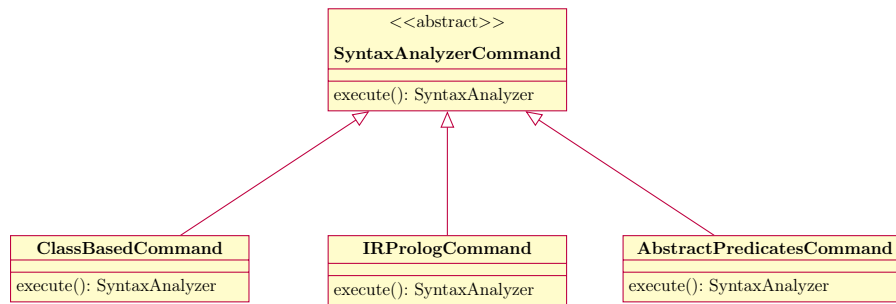


Рисунок 6.21: Пакет frontend

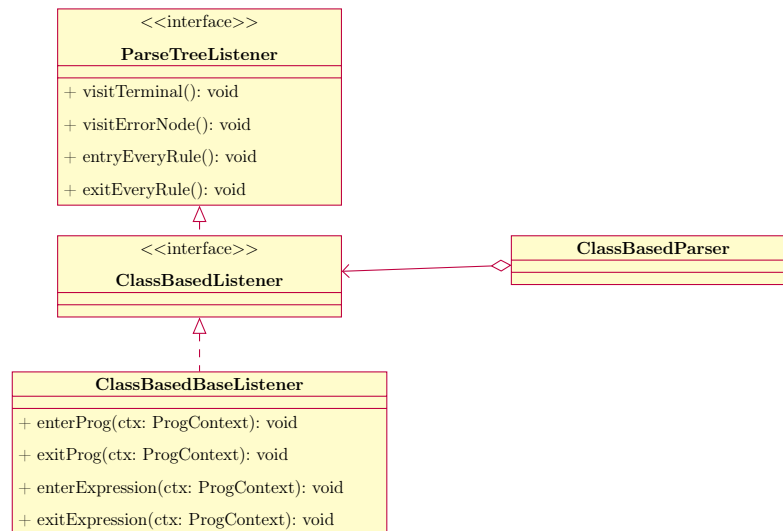


Рисунок 6.22: Пакет parsers.incoming

Метод `GrammarBuilder.loadRules` читает все `PRule` и заносит их в формальную грамматику, при этом, атрибуты остаются как в Прологе. Класс `GrammarBuilder` реализует оба интерфейса, `PTermVisitor` и `PRuleVisitor`, т.к. необходима интерпретация правил и должен иметься один нетерминал в качестве стартового символа грамматики.

`PTerm` и `PRule`-экземпляры генерируются в зависимости от экземпляров описаниях (Description-objects).

### Абстрактные предикаты

Абстрактные предикаты необходимы для представления и логической обработки в рамках верификации правил Хорна.

Используемые анализаторы имеют один начальный символ нетерминал, однако интерфейс анализатора должен предоставлять возможность распознавать одно или более того нетерминалов данной формальной грамматики, в данном случае (ANTLR-)грамматики.

Пакет `internal.apr` см. рисунок 6.27.

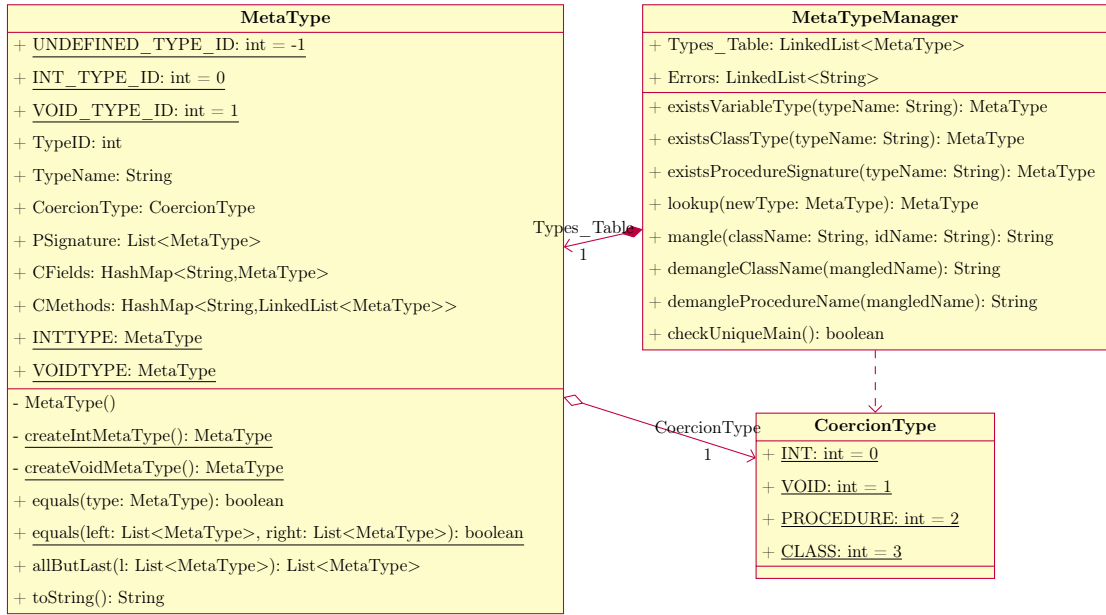


Рисунок 6.23: Пакет parsers

Замечания:

Все классы вовлечены в представление формальной грамматики (далее рассматривается возможность реализации на примере ANTLR).

`PRule` может быть представлено в виде `FG_AP_Rule`, а также обратно, следующим образом:

$$p1(X, pointsto(X, value1)) :- \dots$$

$$\Leftrightarrow$$

$$p1_{x1,x2} \rightarrow \{X_1 \approx X\} \{X_2 \approx pointsto(X, value1)\}, \dots$$

Далее, `FG_NT` имеет атрибуты `List<PTerm>`, но только не `List<FG_Att>`, т.к. оно ссылается только на существующий нетерминал с конкретной последовательностью `PTerm`, согласно данному порядку определения атрибутов.

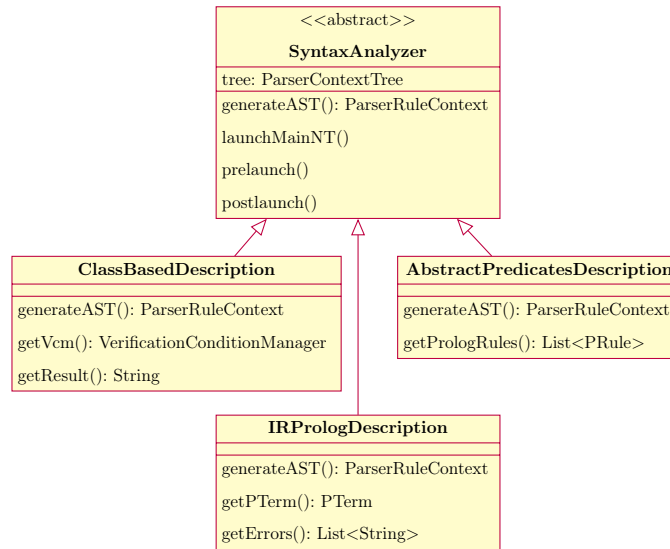


Рисунок 6.24: Пакет frontend

## Термы кучи

Прологовские термы определены индуктивно над операторами  $\{\circ, ||\}$ . Полученный терм является итогом предыдущего синтаксического анализа.

Пакет `internal.ht` см. рисунок 6.28.

Подцели в абстрактных предикатах могут создавать связанный (под-)граф (ради простоты на данный момент, в общем случае с помощью проверок свойства  $a \circ a$ ).

$a_1 \circ a_2 \circ a_3$  может быть записан непосредственно, используя  $a_1, a_2, a_3$  (как часть абстрактного предиката) или как абстрактные предикаты  $a_x : -a_1, a_2.$ , где оно же далее может опять же выступить в качестве некоторой подцели:  $..., a_x, a_3, ....$

Пример 2)  $a_1 \circ (a_2 || a_3)$  можно перезаписать, как абстрактный предикат как:  $a_x : -a_2, disj, a_3.$ , где имеется некоторая последовательность как часть абстрактного предиката  $..., a_1, a_x, ....$

Замечания:

`HeapTerm` является внутренним представлением кучи на основе ЛРП.

Термы кучи над  $\{\circ, ||\}$  может также содержать частичные спецификации. Терм кучи строится пошагово при анализе данных абстрактных предикатов.

`HeapTerm.simplify` ссылается на `BuiltinLibrary.simplify_1()`. Упрощения термов кучи осуществляется с помощью перегруженных встроенных предикатов Пролога `simplify_2`.

Граф кучи представленный термом кучи, может также описывать не полностью специфицированные части с помощью Пролог-оператора “`_`”. Нормализация данных конъюнктов заключается в лексикографической сортировке порядка всех левых сторон базисных куч.



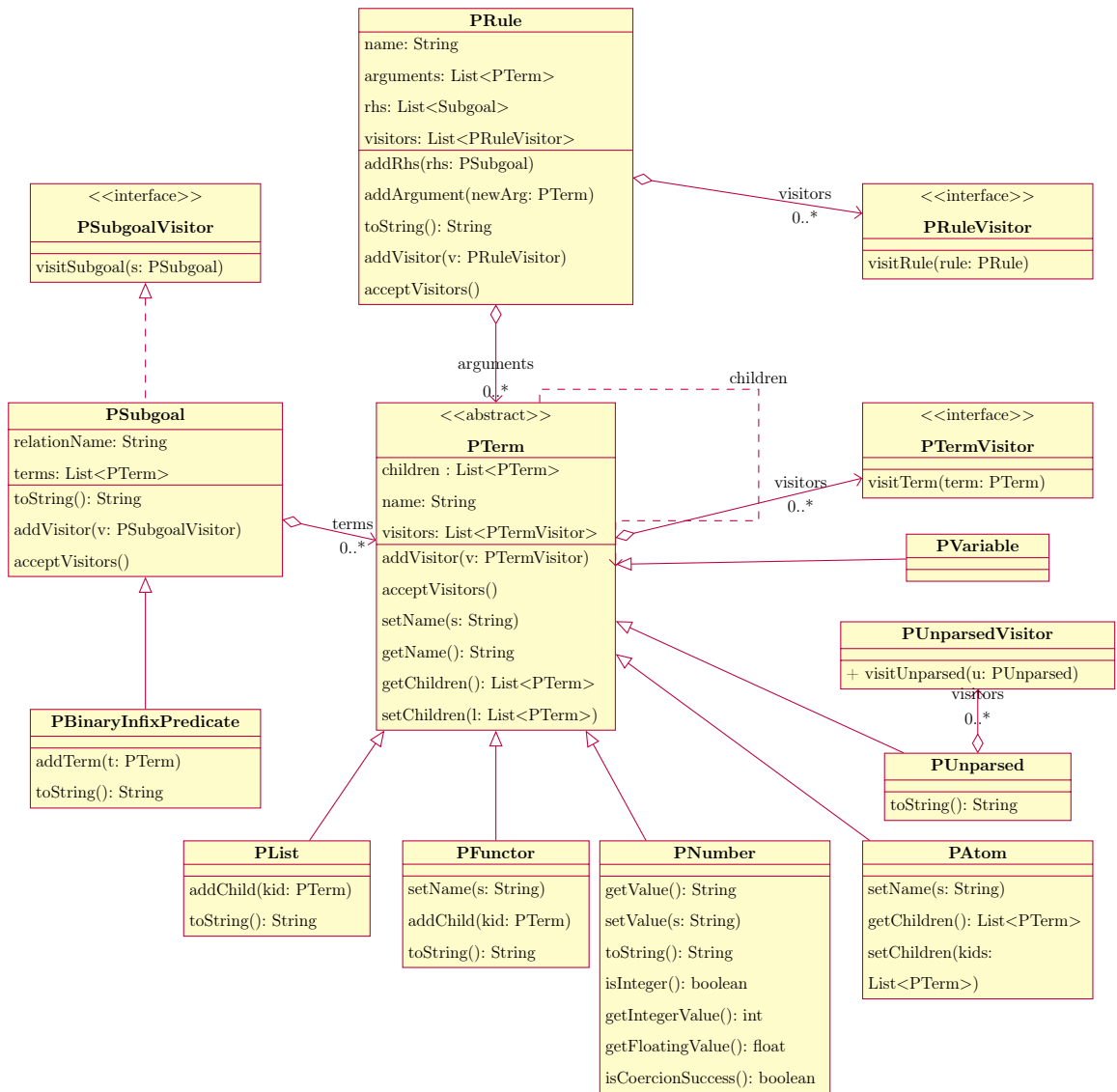


Рисунок 6.25: Пакет internal.core

**DotGenerator** генерирует векторную графику на основе DOT-формата или в качестве его представления в качестве одной строки, которую позже можно записать в файл, например.

Замечания:

Проводится проверка корректности, например (i) соблюдается ли неповторимость кучи, (ii) связана ли куча, и т.д.

Пакет **prolog** см. рисунок 6.29.

Библиотека **BuiltinLibrary** содержит все Прологовские предикаты, которые становятся доступными для логического анализатора на основе Пролога. Если распознавание одного абстрактного предиката будет завершено целиком, тогда проводится проверка  $a \circ a$ .

Замечания:

Пакет frontend

Пакет

internal.core

Пакет

internal.aps

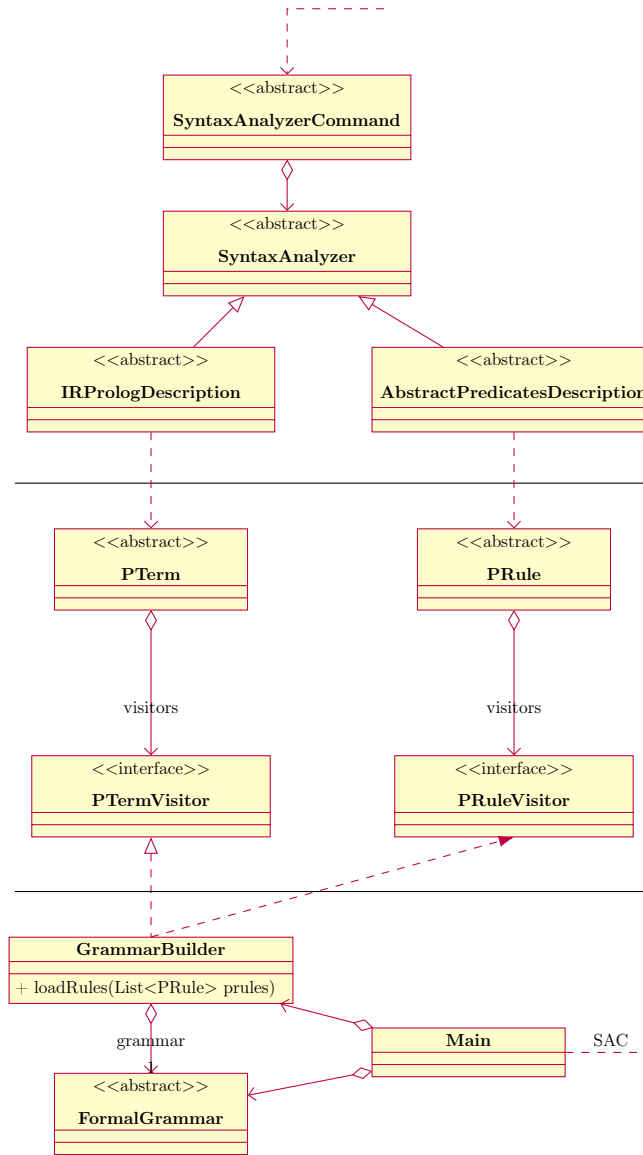


Рисунок 6.26: Слои пакетов frontend, internal.core, internal.aps

`BuiltinLibrary:disj_0` совершает один шаг в верификации динамической памяти. Для этого могут вводиться специальные метки в случае не достижения границ куч.

`BuiltinLibrary:unify` предоставляет замену классической унификации термов ( $=/2$ ) с помощью и без проверки повторений.

Класс `HeapTerm` ссылается на `BuiltinLibrary:simplify_1`.

`BuiltinLibrary::serialize_1` почти эквивалентна к `serialize_2`, ради исключения, что содержимое выдается на консоль.

`StackState` необходимо для проверки диапазонов годности (например, является ли символ частью данного контекста, когда все остальные семантические проверки уже успешно завершились).

Сигнатур имеет тип:  $\sigma :: String \rightarrow Type \rightarrow Value$ .



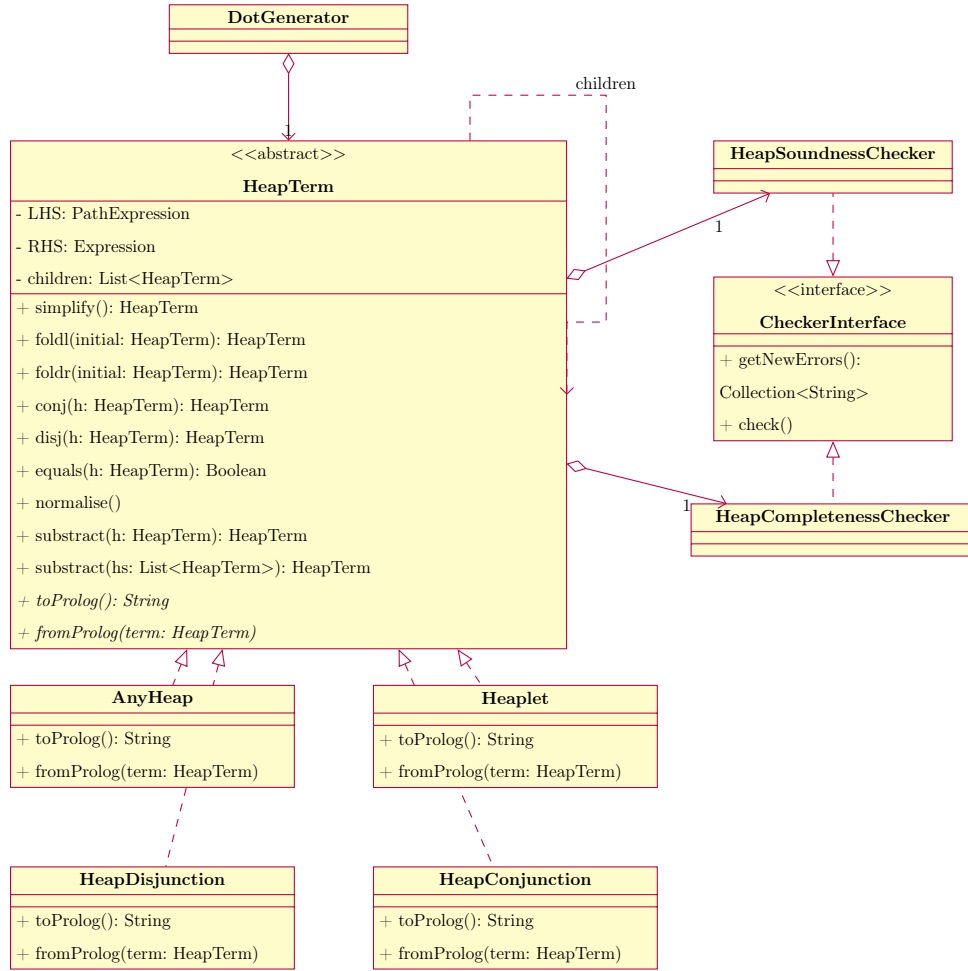


Рисунок 6.28: Пакет internal.ht

не на первом месте. На первом месте стоят вопросы сложности и выразимости. Таким образом, умышленно используется не самый быстрый способ верификации.

В отличие от предыдущих подходов, представлен новый подход, отличающийся резко тем, что символы больше не являются просто локальными переменными. Символы как логические используются произвольно в логических термах и предикатах. Язык утверждений и верификации теперь совпадают при использовании диалекта «Пролог». Теперь дополнительные преобразования для обеих сторон отпадают, также как и имеющиеся ограничения в предыдущих подходах (см. [32]).

Открытым вопросом остаётся, как *правила ошибки* могут быть эффективно подключены к генерации контр-примеров и какие имеются способы для отслеживания ошибок в связи с этим? Из-за приостановки при первой ошибке, все остальные ошибки не рассматриваются, если *настоящей* ошибкой по происхождению является одна из последовательных. Поэтому предлагается рассматривать «*конечный автомат*» для сравнения и минимизации «*прописной дистанции*» между имеющейся и ожидаемой кучами, аналогично глобальному *алгоритму Левенштейна* [287],[264],[11] для вычитания минимального количества операций замены за  $\Theta(n^3)$ , удаления и вставления подкуч. Метод Левенштейна может быть не обобщён, но модифицирован и расширен к деревьям [230],[284],[10].

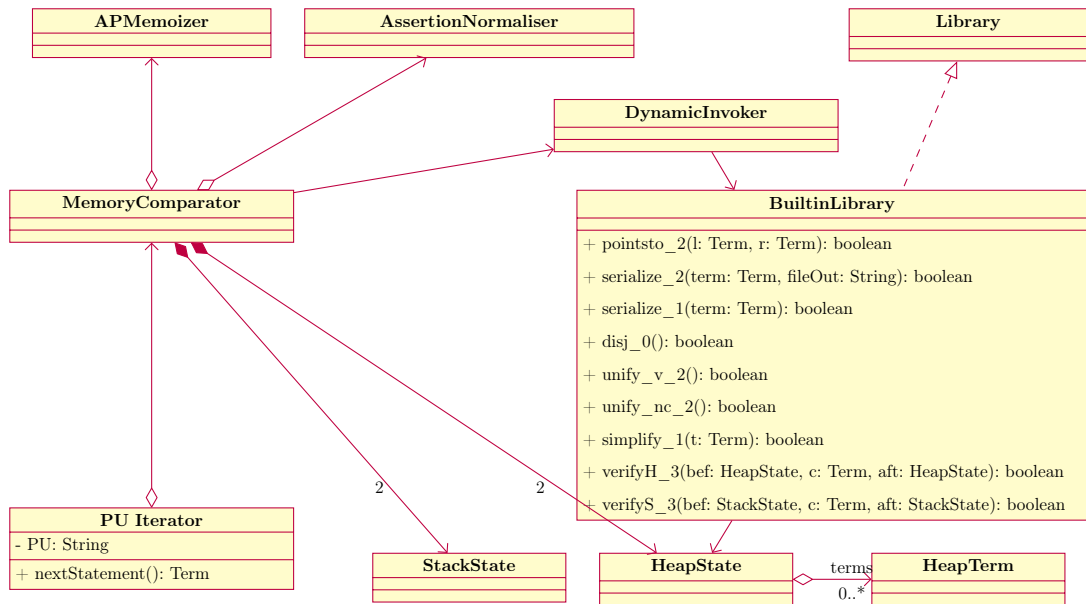


Рисунок 6.29: Пакет Prolog

«Метод паники» [110],[197] который при возникновении ошибки во время синтаксического анализа пытается продолжить перебор с помощью обнаружения самого ближнего следующего состояния, гарантированно верное и стабильное при удалении всех «лишних» токенов.

Далее предлагается рассматривать вопрос о частичных спецификациях, как константных функций **emp**, **true** или **false** для достижения упрощённой верификации (см. следующие главы). В частности, задаётся вопрос, можно ли с помощью частичных спецификаций выявить недостижимые мосты, которые предположительно связывают независимые кучи для исправления имеющейся спецификации.

---

**Алгоритм 2** Наивный алгоритм для проверки равенства абстрактных предложений. **Вход:**  $\alpha_1 = [i_0, \dots, i_{m_1}, p_0, \dots, p_{n_1}]$ ,  $\alpha_2 = [j_0, \dots, j_{m_2}, q_0, \dots, q_{n_2}]$  с  $i$  и  $j$  как **pointsto**-терминалы, и подцели  $p$  и  $q$  как нетерминалы.  $\Gamma$  содержит все определения предикатов. **Результат:** «Истина» в случае  $\alpha_1$  равно  $\alpha_2$ , «Ложь» иначе.

---

```

1: procedure SHIFT-TERMS( $\Gamma, \alpha_1, \alpha_2$ )
2:   for all  $k \in \{0, \dots, m_1\}$  do
3:     if  $\exists l. l \in \{0, \dots, m_2\} \wedge i_k \approx j_l$  then
4:        $\alpha_1 \leftarrow \alpha_1 \setminus i_k$ 
5:        $\alpha_2 \leftarrow \alpha_2 \setminus j_l$ 
6:     end if
7:   end for
8: end procedure

9: procedure REDUCE-PREDS( $\Gamma, \alpha_1, \alpha_2$ )
10:  for all  $k \in [0..n_1]$  do                                      $\triangleright$  сравни терминалы
11:    if  $\exists l. l \in [0..m_2] \wedge j_l \in \pi(p_k)$  then
12:       $expand(p_k, \alpha_1)$ 
13:    else                                                          $\triangleright$  редуцируй в  $\alpha_2$ , если возможно
14:      if  $\exists l. l \in [0..n_2] \wedge (\pi(q_l) \cap \pi(p_k) \neq \emptyset)$  then
15:         $\alpha_1 \leftarrow expand(p_k, \alpha_1)$ 
16:         $\alpha_2 \leftarrow expand(q_l, \alpha_2)$ 
17:      else                                                          $\triangleright$  Совпадение
18:        if  $m_1 = m_2 = n_1 = n_2 = 0$  then
19:           $true \rightarrow Halt!$ 
20:        else                                                          $\triangleright$  Нет совпадения
21:           $false \rightarrow Halt!$ 
22:        end if
23:      end if
24:    end if
25:  end for
26: end procedure

```

---

## 7 Заключение

«*Значимые результаты*» Во время работы были получены результаты, которые можно классифицировать: в области автоматизации доказательств:

1. Анализ разрыва между языками спецификации и верификации динамической памяти и анализ его сближения.
2. Анализ и измерение выразимости трансформаций термов в логической парадигме на примере диалекта Пролога. Выявление, что логическое описание утверждениями из символов, переменных и термов в общем, а также реляций, естественно лучше приспособливает и ближе к верификации задач динамической памяти в Прологе.
3. Термы Пролога как главное промежуточное представление на протяжении всего конвейера статического анализа проблем динамической памяти.
  - Обоснование и использование логического языка в качестве спецификации и верификации. Главная мысль — логическое программирование для решения верификации теорем.
  - Предложенная платформа, изначально построена как открытая для расширений и вариаций. Платформа позволяет включение дополнительных экстерных библиотек и логических решателей в Прологе.
  - Исследование демонстрирует преимущество Пролога.
4. Предложение об автоматизации верификации куч, как синтаксический перебор абстрактных предикатов, основано на пошаговой обработке граней графа кучи.

в области выразимости языков спецификации и верификации:

1. Теперь доказуемые абстрактные предикаты могут без ограничения содержать символы и переменные согласно правилам Хорна, что расширяет сегодняшние ограничения. Теперь предикаты могут иметь потенциально любые рекурсивные определения, где выбранный метод синтаксического перебора может ограничить рекурсию, возможно потребовав переоформление правил.
2. Повышение выразимости, благодаря ужесточению операций над кучами. Имеется строгое соотношение между выражением кучи и ее графом. Теперь пространственные операторы куч

формируют однозначные термы утверждения. Доказываются алгебраические свойства моноида и группы, которые позволяют определение вычислений над кучами. Предложение о расширении и возможной стандартизации «*UML/OCL*» указателями.

3. Повышение выразимости и полноты, благодаря частичной спецификации куч переменных и объектных экземпляров. Сравнимость теперь позволяет все входные кучи одного набора сравнить и «*ямы*» или «*пересечения*» эффективно вычислить. Теперь правила могут быть написаны, охватывая потенциально больше случаев.
4. Теоретическая возможность, исходя из ужесточенного представления, выявить и проверить данную входную программу на минимальность для покрытия граф кучи полностью («*проблема содержимого*») при исключении определений абстрактных предикатов в общем случае.

#### Выводы:

1. Единый язык спецификации и верификации. Унификация языков спецификации и верификации к логическому языку программирования диалекту Пролога представляется как одна из крайних мер борьбы с многозначностью и позволяет решить поставленные перед собой проблемы выразимости и полноты. Чем меньше имеется языков и особенностей, тем меньше требуется рассматривать. Особенно, когда языки спецификации и верификации являются по синтаксису и семантике простым единым языком. Процесс верификации представляется как сравнение имеющейся и должной кучи.

По ходу исследований существующих средств верификации динамической памяти и соответствующих проектов в качестве примеров, мною были разработаны «*shrinker*» и «*builder*» в качестве вспомогательных программных систем. Итогом расследования оказалось, что проблемы решённые верификаторами на практике крайне редкие или применимы только в случаях, для которых они были разработаны, а применение в других областях практически исключено.

2. «Нет» новым и дубликатным определениям. Использование Пролога позволяет избежать введение всё новых конвенций, определений, языков спецификации, программирования, верификаций и даже сред. Отсутствует необходимость введения определений, которые оказывались в прошлом лишними и дубликатными. Выявлены важные свойства и критерии, которые учтены при построении платформы. Теперь кучи и части могут быть выбраны термами и символами. Абстрактные предикаты представляются обыкновенными прологовскими правилами. Используется промежуточное термовое представление, в качестве входного языка, а также разрешается Си-диалект, либо любой другой императивный язык программирования. Также допускается полное отсутствие входного языка, в таком случае, разрешается непосредственная передача промежуточного представления в ядро верификации.



3. Повышенная выразимость в связи с символами на основе Пролога. Он основан на и реализует предикатную логику в качестве модели и реализации. Предикаты куч стали по-настоящему логически абстрактными (и не «Абстрактными»), а именно прологовскими предикатами. В предикате символы допускаются в любых местах и ограничения касательно символьного использования отсутствуют. Логический вывод производится символами, логические выводы не возвращаются, благодаря унификации и реализации стека с возвратом на основе машины Уоррена. Проводилось обсуждение, что граф кучи можно полностью выразить. Объекты представляются как кучи, методы классов специфицируются не обязательно пред- и постусловием аналогично к процедурам. Правило фрейма способствует аналогичному поведению у процедур и позволяет включение в фрейм утверждения по всему циклу существования объекта. Объекты могут быть переданы в качестве символьного параметра абстрактным предикатам, таким образом, повышается модульность и благодаря неполным константным функциям неполный объект необходимо специфицировать.
4. Реляционная модель правильна и нужна для эффективного описания абстрактных предикатов. Реляционная модель в отличие от других представлений была в итоге выбрана как наиболее удобная для описания и проверки куч. Также упоминают источники из списка литературы. Хотя численный анализ трудный, из-за выбора адекватных примеров и численной значимости, всё равно, проведённая экспертиза показала, что запись в прологовских терминах для проверенного набора примеров всегда лучше, а в среднем арифметическом на 30% компактнее и более гибкая, чем например эквивалентная функциональная запись. Приведены типичные примеры, которые не могут быть эффективно или точно представлены в функциональных или императивных парадигмах.
5. В Прологе предложена платформа конвейера. Платформа предлагает крайне простую архитектуру по обработке входного языка и правил верификации. Платформа основывается на Прологе, на конвейере фаз по обработке динамической памяти, которые могут индивидуально меняться и добавляться. Платформа отличается принципами: (а) автоматизацией, (б) открытостью, (в) расширяемостью и (г) понятностью. Выводимо только то, что выводимо из данного набора прологовских правил — это предоставляет известно резкое, но умышленное, ограничение. Правила верификации, леммы и индуктивно определённые абстрактные предикаты, всё записывается на Прологе. Модификации в структуре доказательства получаются задачами программирования и могут быть проверены на каждой фазе.
6. Упрощение спецификации и верификации ради ужесточения. Нашлось ужесточенное определение одночленной кучи, вместо множественного числа куч. Ужесточение заключается в атолизме куч. Благодаря этому, чётче, проще и короче можно специфицировать кучи. Принудительное сравнение куч со всеми остальными кучами отпадает, основные кучи сохраняются.

Сейчас кучи можно эффективно сравнивать. Благодаря групповым свойствам, теперь можно высчитывать, устанавливать в наборе правил не хватающую кучу и таким образом, можно проверять полноту. Далее, благодаря частичным спецификациям, сейчас можно уменьшать и редуцировать количество и объём специфицируемых правил с кучами — исключения исключаются, полнота доказывается с помощью неполноты. На практике ужесточение означает, что подвыражения кучи больше не должны полностью анализироваться.

Так как «*UML/OCL*» не содержит указателей, но представленные условия и критерии совместимы, то предлагается соответствующее расширение.

7. Возможность определения всё новых теорий над кучами. Выявленные алгебраические свойства позволяют определять равенства и неравенства между кучами. Эти формальные теории могут непосредственно задаваться в Пролог, либо соответствующим SAT-решателям (возможно, также на основе Пролога). Теперь правила и теории проверяются проще.
8. Абстрактные предикаты определяют атрибутируемую грамматику. Аналогично к проверкам схем при слабо структурированных данных, абстрактные предикаты являются шаблонами, которые содержат «дыры». Они наполняются во время верификации. Эта мета-концепция приводит к синтаксическому анализатору. В итоге логический вывод абстрактных предикатов можно интерпретировать как синтаксический анализ. Сравнение актуальной кучи с ожидаемой, можно расталкивать как проблему слов в формальных языках. Естественно, этот универсальный метод имеет ограничения касательно распознавателя, однако, подход очень практичен, несмотря на теоретический формализм, на котором основан процесс перевода.
9. Универсальный подход к автоматизации утверждений. Предикаты и доказательство производятся на основе Пролога и могут мульти-парадигмально включать любые другие библиотеки. Доказательство приближается к программированию. Необходимость повторно определять процесс перевода и описания правил отпадает потому, что Пролог всё это включает в себя, хотя сам язык минимален. Необходимость проводить свёртывание и развёртывание вручную отпадает. Если куча не выводима по данным правилам, то можно их переписать. Синтаксический анализ решим и терминация гарантирована, например, на основе LL-распознавателей за линейное время. Процесс генерации анализатора производится лишь после изменения правил. В зависимости от доступности начальных нетерминалов, распознавание подграмматик избегает необходимости принудительного построения всё новых анализаторов. На практике все нетерминалы могут теоретически быть использованы после построения анализатора. Синтаксические анализаторы не нуждаются в дальнейших исследованиях, т.к. они отлично изучены на сегодняшний день, следовательно, не требуется дальнейшее исследование анализатора, который используется в данной реализации.

В случае ошибки, автоматическая генерация и выдачи контр-примера, производится анализатором без дополнительных затрат.

«*Дальнейшее исследование*» В числе дальнейших исследовательских и практических работ можно выявить следующие (не отсортировано по важности):

1. *Интеграция решателя.* После определения теории куч, для ускоренного логического вывода необходимо подключение «*SAT*»-решателя. Решатель также может быть написан в самом Прологе.
2. *I Интеграция в существующие среды.* Помимо упомянутых ограничений, необходимо на практике преобразовать термовое IR в IR пактов «*GCC*», либо «*LLVM*» для индустриального применения.
3. *II Интеграция в существующие среды.* Точечный анализ мест «разрастания» (образов) может также привести к более выгодному выделению и расположению памяти, подключая аппроксимацию метод абстрактной интерпретации. Также необходимо рассмотреть возможность выражений с статически выявляемыми офсетами ради широкой практической применимости.
4. *Изоморфизм куч.* Проверка изоморфизма графов в случае возникновения вопроса: может ли куча в принципе быть представлена данными указателями, при этом наименования кучи не уточняются?
5. *Нахождения минимального отклонения от спецификации.* При синтаксическом переборе может иметь смысл найти глобальный минимум прописной дистанции для проверки и возможной редакции спецификации.
6. *Анализ зависимостей указателей.* В отличие от анализа псевдонимов, рекомендуется исследовать расширение SSA-формы для динамических переменных, что до этого ещё никем не рассматривалось. Ожидается расхождение, т.к. понятие о зависимости различается. Возможность подключения может также подтвердить более безопасный код при оптимизации, т.к. на первый взгляд несвязанные ячейки связываются, либо точно не связываются. Это также способствует эффективному коду.
7. *Верификация кодов баз.* В этой работе после устранения технических ограничений в связи с быстрой прототипизацией, предлагается проводить анализ имеющихся кодов баз. Для этого предлагается использовать в открытом доступе пользования, потому, что они также значительно различаются.
8. *Исследование применимости языков трансформаций на практические нужды.* Обсуждённые языки трансформации могут иметь компактную запись и также часто записываются с помощью правил состояний до и после трансформации, однако ожидается большое расстояние

между используемой моделью кучи, моделью выразимости и содержанием динамических куч, входным языком.

9. *Абстракция вывода.* Некоторые диалекты Пролога поддерживают абдуктивный вывод. В общем, абстракция механизма, который позволяет правила Пролога упростить, сравнить с подходящими пред- и постусловиями и выбрать наиболее подходящее правило. Порядок вывода можно упростить, если соотношения между термами обратимы. Это как раз было предложено и обсуждено в этой работе.

**«Рекомендации для применения»** В числе рекомендаций по итогам работы можно вывести следующие:

- Практически спецификация куч стала (немного) проще, если даже не больше. Данная программа аннотируется спецификацией.
- Абстрактные предикаты должны быть синтаксически разборчивыми, тогда подцели соответствующей формальной грамматики можно распознать. В зависимости от мощи анализатора часто переписывание правил приводит к разборчивости.
- Использование ужесточенных операторов даёт возможность в спецификациях «*UML/OCL*» пространственность объектных экземпляров выражать.
- Введение всё новых языков программирования не обязательное. Язык программирования может даже полностью отсутствовать. Расширяемое и модифицируемое промежуточное представление позволяет максимальную гибкость. Архитектура верификации имеет те же самые свойства.
- Сравнение куч с данными выражениями в правилах верификации способствует выявить не специфицированные кучи, а более широко способствует к решению вопроса полноты.
- Перегрузка значений реляций упрощает гибкость логического вывода, например, в связи с абдукцией, но также может быть использована различными способами.
- Использование SAT-решателя для задаваемых теорий куч, будь-то в Прологе или мультипарадигмально, может привести к резкому ускорению преобразования куч в нормализованную и упрощённую форму.
- Использование мемоизатора (в некоторых диалектах Пролога «*tabling*») для абстрактных предикатов может привести к резкому ускорению логического вывода, в частности при пошаговой верификации.
- При введении и модификации фаз работы с динамической памятью рекомендуется придерживаться к термовому представлению.

## Список сокращений

<b>АО</b>	Анализ Образов
<b>АТ</b>	Автоматизация Теста
<b>АТД</b>	Абстрактный Тип Данных
<b>ВО</b>	Вычисление Объектов
<b>ВР</b>	Вычисление Регионов
<b>ЛРП</b>	Логика Распределённой Памяти
<b>ООМ</b>	Объектно-Ориентированное Моделирование
<b>ОС</b>	Операционная Система
<b>ПО</b>	Программное Обеспечение
<b>РФБН</b>	Расширенная Форма синтаксиса по Бэккусу-Наура
<b>ТО</b>	Теория Объектов
<b>ЧУМ</b>	Частично-Упорядоченное Множество
<b>ABI</b>	Application Binary Interface
<b>BSS</b>	Block Started by Symbol
<b>DCG</b>	Definite clause grammar
<b>DEC</b>	Digital Equipment Corporation
<b>GCC</b>	GNU Compiler Compiler
<b>GIMPLE</b>	a GNU tree representation of programming units
<b>IR</b>	Intermediate Representation
<b>LCF</b>	Logic of Computable Functions (Scott)
<b>LLVM</b>	Low-level virtual machine
<b>OCL</b>	Object Constraint Language
<b>PCF</b>	Programming Computable Functions (Plotkin)
<b>SAT</b>	SATisfaction of a logical formula
<b>SMT</b>	Satisfactority Modulo Theory
<b>SSA</b>	Single Static Assignment
<b>UML</b>	Unified Modeling Language
<b>WAM</b>	Warren's Abstract Machine
<b>poset</b>	partially-ordered set

# Список терминов

**SSA-форма** ↑ *Промежуточное представление* о зависимостях данных, в котором имеется строго одно определение и несколько мест использования присвоенного значения. Если переменная (возможно) переписывается, то присваивается следующее значение.

**Абдукция** Принцип доказательства, ↑ *мета-схема*, основана на логическом выводе, который основан на ↑ *фактах* и на следствиях импликаций.

**Абсорбция** Для ↑ *утверждений*  $p$  и  $q$  в булевой логике в силе:  $p \wedge (p \vee q) \equiv p \vee (p \wedge q) \equiv p$ .

**Абстрактная машина Уоррена (WAM)** В отличие от традиционных операционных семантик и полученные от них абстрактные машины по работе со стековыми архитектурами. WAM обогащает стековые окна различными ссылками на предыдущие и последовательные окна. Ссылки меняют стековое окно и зависимые от них до и после вызова некоторой процедуры, это может также поощрять комплексное обновление зависимых смежных термов ниже и выше по иерархии вызова.

**Абстрактный предикат** Предикат над кучами использующий параметры, может быть сложным или атомной кучей.

**Абстрактный тип данных** Предшественник стоящий за концепцией «↑ *класс*». Устанавливается поведение с внешним миром и другими актерами. Определяет интерфейсы, ↑ *методы*. Хранителями состояния объекта выступают ↑ *поля*.

**Абстракция доказательства** Планировка достижимого доказательства высчитывает промежуточные пути к успешному доказательству.

**Абстракция функций** Как частный случай лямбда-абстракции, абстракция функций являются модуляризацией процедур.

**Автомат, конечный** Направленный простой граф, чьи вершины являются ↑ *состоянием вычисления*, а грани являются переходами между состояниями. Граф должен иметь одно начальное состояние и любое число конечных состояний. Переходы определяются правилами, которые должны определять регулярную (праворекурсивную) грамматику, т.е. имеют форму:  $A \rightarrow aB$ , где  $a$  является терминалом, а  $A, B$  являются нетерминалами.

**Автоматизированное доказательство теорем** Доказательство  $\uparrow$ теорем с помощью некоторых механизмов, которые позволяют успешно завершить доказательство полностью, в отличие от  $\uparrow$ частично-автоматизированных доказательств, либо  $\uparrow$ вручное доказательство, при которых ожидаются интервенции со стороны пользователя.

**Адресное пространство динамической памяти** Расположенный сегмент в операционной памяти, который адресуется линейно и не содержит дырок, выделены при запуске процесса операционной системы, не структурированная часть памяти, где выделение памяти осуществляется пользователем, в отличие от  $\uparrow$ стека.

**Аксиомы Пиано** Имеются операции «+», « $\cdot$ » и  $\uparrow$ упорядоченное (в общем случае некоторое обобщённое и эквивалентное к) множество(-у) натуральных чисел, на которых устанавливаются хорошо известные аксиомы сложения. Например, равенство определяется как равенство первого числа множества и (индуктивное) равенство последующих чисел (ср. также  $\uparrow$ арифметику термов по Чёрчу).

**Алгебраическое поле** Дискретная структура с двумя операциями (специальное кольцо), для которой соблюдаются например, законы как для «+» и « $\cdot$ », имеются два различающихся нейтральные и обратимые элемента. Если поле конечное, то поле называется полем Галуа.

**Алгоритм Левенштейна** Базисный алгоритм для вычисления минимальной прописной дистанции, сложность которого ограничивается кубическим полиномом.

**Анализ образов** Памятная модель, в которой описываются  $\uparrow$ кучи, как геометрические фигуры.

**Анализ зависимости данных** Выделение всех зависимостей переменных для данной программы. Анализ зависимостей необходим например, для построения SSA-формы.

**Анализ псевдонимов, внешний** Вычисление  $\uparrow$ псевдонимов параметров и  $\uparrow$ глобальных переменных, которые могут меняться между вызовами процедур. Внешний анализ является более трудным, чем  $\uparrow$ внутренний, сложность в общем случае оценивается как экспоненциальная.

**Анализ псевдонимов, внутренний** Вычисление  $\uparrow$ псевдонимов внутри одной процедуры. Внутренний анализ практически можно считать решимым.

**Аналогия доказательств** При доказательстве  $\uparrow$ теорем  $\uparrow$ мета-схема, которая сравнивает уже сделанное доказательство и пробует с помощью  $\uparrow$ абстракции применить доказательство для решения данной проблемы, которая похожа.

**Антецедент** В правиле: «Если  $A$ , то  $B$ »,  $A$  является антецедентом.

**Арифметика термов по Чёрчу** Арифметика натуральных чисел, где множество носителя заменяется множеством термов, которые сопоставляют натуральные числа следующим образом:

$0 \mapsto z$ ,  $1 \mapsto s(z)$ ,  $2 \mapsto s(s(z))$ , и т.д. Таким образом,  $m + n$  может быть определен как:  $s^m + s^n = s^{m+n}$ , аналогично минус с помощью унификации и сопоставления с образцами. Преимуществом является  $\uparrow$ *обратимость* операции над термами.

**Атомизм** Аналитический метод, основан на делении сложной проблемы на маленькие неделимые единицы, атомы. Широкое применение используется в самых различных научных дисциплинах.

**Атрибутируемая грамматика**  $\uparrow$ *Формальная грамматика* расширена атрибутами, которые могут быть, либо  $\uparrow$ *наследственными*, либо  $\uparrow$ *синтезированными*.

**Безопасные операции указателей** Используются в контексте « $\uparrow$ *ротации указателей*» для обозначения тех ротаций, которые известны и использование чьё не приводит к непредвидимым ошибкам, либо к другим побочным ошибкам.

**Безусловная ошибка** Встроенный  $\uparrow$ *предикат fail* в Прологе приводит к безусловной неудаче поиска и  $\uparrow$ *отсечения* пространства поиска до вызова предиката.

**Быстрая прототипизация** Разработка прототипа для демонстрации одного и более примеров без дорогостоящего цикла разработки. Часто страдает от ограничений и ошибок.

**Висячие указатели**  $\uparrow$ *Указатели* на не корректные ячейки в памяти, которые, до недавних пор ещё являлись корректными.

**Выделение памяти на стеке** На  $\uparrow$ *стеке* выделенная  $\uparrow$ *переменная автоматически* освобождается после выхода из процедуры, в отличие от иных  $\uparrow$ *модусов переменных*.

**Выравнивание адресов офсетов в полях записей** Перемещение офсетов полей записей и  $\uparrow$ *объектов* внутри ячейки  $\uparrow$ *динамической памяти* может привести к минимизации объёма употребляемой памяти. Обязательным условием остаётся: ячейка распределяется в непрерывном регионе памяти.

**Вычисление Хиндлея и Миллнера** Вычисление типовых систем, где главным правилом типизации является: если имеется любая переменная типа  $a$ , которая применяется к некоторому  $\uparrow$ *функционалу* типа  $a \rightarrow b$ , то результат будет иметь тип  $b$ .

**Вычисление Хора** Формальный метод доказательства свойств программ при использовании математико-логических формул. Состояние до и после загрузки команды данного программного оператора некоторого императивного языка программирования, используется для сравнения ожидаемого, с полученным  $\uparrow$ *состоянием* вычисления. Ожидаемые условия, это  $\uparrow$ *пред-* и  $\uparrow$ *постусловия*.



**Вычисление регионов** Памятная модель, в которой элементы оперативной памяти приписываются «региону» для более удобного управлению памяти.

**Вычитаемые правила** Правила, которые характеризуются тем, что оставшаяся часть доказуемой  $\uparrow$ теоремы после применения правила становится меньше (ср.  $\uparrow$ структурные правила).

**Гомоморфизм** Если имеется одна функция (или операция)  $g$ , то гомоморфизм имеется в отношении другой функции  $h$ , если в силе:  

$$h(g(x_0, x_1, \dots, x_n)) \equiv g(h(x_0), h(x_1), \dots, h(x_n)).$$

**Граф кучи** Графовое представление  $\uparrow$ кучи, где грани представляют зависимости между кучами, а вершина представляет пары  $loc \mapsto val$ .

**Граф кучи, определяемый по граням**  $\uparrow$ Граф кучи, чьи грани создают список с двумя столбцами: начало и конец.

**Группа (алгебра)**  $\uparrow$ Моноид, для которого соблюдается обратимость.

**Дедукция** Принцип доказательства,  $\uparrow$ мета-схема, основана на логическом выводе имеющегося  $\uparrow$ факта и подходящие к нему предпосылки импликаций.

**Декларативный язык программирования** Например: логический и функциональные языки программирования. В отличие от императивного языка, декларативный язык лишь описывает, что должно вычисляться.

**Деление ответственности** Распределение ролей и интерфейсов по объектам в зависимости от ответственности.

**Дерево доказательства** Структура доказательства является деревом, вершины чьи представляют  $\uparrow$ состояние верификации. Грани дерева подписываются номером применяемого правила. Листьями дерева являются аксиомы.

**Диапазон видимости** Отрывок программы, с которой  $\uparrow$ стековая переменная вталкивается в  $\uparrow$ стек и после которой она выталкивается обратно.

**Динамическая память** Вместе со  $\uparrow$ стеком и другими областями, один из сегментов процесса расположенный в операционной памяти. Неорганизованная часть памяти, которая содержит все динамически выделенные  $\uparrow$ кучи.

**Доминатор, вершина графа** Вершина в направленном графе относительно другой вершины, если не существует альтернативный путь между двумя вершинами, кроме как через вершину доминатора.

**Естественная дедукция** Логическое доказательство, основанное на предположениях и последовательных доказательствах, либо отрицаниях. Структура естественной дедукции является основным принципом  $\uparrow$ *вычислением Хора*, в отличие например, от метода резолюции (по Робинзону).

**Жизненный цикл объекта** Период времени с момента создания объекта до момента его уничтожения.

**Запрос (в Прологе)** Правила и  $\uparrow$ *факты* Пролога представляют базу знаний, которой можно задавать запросы в качестве  $\uparrow$ *подцелей*, которые перечисляются запятой. Количество ответов ноль или более того, в зависимости от того, сколько ответов интерпретатор Пролога выявит согласно выбранной стратегии поиска.

**Изначальное определение** Значение присвоенное переменными. Проблема неприсвоенного значения может приводить к ряду проблем, всегда тогда, когда читается значение переменной без инициализации.  $\uparrow$ *Динамическая память* и *bss* обычно не инициализируются.

**Изоморфизм Карри-Хауарда** Постулат, по которому доказательства и программирование связаны между собой.

**Инвариант Цикла** Формула условия, которая не меняется при загрузке цикла и только при выходе из цикла не верное. Часто инварианты цикла трудно выявить, потому, что полный и чёткий инвариант содержит все  $\uparrow$ *переменные* из тела цикла и не содержит лишних случаев. Полный и точный инвариант автоматически трудно угадывать и часто является проблемой контринтуитивной.

**Инверсия кучи**  $\uparrow$ *Слияние кучи* вместе с инверсией той самой  $\uparrow$ *кучи* определяется как  $\uparrow$ *пустая куча*.

**Инверсия над контролем** Контроль над вызовами от вызывающей стороны передаётся к вызванной стороне. Инверсия необходима для защиты и является важным инструментом для реализации  $\uparrow$ *фреймворков*.

**Индуктивно определённая структура** Например: линейный список, тройное дерево,  $\uparrow$ *куча*, и т.д.

**Индукция** Принцип доказательства,  $\uparrow$ *мета-схема*, основана на общепринятом, не противоречивом  $\uparrow$ *утверждении*. Доказательство индуктивно-определённых структур данных, часто доказывалось простым способом.

**Интерпретация кучи** Сравнение данной  $\uparrow$ *кучи* согласно данной спецификации.

**Интроспекция кода** Модификация и чтение имеющихся и желаемых  $\uparrow$ *полей* и  $\uparrow$ *методов* во время запуска программы. Интроспекция может менять загружаемый код, либо загруженный код

писания в отдаленные регионы памяти, через сеть, либо некоторый файл, который несколько процессов делят между собой.

**Интуиционистское суждение** Суждение, при котором используются только  $\uparrow$ факты и правила формой «если  $A$ , то  $B$ » (известно как «*modus ponens*»).

**Класс** Записной тип объединивший множество классных  $\uparrow$ полей и им ассоциированных  $\uparrow$ методов с различными наименованиями.

**Клауза** Нормализованная форма, в которой все  $\uparrow$ литералы связаны между собой в дизъюнктивной нормализованной форме.

**Комбинатор плавающей точки** Синтаксически искусственный оператор, который предлагается, не является элементом языка программирования. Для симуляции рекурсии. Комбинаторы являются элементами лямбда-вычислений, как основы синтаксиса и семантики языков программирования.

**Консеквент** В правиле: «Если  $A$ , то  $B$ »,  $B$  является консеквентом.

**Контр-пример** Пример, который показывает, почему некоторое данное  $\uparrow$ утверждение не является универсально правильным.

**Контр-примера, генерация** Когда универсальность не может быть доказана, выявляется хотя бы один  $\uparrow$ контр-пример. Унификация термов в Прологе сравнивает совпадающие и несовпадающие части, которые могут быть использованы для генерации контр-примера.

**Конфигурация кучи** Некоторый конкретный связанный  $\uparrow$ граф кучи.

**Куча** Расположенная, связанная структура данных в  $\uparrow$ динамической памяти, которая имеет  $\uparrow$ указатель-/и в  $\uparrow$ стеке, либо в  $\uparrow$ куче.

**Лемма доказательства** Как  $\uparrow$ теорема, но без отдельной отличающейся значимости единицы теоремы. Лемма может быть использована для доказательства теоремы различными параметрами и может быть применена, либо  $\uparrow$ развёрнута, либо  $\uparrow$ свёрнута.

**Ленивое вычисление** Вычисление выражений происходит при необходимости снаружи во внутрь. Параметры передаются как не интерпретированные выражения, и вычисляются только при конкретной необходимости, как например, требуется актуальное число для сохранения значения. Без необходимости, выражение используется как символьное.

**Литерал** Атомное  $\uparrow$ утверждение, либо его отрицание.

**Логика Распределённой Памяти** Памятная модель, в которой описываются  $\uparrow$ кучи как ансамбль ссылок  $a \mapsto b$ .

**Логический оператор кучи** Логическая импликация, «и», «или», отрицание.

**Локализатор** ↑*Локализатор памяти*.

**Локализатор памяти** Памятный адрес, который может храниться, либо в ↑*стеке*, либо в ↑*куче*.

**Локализация ошибочного кода** Процесс поиска некоторого ↑*симптома ошибки* до источника возникновения ошибки, который часто находится в очень маленькой части данной программы. Запуск программы ручным поиском симптома называется отладка программы.

**Локальность (принцип о кучах)** Принцип, по которому локальные изменения ↑*кучи* не должны влиять на весь ↑*граф кучи*. Локальность может резко упрощать верификацию куч.

**Мемоизация** Кэширование вызовов ↑*предикатов* и функций.

**Мета-схема** В отличие от ↑*свёртывания*/↑*развёртывания* универсальная схема в доказательствах, как например, ↑*аналогия*, ↑*индукция*, ↑*абдукция*.

**Метод Кусо (Абстрактная интерпретация)** ↑*Статический метод анализа* приближённого вычисления сжатых и расширенных интервалов значения.

**Метод Резолюции (по Робинзону)** В отличие от ↑*естественной дедукции*, резолюция основана на попытке доказать противоречивость данной формулы, которая должна задаваться изначально в конъюнктивной нормальной форме. После отрицания получается дизъюнктивная нормальная форма и каждая из дизъюнктов доказывается. Пролог ищет выводимые решения, которые соответствуют конъюнктам, основываясь на ↑*факты*. Если все конъюнкты (↑*подцели*) верны, то верна и голова данного ↑*предиката* с исключительно замкнутыми термами.

**Метод класса** Процедура, приписанная к данному определению ↑*класса*, которая использует ↑*поля*. Все не ↑*локальные*, далее не специфицируемые переменные, являются ссылками на ↑*поля объектного экземпляра*, либо являются переменными ординарного типа. В отличие от полей, в классических видах ↑*объектного вычисления*, методы не меняются во время запуска.

**Метод логических таблиц (по Бету)** Вывод производится только, если логический вывод прописан в данной таблице.

**Моноид** Дискретная структура с одной ↑*тотальной* операцией. Любой моноид является полугруппой с нейтральным элементом (т.е. замкнутость, ассоциативность и нейтральность соблюдаются).

**Мусор (в динамической памяти)** ↑*Куча*, которая не имеет ↑*указателей*, в частности, куча, которая не связана.

**Наследование классов** Наследование  $\uparrow$ полей и  $\uparrow$ методов классов из одной генерации в следующую, согласно данному  $\uparrow$ уровню видимости.

**Наследственный (порождаемый) атрибут** Атрибут в  $\uparrow$ атрибутируемой грамматике, который по иерархии зависимостей передаётся от вызванного  $\uparrow$ предиката к вызывающему предикату.

**Неверный доступ к памяти** Доступ по неверному адресу может прочесть неверные данные, испортить иные ячейки памяти, либо скомпрометировать вычислительный процесс.

**Неопределённая спецификация** Спецификация, которая содержит сопоставляющие символы образцами.

**Непересекаемые кучи** Не существует вершина  $\uparrow$ графа кучи, которая относится к двум  $\uparrow$ кучам одновременно.

**Неповторимость (принцип о кучах)** Принцип, по которому утверждения об одном и том же  $\uparrow$ графе кучи, не повторяются. Неповторимость может резко упрощать верификацию  $\uparrow$ куч.

**Неполная куча** Не полностью определённая куча, содержащая не присвоенные символы. Неполная куча в спецификации может быть сопоставлена с образцами.

**Непосредственный доминатор (Immediate Dominator)** Строгий  $\uparrow$ доминатор, который не может быть одной из данных доминаторов.

**Обобщённая куча**  $\uparrow$ Простая куча, либо композиция из простых или сложных  $\uparrow$ куч, возможно, держащие  $\uparrow$ абстрактные предикаты.

**Обратимость предиката** Когда вызов  $\uparrow$ прологового предиката изменен так, что входящие и выходящие термы могут поменять свои места, то речь идёт об обратимости предиката. Примером является встроенный в Прологе предикат `append/3`.

**Объектная спецификация** Включает в себя  $\uparrow$ объектный инвариант,  $\uparrow$ пред- и  $\uparrow$ постусловия всех методов и все внутренние спецификации методов.

**Объектное вычисление** (1-ый) классический вид по Абади-Лейно при использовании  $\uparrow$ классов, как это принято в языках Ява или Си++, либо (2-ой) объектный вид вычисления по Абади-Карделли при конструкции  $\uparrow$ объектного экземпляра без классов, как это принято в Baby-Modula 3 и иных разработанных прототипных языках Карделли.

**Объектный инвариант** Инвариантное  $\uparrow$ утверждение (ср.  $\uparrow$ инвариант цикла), которое всегда должно соблюдаться до и после любых вызовов  $\uparrow$ методов и модификации  $\uparrow$ полей  $\uparrow$ объектного экземпляра.

**Объектный тип**  $\uparrow$ Класс.

**Объектный экземпляр** Переменная ссылающаяся на некоторый выделенный, неделимый и непрерывающийся регион в операционной памяти данного  $\uparrow$ класса или всех его подклассов.

**Опровержение доказательства** Результат доказательства  $\uparrow$ теоремы, может сопровождаться  $\uparrow$ контр-примером для иллюстрации частного случая, который противоречит выполнению данной  $\uparrow$ теоремы или ее части.

**Отмотка стека** Восстановление состояния  $\uparrow$ стека, которое может происходить после возникновения исключения и оно было в действии до исполнения критичной последовательности программных операторов.

**Отсечение прологовских решений** Отсечение всех имеющихся альтернативов решений унификаций  $\uparrow$ подцелей внутри одного тела  $\uparrow$ предиката с помощью встроенного оператора «!». Когда отрезаются ненужные дубликаты, либо последовательные альтернативы, которые не вычисляют необходимые решения данного алгоритма, то отсечение называется «зелёным», когда отрезаются годные решения, то оно «красное».

**Памятная модель по Бурстоллу**  $\uparrow$ Простые кучи в ЛРП выглядят так: локация  $\mapsto$  адрес.

**Памятная модель по Рейнольдсу**  $\uparrow$ Простые кучи в ЛРП выглядят так: локация  $\mapsto$  значение.

**Парадигма (программирования)** Например императивная или  $\uparrow$ декларативная, либо мульти-парадигмальная [81].

**Переменная, автоматически выделенная** Выделение и уничтожение производятся при входе и при выходе из процедур автоматически на  $\uparrow$ стеке.

**Переменная, глобальная** Глобальная переменная может быть перекрыта во внутренних блоках  $\uparrow$ локальными переменными с одинаковым именем. Глобальные переменные выделяются лишь один раз в сегменте .bss операционной системы вместе с неинициализированными данными, однако, инициализация и уничтожение могут быть совершены произвольно в любых местах кода во время запуска процесса.

**Переменная, динамически выделенная** Выделение и уничтожение производятся явной командой до тех пор, пока процесс существует.

**Переменная, локальная**  $\uparrow$ Переменная, автоматически выделенная.

**Переменная, модус** Модус зависит от жизненного цикла переменной и области видимости. Модусы могут быть  $\uparrow$ автоматическими,  $\uparrow$ статическими,  $\uparrow$ динамическими или  $\uparrow$ глобальными и временно хранящими в регистрах процессора, и т.д.

**Переменная, статическая** Выделение происходит один раз за весь процесс приложения и уничтожается лишь при утилизации процессом операционной системы.

**Переменная, стека** ↑*Переменная, автоматически выделенная.*

**Пересечение ячеек памяти** Когда ячейка памяти полностью или ↑*частично* интерпретируется по-разному, например, разными ↑*указателями*, либо различными типами и разрядностями (например, с помощью оператора `union` в языках Си).

**Подтип(-изация)** Подтипом класса является любой наследованный ↑*класс* согласно иерархии классов. Словесная (под-)типизация преобразует данный объект соответственного класса в новый ↑*объект* с одинаковым начальным адресом, но с соответственным наследованным классом.

**Подцель** При ↑*запросе в Прологе* обрабатывающая часть доказательства.

**Поиск доказательства** Поиск возможного решения доказательства данной ↑*теоремы*, ограничивается ↑*мета-схемами* и ↑*тактиками* поискового пространства.

**Поле классного экземпляра** Памятная ячейка данного типа или ↑*класса*, которая принадлежит соответствующему объекту.

**Поле объектного экземпляра** Памятная ячейка данного типа или ↑*класса*, которая принадлежит соответствующему объекту и может быть добавлена, изменена или удалена во время запуска программы.

**Полиморфизм** Разновидность процедур, например, ↑*классов*, а в общем переменных различных типов. Вызов ↑*объектного* экземпляра `m1` может вызвать, либо `m1` объявленного класса, либо `m1` подкласса во время запуска программы. По Абади это ↑*полиморфизм по классу/типу*. Кроме того, в некоторых ↑*декларативных языках программирования* ещё имеет *настоящий полиморфизм*, который позволит вызвать одну и ту же функцию с аргументами различных типов.

**Полиморфизм классов** Принадлежность ↑*класса* данного ↑*объектного экземпляра* может меняться во время запуска, следовательно, вызов метода может быть вызовом некоторого метода подкласса.

**Полу-автоматизированное доказательство теорем** Доказательство совершается, либо зависит от ввода дополнительных лемм или преобразований равенств при процессе доказательства, если доказательство существует в принципе.

**Порядок вычисления** Порядок вычисления выражений, как например, слева-направо, снаружи во внутрь, ↑*ленивое вычисление*, и т.д.

**Постусловие** Третья компонента  $Q \uparrow$  *тройки Хора*.

**Поток токенов** Последовательность  $\uparrow$  *токенов* полученные после лексического анализа.

**Правила синтаксических ошибок** При синтаксическом анализе грамматическая ошибка может влечь за собой исправление синтаксической ошибки для успешного дальнейшего синтаксического анализа.

**Правила, прологовские**  $\uparrow$  *Предикат, прологовский*.

**Предикат, прологовский** Близок к предикатам в предикатной логике, имеющие  $\uparrow$  *входящие и/или выходящие термы* в качестве параметров. Предикат имеет голову и тело, которое может иметь ноль или более  $\uparrow$  *подцелей*. Когда имеется ровно ноль подцелей, то предикат является  $\uparrow$  *фактом*.

**Предикатная логика (первого порядка)** Логика  $\uparrow$  *утверждений* плюс предикат и квантифицируемые символы.

**Предусловие** Первая компонента  $P \uparrow$  *тройки Хора*.

**Примитивная рекурсия** Рекурсия с числом итераций, которое известно до начала итерации.

**Принцип скрывания данных** Гласит по определению о том, что во избежание конфликтов,  $\uparrow$  *абстрактный тип данных* нужно так ограничить  $\uparrow$  *уровень видимости*, чтобы базовая коммуникация была обеспечена исключительно по ранее уговоренным  $\uparrow$  *методам*.

**Проблема корреспонденции Поста** Имеется два набора правил с помощью чьей производятся два слова. Вопрос в том, производят ли два набора правил тот самый  $\uparrow$  *формальный язык* или нет, доказано, что вопрос нерешим.

**Проблемы с динамической памятью** Такие проблемы, как например,  $\uparrow$  *висячие указатели*,  $\uparrow$  *утечка памяти*,  $\uparrow$  *неверный доступ к памяти*, и т.д.

**Проверка моделей** Модель описывается формальной системой равенств и неравенств данной ( $\uparrow$  *формальной*) теории.  $\uparrow$  *Состояние вычисления* описывается формулой и может проверяться  $\uparrow$  *решателем*.

**Проверка типов** Проверка, применяются ли в данной программе все выражения в соответствии с типами декларируемых переменных.

**Программирование через доказательство** Философия, которая внушает, что благодаря доказательству, можно производить задачу программирования.

**Промежуточное представление (программы)**  $\uparrow$  *Термы*, триады, тетрады, и т.д. представляют входную программу и возможные далее аннотации в более удобном виде для дальнейшей обработки.



**Простая куча** Куча, которая состоит из одного соотношения  $a \mapsto b$ , в отличие от  $\uparrow$ обобщённой кучи или  $\uparrow$ абстрактного предиката.

**Пространственный оператор куч** Перечисляет, но не соединяет, две  $\uparrow$ кучи. Связь между кучами допускается с помощью двоичного пространственного оператора с помощью употребления символов в разных кучах на обоих сторонах ссылок  $a \mapsto b$ .

**Псевдоним**  $\uparrow$ Указатель, который ссылается на одну и ту же структуру данных, как и второй указатель.

**Развёртывание предиката** Операция развёртывания сопоставляет данное определение некоторой используемой  $\uparrow$ леммы более подробным определением  $\uparrow$ леммы.

**Разделение куч** Две  $\uparrow$ кучи разделяются на две независимые кучи.

**Расхождение формальных языков** Языки различного назначения могут расходиться по синтаксическим, семантическим и по интуитивному обозначению. Знаки коннотации могут иметь различные и многозначные обозначения, что приводит к целому ряду проблем сравнимости, например, выразимости.

**Рекурсивность термов, проверка на** В частном случае, Пролог по определению не проверяет, является ли, например,  $X = s(1, s(2, X))$  унифицируемым термом.

**Рекурсивный тип (обобщённый зависимый тип)** Зависимый, в частности рекурсивный тип, является сложным типом, например записью, которая определяется другими типами.

**Реляционная алгебра** Отношения реляционных функций, можно интерпретировать как  $\uparrow$ предикаты, в частности представить как  $\uparrow$ прологовские предикаты. В реляционной алгебре имеются операции объединения, пересечения, вычитания, деления и Декартовское произведение.

**Решатель** Автоматическое решение формулы равенств и неравенств заданной  $\uparrow$ формальной теории.

**Ротация указателей** Операции над указателями, как например, поменять указатели местами по часовой, или иных сложных команд, которые простые, но совершают довольно сложные ротации, которые могут описывать ротации деревьев, для установления баланса.

**Самообратимая операция** Операция, которую если применить дважды к данному аргументу, возвращает изначальный аргумент. Часто  $\uparrow$ тотальность способствует доказательству обобщённого свойства дискретной структуры, как например,  $\uparrow$ группа.

**Самосодержащий терм**  $\uparrow$ Рекурсивность термов.

**Сбор мусора** ↑*Указатели* на ↑*содержимое*, которые без пользы. Примерами служат ↑*сбор мусора по поколениям*, ↑*сбор мусора по картам*, и т.д.

**Сбор мусора по поколениям** ↑*Сбор мусора по поколениям*, когда ↑*указатели* со старым числом, но без действия, проверяются и при необходимости удаляются первыми из ↑*динамической памяти*.

**Сбор мусора по картам** Предположительно, параллельными нитями по принципу Z-буферизации, регионы ↑*динамической памяти* выкладываются на карту, которые нуждаются чаще всего в сборе мусора, затем проводится минимизированный по затратам сбор близких независимых регионов.

**Свёртывание предиката** Операция свёртывания сопоставляет данные ↑*утверждения* сокращённым представлением определения данной ↑*леммы* в целях простой читаемости и для совершенства доказательства с наименьшим количеством шагов.

**Семантическая функция** Часто неявно-определённая функция, где входные и выходные семантические домены являются хорошо известными семантическими множествами значений.

**Сигнатура (метода)** Состоит из наименований ↑*метода*, принадлежащего ↑*классу*, типы и входных параметров, типы и выходных параметров.

**Сильное Постусловие** Ужесточённое ↑*утверждение*  $Q'$ , которое может быть использовано при логическом выводе, вместо более широкого постусловия  $Q$ , т.е.  $Q'$  является сужением  $Q$ :  $Q \Rightarrow Q'$ .

**Симптом ошибки** Наблюдаемое поведение программного обеспечения, которое не совпадает с ожидаемым, возможно ошибка. Симптом соответствует одному или более мест в программном коде. Симптом должен быть эмпирически выводим.

**Синтаксический анализатор Эрли** Основан на вталкивании и интерпретации контекст-свободных правил в обрабатывающий ↑*стек*.

**Синтаксический анализатор сверху-вниз и слева-направо** ↑*Поток токенов* обрабатывается слева-направо, правила ↑*формальной грамматики* обрабатываются сверху-вниз.

**Синтаксический анализатор снизу-вверх и слева-направо** ↑*Поток токенов* обрабатывается слева-направо, правила ↑*формальной грамматики* обрабатываются снизу-вверх.

**Синтезированный атрибут** Атрибут в ↑*атрибутируемой грамматике*, который по иерархии зависимостей передаётся от вызывающего ↑*предиката* к вызванному предикату.

**Система переписывания термов** Правила применяются согласно данной стратегии согласно сопоставлению с образцами.

**Слабое Предусловие** Ослабленное  $\uparrow$ утверждение  $P'$ , которое может быть использовано при логическом выводе вместо более узкого предусловия  $P$ , т.е.  $P'$  является обобщением  $P$ :  $P' \Rightarrow P$ .

**Слияние куч** Соединение двух  $\uparrow$ куч к одной связанной куче с помощью одного или более объединяющих вершин.

**Со-процедура** Передача  $\uparrow$ указателя при вызове процедуры, которая может загружаться одновременно от главной процедуры и осуществить параллелизм. Со-процедура может быть использована для реализации  $\uparrow$ функционалов.

**Содержимое указателя**  $\uparrow$ Указатель ссылается на ячейку памяти, чьё содержимое высчитывается.

**Состояние вычисления** Описание  $\uparrow$ стека и  $\uparrow$ кучи, т.е. перечень всех  $\uparrow$ переменных, которые на данный момент вычисления выделены и которое  $\uparrow$ содержимое к ним приписывается.

**Ссылочная модель** Опираясь на ссылочную модель,  $\uparrow$ утверждение гласит о том, какие  $\uparrow$ указатели имеются и какая структура содержится в  $\uparrow$ динамической памяти.

**Статический анализ** Анализ проводимый без запуска программы.

**Стек** Внутри процесса операционной памяти структурированный сегмент, который выделяется операционной системой, хранит  $\uparrow$ локальные переменные при вызове процедур в  $\uparrow$ стековом окне, которое освобождается при выходе.

**Стек Трибера**  $\uparrow$ Стек, который доступен некоторым нитям одновременно.

**Структурные правила** В отличие от  $\uparrow$ вычитаемых правил, являются правилами, которые определяются программными операторами, т.е. второй компонентой  $\uparrow$ троек Хора.

**Сходимость доказательств** Если доказательство разлагается согласно корректным правилам на несколько вариантов и все варианты применимы к допустимым правилам и в итоге всё это приводит к одному и тому же результату, то доказательство сходимо. В этом случае правила сходимы.

**Тактика** Применение, которое позволит решить доказательство, либо  $\uparrow$ частично, либо полностью.

**Теорема**  $\uparrow$ Утверждение, из области математики, которое представляет некоторую значимость в своей области, доказуемо с помощью ранее согласованного набора аксиом.

**Терм, входящий** Прологовский терм, который используется только как входящий. Терм частично или полностью определен в голове прологовского правила и может быть использован при сопоставлении с образцом.

**Терм, входящий и выходящий** Прологовский терм, который используется внутри тела правила, подтерм присваивается или унифицируется. При этом  $\uparrow$ *рекурсивные термы* в Прологе не запрещаются по определению и могут послужить примером входящего терма, который одновременно является выходящим.

**Терм, выходящий** Прологовский терм, который используется только как выходящий. Вызов  $\uparrow$ *подцели* с не унифицируемым термом должен приводить к ошибке унификации.

**Токен** Полученная атомная единица после лексического анализа, представляющий самую маленькую единицу некоторого языка программирования, например, символ присваивания «:=».

**Тотальность** Функция определена полностью для любого элемента из домена.

**Тройка Хора** Содержит следующие компоненты: (1)  $\uparrow$ *состояние вычисления*  $P$  до выполнения данного оператора, (2) программный оператор  $C$ , (3) состояние вычисления  $Q$  после выполнения данного оператора, записывается в виде:

$$\{P\}C\{Q\}.$$

**Указатель**  $\uparrow$ *Локализатор памяти*, где  $\uparrow$ *содержимое* значение не вещественное число, либо строка, содержимое чьё интерпретируется, как адрес в операционной памяти. Указатели могут ссылаться на указателей.

**Указатель на указателя**  $\uparrow$ *Указатель*, где содержимое содержит адрес указателя.

**Уровень видимости класса**  $\uparrow$ *Поля* и  $\uparrow$ *методы*  $\uparrow$ *класса* имеют видимость, по которой они, либо передаются с урезанной видимостью, либо вообще не передаются в наследованный класс. Уровни видимости включают в себя, например, `private`, `public`, `protected`.

**Утверждение (о свойствах программ)** Утверждение о состоянии вычисления, описывается  $\uparrow$ *формальным языком*, например, логикой  $\uparrow$ *предикатов* первого порядка.

**Утверждение о куче** Опираясь на ссылочную модель, утверждение гласит о том, какие имеются  $\uparrow$ *указатели* и какая структура содержится в  $\uparrow$ *динамической памяти*.

**Утечка памяти**  $\uparrow$ *Ячейки памяти*, которые не доступны от  $\uparrow$ *стековых переменных*, засоряющие  $\uparrow$ *динамическую память*.

**Факт, фактум** Неоспоримый логический факт, который принято считать истинным и далее в области расследования нельзя разделять.

**Формальная грамматика** Грамматика для однозначных, часто исключительно искусственных языков, например, языков программирования, которые определяются как:  $(S, T, NT, P)$ , где  $S \in NT$  некоторый стартовый нетерминал, множество терминалов,  $NT$  множество нетерминалов, а  $P$  множество правил вывода.

**Формальная теория** Множество правил и аксиом для рассматриваемой области, в которой проводятся доказательства.

**Формальный язык** Генерируемый язык  $\uparrow$ формальной грамматикой.

**Формула кучи**  $\uparrow$ Утверждение о куче, которое не имеет несвязанных свободных  $\uparrow$ переменных. Формула принадлежит интерпретации, и, либо истина, либо ложь.

**Фрейм над кучей** Части  $\uparrow$ кучи, которые не меняются при вызове процедуры.

**Фреймворк** Программное обеспечение, которое предоставляет, часто в качестве подключаемой библиотеки, базовые функции, которые пользователь включает в своё приложение, соблюдая конвенции использования и вызова фреймворка. Фреймворк постановляет некоторый сценарий сотрудничества абстрактных типов данных и как они между собой коммуницируют. Для этого, имеются точки соприкосновения и зависимостей. Пользователю важно знать только о точках модификации и расширений, всё остальное, при запуске программы совершается ранее задуманным способом.

**Функционал** Функция высшего порядка. Функция принимает в качестве входных параметров не только другие функции, которые могут далее вызывать, а также функция может вычислять (например, символически и/или частично) обратно.

**Частичная корректность** Свойство корректности программы согласно данной спецификации, при которой программа не всегда завершает свою работу.

**Частичная спецификация куч** Спецификация  $\uparrow$ куч содержит зарезервированные символы для куч, которые означают неявное продолжение спецификации куч.

**Частично-определённая Тройка Хора** Если выполнение программного оператора данной  $\uparrow$ тройки Хора не обязательно завершается, то  $\uparrow$ постусловие данной  $\uparrow$ Тройки Хора не определено.

**Эвристика доказательства** Не доказуемая  $\uparrow$ мета-схема, с помощью которой ожидается, но не гарантируется, ускоренное доказательство. Часто эвристики не завершают доказательства, а лишь преобразуют его в предположительно более удобное  $\uparrow$ состояние.

**Ячейка, памяти (в динамической памяти)** Расположенный непрерывный отрывок памяти, чей размер определен типом соответствующего  $\uparrow$ указателя.

# Литература

- [1] Object Management Group (OMG). *Object Constraint Language*, version 2.2. Object Management Group (OMG). Feb. 2010.  
<http://www.omg.org/spec/OCL/2.2>.
- [2] Martin Abadi. *Baby Modula-3 and a Theory of Object*. Technical Report SRC-RR-95. Systems Research Center, Digital Equipment Corporation, 1993.  
<ftp://gatekeeper.research.compaq.com/pub/DEC/SRC/research-reports/abstracts/src-rr-095.html>
- [3] Martin Abadi, Luca Cardelli. *A Theory of Objects*. Secaucus, New Jersey, USA: Springer, 1996, 396p. ISBN 0387947752.
- [4] Martin Abadi, K. Rustan M. Leino. *A Logic of Object-Oriented Programs*. In: Proc. of the 7th Intl. Joint Conf. CAAP/FASE on Theory and Practice of Software Development. Springer, 1997, p.682–696.
- [5] Serge Abiteboul et.al. *EDOS: Environment for the Development and Distribution of Open Source Software*. In: 1st Intl. Conf. on Open Source Systems. 2005.
- [6] Samson Abramsky, Achim Jung. *Domain Theory*. In: Handbook of Logic in Computer Science. Clarendon Press, Oxford University Press, 1994, 168p.
- [7] Jean-Raymond Abrial. *The B-book – Assigning Programs to Meanings*. Cambridge University Press, 2005, (excerpt from 1–34), 816p. ISBN 978-0-521-02175-3.
- [8] Johnathan Afek, Adi Sharabani. *Dangling Pointers — Smashing the Pointer for Fun and Profit* (Whitepaper from Watchfire Corp.) online. 2007.  
<https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf>
- [9] Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, 1989, 131p. ISBN 0-521-30689-2.
- [10] Pierre America, Jan J. M. M. Rutten. *Elements of Generalized Ultrametric Domain Theory*. In: Journal of Computer and System Sciences, Theoretical Computer Science 39 (1989), p.343–375. DOI 10.1016/S0304-3975(96)80711-0.

- 
- [11] Alexandr Andoni, Krzysztof Onak. *Approximating Edit Distance in Near-Linear Time*. In: SIAM Journal on Computing 41.6 (2012), p.1635–1648.
  - [12] Andrew W. Appel. *Garbage Collection can be Faster than Stack Allocation*. In: Information Processing Letters, Elsevier North-Holland 25.4 (1987), p.275–279. DOI 10.1016/0020-0190(87)90175-X.
  - [13] Andrew W. Appel, Robert Dockins, Xavier Leroy. *A list-machine benchmark for mechanized metatheory*. In: Journal of Automated Reasoning 49.3 (2012), p.453–491. DOI 10.1007/s10817-011-9226-1.
  - [14] Andrew W. Appel, Xavier Leroy. *A List-machine Benchmark for Mechanized Metatheory: (Extended Abstract)*. In: Electronic Notes in Theoretical Computer Science, Elsevier 174.5 (2007), p.95–108. DOI 10.1016/j.entcs.2007.01.020.
  - [15] Krzysztof R. Apt. *Ten Years of Hoare's Logic: A Survey - Part I*. In: ACM Transactions on Programming Languages and Systems 3.4 (1981), p.431–483. ISSN 0164-0925. DOI 10.1145/357146.357150.
  - [16] Krzysztof R. Apt, Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. In: SIAM Review 35.2 (1993), p.330–331.
  - [17] Uwe Assmann, *Trustworthy Instantiation of Frameworks*. In: Architecting Systems with Trustworthy Components. T. 3938/2006. Springer, 2006, p.152–168. ISBN 978-3-540-35800-8. DOI 10.1007/11786160.
  - [18] Mikhail J. Atallah, Susan Fox, eds. *Algorithms and Theory of Computation Handbook*. 1st. Boca Raton, Florida, USA: CRC Press, Oct. 1999, 1312p. ISBN 0849326494.
  - [19] Franz Baader, Tobias Nipkow. *Term Rewriting and All That*. New York, USA: Cambridge University Press, 1998, 297p. ISBN 0-521-45520-0.
  - [20] Greg J. Badros. *JavaML: A Markup Language for Java Source Code*. In: Proc. of the 9th Intl. World Wide Web conf. on Computer Networks, Intl. Journal of Computer and Telecommunications Networking. 2000, p.13–15.
  - [21] Anindya Banerjee, David A. Naumann, Stan Rosenberg. *Regional Logic for Local Reasoning about Global Invariants*. In: European conf. on Object-Oriented Programming. eds. J. Vitek. T. 5142. Lecture Notes in Computer Science. Springer, Berlin Heidelberg, 2008, p.387–411. DOI 10.1007/978-3-540-70592-5\_17.
  - [22] Henk P. Barendregt. *Handbook of Logic in Computer Science*. In: eds. Samson Abramsky, Dov M. Gabbay, Thomas S. E. Maibaum. New York, USA: Oxford University Press, 1992. Chapter on Lambda Calculi with Types, p.117–309. ISBN 0-19-853761-1. DOI 10.1017/CBO9781139032636.

- 
- [23] Michael Barnett, et.al. *Verification of Object-Oriented Programs with Invariants*. In: Journal of Object Technology 3.6 (2004), p.27–56.
  - [24] Kent Beck. *jUnit Testing Framework*. <http://junit.org>.
  - [25] Hans Bekič. *Definable Operation in General Algebras, and the Theory of Automata and Flowcharts*. In: Programming Languages and Their Definition. eds. Cliff B. Jones. T. 177. Lecture Notes in Computer Science. Springer, 1984.
  - [26] Josh Berdine, Cristiano Calcagno, Peter W. O’Hearn. *Smallfoot: Modular Automatic Assertion Checking with Separation Logic*. In: Intl. Symp. on Formal Methods for Components and Objects. 2005, p.115–137. DOI 10.1007/11804192\_6.
  - [27] Josh Berdine, Cristiano Calcagno, Peter W. O’Hearn. *Symbolic Execution with Separation Logic*. In: 3rd Asian Symp. on Programming Languages and Systems. Tsukuba, Japan, Nov. 2005, p.52–68. DOI 10.1007/11575467\_5.
  - [28] Joachim van den Berg, Bart Jacobs. *The LOOP Compiler for Java and JML*. In: Proc. of the 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. 2001, p.299–312.
  - [29] Mark de Berg, *Computational Geometry: Algorithms and Applications*. 3rd. Santa Clara, California, USA: Springer Berlin, Heidelberg, 2008, 386p. ISBN 3540779736.
  - [30] Jan A. Bergstra, Jan Heering, Paul Klint. *Module Algebra*. In: Journal of the ACM 37.2 (1990), p.335–372. DOI 10.1145/77600.77621.
  - [31] Yves Bertot, Benjamin Gregoire, Xavier Leroy. *A Structured Approach to Proving Compiler Optimizations based on Dataflow Analysis*. In: Types for Proofs and Programs, Workshop TYPES 2004. T. 3839. Lecture Notes in Computer Science. Springer, 2006, p.66–81.
  - [32] Yves Bertot, *Interactive Theorem Proving and Program Development — Coq’Art: The Calculus of Inductive Constructions*. eds. Brauer, Rozenberg, Salomaa. Texts in theoretical computer science. Berlin, New York: Springer Publishing, Inc., 2004, p.472. ISBN 3642058809.
  - [33] Al Bessey, *A Few Billion Lines of Code Later: using Static Analysis to find Bugs in the Real World*. In: Communications of the ACM 53.2 (Feb. 2010), p.66–75. DOI 10.1145/1646353.1646374.
  - [34] Richard Bird, Oege de Moor. *Algebra of Programming*. Upper Saddle River, New-Jersey, USA: Prentice-Hall, Inc., 1997, 312p. ISBN 0-13-507245-X.
  - [35] Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Hertfordshire, England: Prentice Hall Intl. UK, 1988, 310p. ISBN 0-13-484189-1.



- 
- [36] Lars Birkedal, Noah Torp-Smith, John C. Reynolds. *Local Reasoning about a Copying Garbage Collector*. In: ACM SIGPLAN-Notes 39.1 (2004), p.220–231. DOI 10.1145/982962.964020.
  - [37] Lars Birkedal, Noah Torp-Smith, Hongseok Yang. *Semantics of Separation-Logic Typing and Higher-order Frame Rules for Algol-like Languages*. In: Logic in Computer Science 2.5 (2006). DOI 10.2168/LMCS-2(5:1)2006.
  - [38] Lars Birkedal, Hongseok Yang. *Relational Parametricity and Separation Logic*. In: Foundations of Software Science and Computational Structures. 2007, p.93–107. DOI 10.1007/978-3-540-71389-0\_8.
  - [39] Lars Birkedal, *A Simple Model of Separation Logic for Higher-Order Store*. In: Intl. Colloquium on Automata, Languages and Programming. 2008, p.348–360. DOI 10.1007/978-3-540-70583-3\_29.
  - [40] Stephen M. Blackburn, Perry Cheng, Kathryn S. McKinley. *Myths and Realities: The Performance Impact of Garbage Collection*. In: Proc. of the Joint Intl. Conf. on Measurement and Modeling of Computer Systems. SIGMETRICS'04. New York, USA: ACM, 2004, p.25–36. ISBN 1-58113-873-3. DOI 10.1145/1005686.1005693.
  - [41] Sandrine Blazy, Zaynah Dargaye, Xavier Leroy. *Formal Verification of a C Compiler Front-End*. In: Intl. Symp. on Formal Methods. T. 4085. Lecture Notes in Computer Science. Springer, Aug. 2006, p.460–475.
  - [42] Sandrine Blazy, Xavier Leroy. *Formal Verification of a Memory Model for C-like Imperative Languages*. In: Intl. Conf. on Formal Engineering Methods. T. 3785. Lecture Notes in Computer Science. Springer, 2005, p.280–299.
  - [43] Sandrine Blazy, Xavier Leroy. *Mechanized Semantics for the Clight Subset of the C Language*. In: Journal of Automated Reasoning 43.3 (2009), p.263–288. DOI 10.1007/s10817-009-9148-3.
  - [44] Richard Bornat. *Proving Pointer Programs in Hoare Logic*. In: Proc. of the 5th Intl. Conf. on Mathematics of Program Construction. eds. Roland Backhouse, José Nuno Oliveira. T. 1. Lecture Notes in Computer Science, Springer 8. Ponte de Lima, Portugal, 2000, p.102–126. DOI 10.1007/10722010\_8.
  - [45] Marius Bozga, Radu Iosif, Swann Perarnau. *Quantitative Separation Logic and Programs with Lists*. In: Intl. Joint conf. on Automated Reasoning. 2008, p.34–49. DOI 10.1007/978-3-540-71070-7\_4.
  - [46] Ivan Bratko. *Prolog: Programming for Artificial Intelligence*. 3rd. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, 673p. ISBN 0-201-40375-7.
  - [47] Luitzen Egbertus Jan Brouwer. *Brouwer's Intuitionistic Logic*. online free encyclopedia from May 2010. <http://plato.stanford.edu/entries/intuitionism>.

- 
- [48] Kim B. Bruce. *Foundations of Object-oriented Languages: Types and Semantics*. Cambridge, Massachusetts, USA: MIT Press, 2002, 384p. ISBN 0-262-02523-X.
  - [49] Nicolaas Govert de Bruijn. *Lambda Calculus Notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), p.381–392. ISSN 1385-7258. DOI 10.1016/1385-7258(72)90034-0.
  - [50] Janusz A. Brzozowski. *Derivatives of Regular Expressions*. In: *Journal of the ACM* 11.4 (Oct. 1964), p.481–494. ISSN 0004-5411. DOI 10.1145/321239.321249.
  - [51] Rodney M. Burstall. *Some Techniques for Proving Correctness of Programs which Alter Data Structures*. In: *Machine Intelligence*. eds. Bernard Meltzer, Donald Michie. T. 7. Scotland: Edinburgh University Press, 1972, p.23–50.
  - [52] Cristiano Calcagno, Simon Helsen, Peter Thiemann. *Syntactic Type Soundness Results for the Region Calculus*. In: *Information and Computation* 173 (2001), p.173–2. DOI 10.1006/inco.2001.3112.
  - [53] Cristiano Calcagno, Peter O’Hearn, Richard Bornat. *Program Logic and Equivalence in the Presence of Garbage Collection*. In: *Theoretical Computer Science, Foundations of Software Science and Computation Structures* 298.3 (2003), p.557–581. ISSN 0304-3975. DOI 10.1016/S0304-3975(02)00868-X.
  - [54] Cristiano Calcagno, *Compositional Shape Analysis by means of Bi-abduction*. In: *Proc. of the 36th annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* 36 (2009), p.289–300. DOI 10.1145/1480881.1480917.
  - [55] Luca Cardelli. *Type Systems*. In: *The Computer Science and Engineering Handbook*. eds. Allen B. Tucker. CRC Press, 1997. p.2208–2236.
  - [56] Luca Cardelli, Martin Abadi. *A Theory of Objects*, Digital Equipment Corporation, invited workshop presentation in Sydney, Australia on 19th December 1997, 574 slides.
  - [57] Rudolf Carnap. *Logische Syntax der Sprache*. 2nd. Vienna, Austria: Springer-Verlag, 1968, 274p. ISBN 978-3-662-23331-3.
  - [58] Ramkrishna Chatterjee. *Modular Data-flow Analysis of Statically Typed Object-Oriented Programming Languages*. PhD thesis. . . . Rutgers University, Jan. 2000.
  - [59] Yoonsik Cheon, Gary T. Leavens. *A Thought on Specification Reflection*. In: *Proc. of the 8th World Multi-conf. on Systemics, Cybernetics and Informatics, Orlando, Florida, USA, Volume II, Computing Techniques* (appeared on Dec 2003 as TR03-16 at Iowa State University). eds. N. Callaos, W. Lesso, B. Sanchez. Jul. 2004, p.485–490.

- 
- [60] Sigmund Cherm, Radu Rugina. *Maintaining Doubly-Linked List Invariants in Shape Analysis with Local Reasoning*. In: Intl. Workshop on Verification, Model Checking and Abstract Interpretation. eds. Byron Cook, Andreas Podelski. T. 4349. 2007, p.234–250. ISBN 978-3-540-69738-1. DOI 10.1007/978-3-540-69738-1\_17.
  - [61] John C. Cherniavsky. *Simple Programs realize exactly Presburger formulas*. In: SIAM Journal on Computing 5.4 (1976), p.666–677.
  - [62] Edmund Melson Jr. Clarke. *Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Axiom Systems*. In: Journal of the ACM, New York, USA 26.1 (Feb. 1979), p.129–147. DOI 10.1145/322108.322121.
  - [63] Edmund Melson Jr. Clarke, Orna Grumberg, Doron A. Peled. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999, 330p. ISBN 0-262-03270-8.
  - [64] Maurice Clint. *Program Proving: Coroutines*. In: Acta Informatica 2 (1973), p.50–63. DOI 10.1007/BF00571463.
  - [65] Avra Cohn. *The Equivalence of two Semantic Definitions — A Case Study in LCF*. In: SIAM Journal on Computing 12.2 (1983), p.267–285.
  - [66] H. Comon et.al, *Tree Automata Techniques and Applications*, 2007.  
<http://www.grappa.univ-lille3.fr/tata>.
  - [67] Stephen Cook. *Soundness and Completeness of an Axiom System for Program Verification*. In: SIAM Journal on Computing 7.1 (1978), p.70–90.
  - [68] Stephen A. Cook. *The Complexity of Theorem-Proving Procedures*. In: Proc. of the 3rd annual ACM Symp. on Theory of Computing. 1971, p.151–158. DOI 10.1145/800157.805047.
  - [69] David C. Cooper. *Theorem Proving in Arithmetic without Multiplication*. In: Machine Intelligence 7. eds. Bernard Meltzer, Donald Michie. Edinburgh University Press, 1972, p.91–100. ISBN 0-85224-234-4.
  - [70] Keith D. Cooper, Ken Kennedy. *Fast Interprocedural Alias Analysis*. In: Proc. of the 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, New York, USA, Jan. 1989, p.49–59. DOI 10.1145/75277.75282.
  - [71] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. *Linux Device Drivers*. 3rd. O'Reilly Media, Inc., 2005, 597p. ISBN 9780596005900.
  - [72] Thomas H. Cormen, *Introduction to Algorithms*. 3rd. MIT Press, Jul. 2009, 1292p. ISBN 0262033844.

- 
- [73] Patrick Cousot, Radhia Cousot. *Abstract Interpretation and Application to Logic Programs*. In: Journal of Logic Programming 13.2&3 (1992), p.103–179.
  - [74] Patrick Cousot, Radhia Cousot. *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: Proc. of the 4th ACM SIGACT-SIGPLAN symp. on Principles of Programming Languages. eds. ACM. Jan. 1977, p.238–252.
  - [75] John Criswell, LLVM group. *SAFECode project*, University of Illinois at Urbana-Champaign, USA. <http://llvm.org>.
  - [76] Ron Cytron, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. In: ACM Transactions on Programming Languages and Systems 13.4 (Oct. 1991), p.451–490. ISSN 0164-0925. DOI 10.1145/115372.115320.
  - [77] Desmond F. D’Souza, Alan Cameron Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley, 1998, 816p. ISBN 9780201310122.
  - [78] Zaynah Dargaye, Xavier Leroy. *Mechanized Verification of CPS transformations*. In: 14th Intl. Conf on Logic for Programming, Artificial Intelligence and Reasoning. T. 4790. Lecture Notes in Artificial Intelligence. Springer, 2007, p.211–225.
  - [79] Martin D. Davis, Ron Sigal, Elaine J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*. 2nd. San Diego, California, USA: Academic Press Professional, Morgan Kaufman Inc., 1994, 609p. ISBN 0-12-206382-1.
  - [80] Anatoli Degtyarev, Andrei Voronkov. *The Inverse Method*. In: Handbook of Automated Reasoning (in 2 volumes). 2001, p.179–272.
  - [81] Enrico Denti, Andrea Omicini, Alessandro Ricci. *Multi-paradigm Java-Prolog Integration in tuProlog*. In: Science of Computer Programming, Elsevier Science 57.2 (Aug. 2005), p.217–250. ISSN 0167-6423. DOI 10.1016/j.scico.2005.02.001.
  - [82] Enrico Denti, Andrea Omicini, Alessandro Ricci. *tuProlog: A Light-weight Prolog for Internet Applications and Infrastructures*. In: Practical Aspects of Declarative Languages. eds. Ramakrishnan. T. 1990. Lecture Notes in Computer Science. Springer Berlin, Heidelberg, 2001, p.184–198. ISBN 978-3-540-41768-2. DOI 10.1007/3-540-45241-9\_13.
  - [83] Daniel Diaz, Salvador Abreu, Philippe Codognet. *On the Implementation of GNU Prolog*. In: Theory and Practice of Logic Programming 12.1-2 (2012), p.253–282. DOI 10.1017/S1471068411000470.
  - [84] Christophe de Dinechin. *C++ Exception Handling for IA64*. In: edition 8. Oct. 2000, p.72–79. <http://www.usenix.org/publications/library/proceedings/osdi2000/wiess2000/dinechin.html>

- 
- [85] Dino Distefano, Peter W. O'Hearn, Hongseok Yang. *A Local Shape Analysis Based on Separation Logic*. In: Proc. of the 12th intl. conf. on Tools and Algorithms for the Construction and Analysis of Systems. Под ред. Heidelberg Springer Berlin. Mar. 2006, p.287–302. DOI 10.1007/11691372\_19.
  - [86] Dino Distefano, Matthew J. Parkinson. *jStar: Towards Practical Verification for Java*. In: Object-Oriented Programming, Systems, Languages and Applications. 2008, p.213–226. DOI 10.1145/1449764.1449782.
  - [87] Mike Dodds. *Graph Transformations and Pointer Structures*. PhD thesis. . . . University of York, England, 2008, 277p.
  - [88] Damien Doligez, Xavier Leroy. *A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML*. In: 20th symp. on Principles of Programming Languages. ACM Press, 1993, p.113–123.
  - [89] Kosta Dosen, *Substructural Logics*. eds. Peter Schroeder-Heister, Kosta Dosen. Clarendon, Oxford University Press, 1993, 400p. ISBN 0198537778.
  - [90] Hartmut Ehrig, Barry K. Rosen. *The Mathematics of Record Handling*. In: SIAM Journal on Computing 9.3 (1980), p.441–469. DOI 10.1137/0209034.
  - [91] *ElectricFence Project*. <http://perens.com/FreeSoftware>.
  - [92] Pär Emanuelsson, Ulf Nilsson. *A Comparative Study of Industrial Static Analysis Tools*. In: Electronic Notes in Theoretical Computer Science 217 (2008), p.5–21. DOI 10.1016/j.entcs.2008.06.039.
  - [93] Loe M. G. Feijs, Yuechen Qian. *Component Algebra*. In: Science of Computer Programming 42.2-3 (2002), p.173–228. DOI 10.1016/S0167-6423(01)00009-0.
  - [94] Bernd Fischer. *Specification-Based Browsing of Software Component Libraries*. In: Automated Software Engineering 7.2 (May 2000), p.179–200. DOI 10.1023/A: 1008766409590.
  - [95] Michael J. Fisher, Michael O. Rabin. *Super-Exponential Complexity of Presburger Arithmetic*. Technical Report no.889578. Cambridge, Massachusetts, USA: Massachusetts Institute of Technology, 1974, p.27–41.
  - [96] Cormac Flanagan, *Extended Static Checking for Java*. In: Proc. of the ACM SIGPLAN conf. on Programming Languages Design and Implementation. 2002, p.234–245.
  - [97] Robert W. Floyd. *Assigning Meanings to Programs*. In: Mathematical Aspects of Computer Science. Под ред. T.R. Colburn, J.H. Fetzer, Rankin T.L. T. 19. Proc. of symp. in Applied Mathematics. American Mathematical Society. 1967, p.19–32.

- 
- [98] Matthew Fluet, Greg Morrisett, Amal J. Ahmed. *Linear Regions Are All You Need*. In: European Symp. on Programming. 2006, p.7–21. DOI 10.1007/11693024\_2.
  - [99] Ira R. Forman, Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications, Oct. 2004, 273p. ISBN 1932394184.
  - [100] Free Software Foundation. *GNU make*, <https://www.gnu.org/software/make/>.
  - [101] Michael L. Fredman, Robert Endre Tarjan. *Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms*. In: Journal of the ACM 34.3 (Jul. 1987), p.596–615. ISSN 0004-5411. DOI 10.1145/28869.28874.
  - [102] Jean H. Gallier. *Foundations of Automatic Theorem Proving*. Harper & Row, Jun. 2003, 534p. ISBN 0-06-042225-4.
  - [103] Holger Gast. *Lightweight Separation*. In: Intl. Conf. on Theorem Proving in Higher Order Logics. 2008, p.199–214. DOI 10.1007/978-3-540-71067-7\_18.
  - [104] Susan L. Gerhart. *Proof Theory of Partial Correctness Verification Systems*. In: SIAM Journal on Computing 5.3 (1976), p.355–377.
  - [105] Alain Giorgetti, *Specifying Generic Java programs: Two Case Studies*. In: Proc. of the of the 10th Workshop on Language Descriptions, Tools and Applications, LDTA/ETAPS 2010, Paphos, Cyprus, Mar. 28-29, 2010. p.1–8. DOI 10.1145/1868281.1868289.
  - [106] Fausto Giunchiglia, Toby Walsh. *Abstract Theorem Proving: Mapping Back (unpublished)*. 1989.
  - [107] John B. Goodenough. *Exception Handling: Issues and a Proposed Notation*. In: 1975, p.683–696.
  - [108] Alexey Gotsman, Josh Berdine, Byron Cook. *Interprocedural Shape Analysis with Separated Heap Abstractions*. In: Intl. Static Analysis Symp. Springer, 2006.
  - [109] Dan Grossman, *Region-Based Memory Management in Cyclone*. In: Journal of Programming Language Design and Implementation. 2002, p.282–293. DOI 10.1145/512529.512563.
  - [110] Dick Grune, Criel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Upper Saddle River, New Jersey, USA: Ellis Horwood Publishing, 1990, 200p. ISBN 0-13-651431-6.
  - [111] Carl A. Gunter, John C. (eds.) Mitchell. *Theoretical Aspects of Object-Oriented Programming — Types, Semantics, and Language Design*. In: MIT Press, 1994. ISBN 026207155X.
  - [112] Maurice H. Hålsted. *Elements of Software Science*. New York, USA: Elsevier Science, 1977. ISBN 0444002057.

- 
- [113] Michael Hind. *Pointer Analysis: Haven't We Solved This Problem Yet?* In: Proc. of the ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering. eds. USA IBM Watson Research Center. ACM, Jun. 2001, p.54–61. ISBN 1-58113-413-4. DOI 10.1145/379605.379665.
  - [114] Michael Hind, Anthony Pioli. *Evaluating The Effectiveness of Pointer Alias Analyses*. In: Science of Computer Programming. 1999, p.31–55. DOI 10.1016/S0167-6423(00)00014-9.
  - [115] Roger Hindley. *The Principal Type-Scheme of an Object in Combinatory Logic*. In: Transactions of the American Mathematical Society 146 (Dec. 1969), p.29–60. DOI DOI 10.2307/1995158.
  - [116] Charles A. R. Hoare. *An Axiomatic Basis for Computer Programming*. In: Communications of the ACM 12.10 (Oct. 1969), p.576–580. ISSN 0001-0782. DOI 10.1145/363235.363259.
  - [117] Kohei Honda, Nobuko Yoshida, Martin Berger. *An Observationally Complete Program Logic for Imperative Higher-Order Functions*. In: Logic in Computer Science. 2005, p.270–279. DOI 10.1109/LICS.2005.5.
  - [118] Susan Horwitz, Phil Pfeiffer, Thomas Reps. *Dependence Analysis for Pointer Variables*. In: Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland, Oregon, USA: ACM, Jul. 1989, p.28–40. ISBN 0-89791-306-X. DOI 10.1145/73141.74821.
  - [119] Xia-Yu Hu, Robert Haas. *The Fundamental Limit of Flash Random Write Performance: Understanding, Analysis and Performance Modelling*. Technical Report no.99781. IBM Research Zürich, Switzerland, Mar. 2010.
  - [120] Marieke Huisman, Clément Hurlin. *The Stability Problem for Verification of Concurrent Object-Oriented Programs*. In: Electronic Notes in Theoretic Computer Science (2007).
  - [121] Clément Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis . . . Université Nice – Sophia Antipolis, France, Sep. 2009, 207p.
  - [122] Graham Hutton. *Fold and Unfold for Program Semantics*. In: Proc. of the 3rd ACM SIGPLAN Intl. Conf. on Functional Programming. Baltimore, Maryland, USA: ACM Press, 1998, p.280–288.
  - [123] Intel Inc. *Intel 64 and IA-32 Architectures*, Software Developer's Manual, Volume 3A: System Programming Guide, Part 1 (online). Oct. 2011.
  - [124] Bart Jacobs, et.al *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*. In: Proc. of the 3rd Intl. Conf. on NASA Formal Methods. Pasadena, California, USA: Springer, Berlin Heidelberg, Apr. 2011, p.41–55. ISBN 978-3-642-20397-8.

- 
- [125] Patricia Johann, Eelco Visser. *Strategies for Fusing Logic and Control via Local, Application-Specific Transformations*. Technical Report no.UU-CS-2003-050. Institute of Information, Computing Sciences, Utrecht University, 2003. DOI 10.1.1.10.8859.
  - [126] Neil D. Jones, Steven S. Muchnick. *Even Simple Programs are Hard to Analyze*. In: Proc. of 2nd ACM SIGACT-SIGPLAN symp. on Principles of Programming Languages. 1975, p.106–118. DOI 10.1145/512976.512988.
  - [127] Richard Jones, Antony Hosking, Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 1st. Chapman & Hall/CRC, 2011, 481p. ISBN 1420082795.
  - [128] Richard Jones, Sun Microsystems Inc. *Memory Management in the Java HotSpot Virtual Machine*. Technical Report Apr. 2006.  
<http://www.oracle.com/technetwork/articles/java/index-jsp-140228.html>.
  - [129] Jacques-Henri Jourdan, Francois Pottier, Xavier Leroy. *Validating LR(1) Parsers*. In: Programming Languages and Systems – 21st European symp. on Programming, ESOP 2012. T. 7211. Lecture Notes in Computer Science. Springer, 2012, p.397–416. DOI 10.1007/978-3-642-28869-2\_20.
  - [130] Michel Kaempf. *Smashing The Heap For Fun And Profit*, Jul. 2006, version from 26.09.2015.  
<https://web.archive.org/web/20060713194734/>,  
<http://doc.bughunter.net/buffer-overflow/heap-corruption.html>.
  - [131] Laura Kallmeyer. *Parsing Beyond Context-Free Grammars*. eds. Dov M. Gabbay, Jörg Siekmann. Cognitive Technologies. Springer Heidelberg, 2010, 246p. ISBN 978-3-642-14845-3.
  - [132] Shmuel Katz, Zohar Manna. *A Heuristic Approach to Program Verification*. In: Proc. of the 3rd intl. joint Conf. on Artificial Intelligence. 1973, p.500–512.
  - [133] Matt Kaufmann, Panagiotis Manolios, Strother Moore, (eds.) *Computer-Aided Reasoning: ACL2 Case Studies, Applicative Common LISP*. 2nd. Kluwer Academic Publishers, 2008, 285p. ISBN 0-7923-7849-0.
  - [134] Matt Kaufmann, J. Strother Moore. *Some Key Research Problems in Automated Theorem Proving for HW/SW-Verification*. In: Revista de la Real Academia de Ciencias Exactas, Serie A Matemáticas 98.1 (2004), p.181–195. ISSN 1578-7303.
  - [135] Ken Kennedy, John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, USA: Morgan Kaufmann Publishers Inc., 2002, 790p. ISBN 1-55860-286-0.
  - [136] Joshua Kerievsky. *Refactoring to Patterns*. 2nd. Addison-Wesley, 2005, 384p. ISBN 3-8273-2262-6.



- 
- [137] Uday Khedker, Amitabha Sanyal, Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. 1st. Boca Raton, Florida, USA: CRC Press, Inc., 2009, 402p. ISBN 0849328802.
  - [138] Peter Kim. *The Hacker Playbook 2 — Practical Guide to Penetration Testing*. Secure Planet LLC, Jul. 2015, p.339. ISBN 1512214566.
  - [139] Christian Kirsch. *Entwicklertrauma, Marktübersicht: Tools für die C-Entwicklung — Hilfe gegen Speicher- und Pointer-Fehler*. In: Magazin für professionelle Informationstechnik, iX 6 (1994), p.124–129.
  - [140] Christian Kirsch. *Volle Checkung – C-Laufzeit-Debugger unter Linux*. In: Magazin für professionelle Informationstechnik, iX 9 (1995), p.88–91.
  - [141] Christian Kirsch. *Zeig's mir – Freie Speichertools ElectricFence und Valgrind (german)*. In: Magazin für professionelle Informationstechnik, iX 3 (2003), p.82–84.  
[https://www.heise.de/artikel-archiv/ix/2003/03/082\\_Zeig-s-mir](https://www.heise.de/artikel-archiv/ix/2003/03/082_Zeig-s-mir)
  - [142] Hans Koch, Alain Schenkel, Peter Wittwer. *Computer-Assisted Proofs in Analysis and Programming in Logic – A Case Study*. In: SIAM Review 38.4 (1996), p.565–604.
  - [143] Rajeev Kohli, Ramesh Krishnamurti, Prakash Mirchandani. *The Minimum Satisfiability Problem*. In: SIAM Journal on Discrete Mathematics 7.2 (1994), p.275–283.
  - [144] Robert A. Kowalski. *Predicate Logic as Programming Language*. In: Information Processing (IFIP). North-Holland Publishing, 1974, p.569–574.
  - [145] Neelakantan Krishnaswami. *Verifying Higher Order Imperative Programs with Higher Order Separation Logic (unpublished at Carnegie Mellon University)*, 2008.  
<http://www.cs.cmu.edu/~neelk>
  - [146] Neelakantan R. Krishnaswami, *Design patterns in Separation Logic*. In: Proc. of 4th Intl. Workshop on Types in Language Design and Implementation. 2009, p.105–116. DOI 10.1145/1481861.1481874.
  - [147] Daniel Kröning, Georg Weissenbacher. *Model Checking: Bugs in C-Programmen finden*. In: Magazin für professionelle Informationstechnik, iX 5 (2009), p.159–162.  
[http://www.heise.de/artikel-archiv/ix/2009/05/159\\_Drum-pruefe](http://www.heise.de/artikel-archiv/ix/2009/05/159_Drum-pruefe)
  - [148] Marta Kwiatkowska, Gethin Norman, David Parker. *Stochastic Model Checking*. In: Proc. of the 7th Intl. Conf. on Formal Methods for Performance Evaluation. Bertinoro, Italy: Springer, 2007, p.220–270.
  - [149] Sven Lämmermann. *Runtime Service Composition via Logic-based Program Synthesis*. PhD thesis. . . . Dpt. of Microelectronics, Information Technology, Royal Institute of Technology, Stockholm, Sweden, 2002, 204p.

- 
- [150] William Landi. *Undecidability of Static Analysis*. In: ACM Letters on Programming Languages and Systems 1 (1992), p.323–337.
  - [151] William Landi, Barbara G. Ryder. *Pointer-Induced Aliasing: A Problem Classification*. In: ACM Principles of Programming Languages. 1991, p.93–103. DOI 10.1145/99583.99599.
  - [152] Peter J. Landin. *The Mechanical Evaluation of Expressions*. In: Computer Journal 6.4 (Jan. 1964), p.308–320.
  - [153] Richard G. Larson. *Minimizing Garbage Collection as a Function of Region Size*. In: SIAM Journal on Computing 6.4 (1977), p.663–668. DOI 10.1137/0206047.
  - [154] Chris Lattner, Vikram Adve. *Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis*. Technical Report no.UIUCDCS-R-2003-2340. Computer Science Dpt., University of Illionois at Urbana-Champaign, USA, 2003, p.1–20.  
<http://lvm.org/pubs/2003-04-29-DataStructureAnalysisTR.html>
  - [155] K. Rustan M. Leino. *Recursive Object Types in a Logic of Object-Oriented Programs*. In: Nordic Journal of Computing 5.4 (Apr. 1998), p.330–360. ISSN 1236-6064.
  - [156] K. Rustan M. Leino, Greg Nelson. *Data Abstraction and Information Hiding*. In: ACM Transactions on Programming Languages and Systems 24 (2002), p.491–553.
  - [157] Xavier Leroy. *A Formally Verified Compiler Back-end*. In: Journal of Automated Reasoning 43.4 (2009), p.363–446. DOI 10.1007/s10817-009-9155-4.
  - [158] Xavier Leroy. *Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant*. In: 33rd symp. on Principles of Programming Languages. ACM Press, 2006, p.42–54.
  - [159] Xavier Leroy. *Formal Verification of a Realistic Compiler*. In: Communications of the ACM 52.7 (2009), p.107–115. DOI 10.1145/1538788.1538814.
  - [160] Xavier Leroy. *Verified Squared: Does Critical Software Deserve Verified Tools?* In: 38th symp. Principles of Programming Languages. Abstract of invited lecture. ACM Press, 2011, p.1–2. DOI 10.1145/1926385.1926387.
  - [161] Xavier Leroy, *The CompCert Memory Model*, Version 2. Research report RR-7987. INRIA, France, Jun. 2012.
  - [162] John R. Levine. *flex and bison — Unix text processing tools*. O'Reilly, 2009, p.292. ISBN 978-0-596-15597-1.

- 
- [163] John R. Levine. *Linkers and Loaders*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, 256p. ISBN 1558604960.
  - [164] Robert Love. *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010, 440p. ISBN 0672329468.
  - [165] Deborah Y. Macock. *Automated Theorem-proving and Program Verification: An Annotated Bibliography*. Technical Report no.CS75027-R. Virginia Polytechnic Institute, State University, Blacksburg, Virigina, USA, Nov. 1975.
  - [166] David MacQueen, Gordon Plotkin, Ravi Sethi. *An Ideal Model for Recursive Polymorphic Types*. In: Proc. of the 11th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. New York, USA: ACM, 1984.
  - [167] Darko Marinov. *Automated Testing of Refactoring Engines Using Test Abstractions*.  
<http://www.researchchannel.org/prog/displayevent.aspx?rID=28186&fID=5919>.
  - [168] Clive Matthews. *An Introduction to Natural Language Processing Through Prolog*. 1st. White Plains, New York, USA: Longman Publishing, 1998, 306p. ISBN 0582066220.
  - [169] Farhad Mehta, Tobias Nipkow. *Proving Pointer Programs in Higher-Order Logic*. In: Information and Computation 199.1-2 (2005), p.200–227. DOI 10.1016/j.ic.2004.10.007.
  - [170] Tim Menzies. *Applications of Abduction: Knowledge-Level Modeling*. In: Intl. Journal of Human-Computer Studies 45 (Mar. 1996), p.305–335. DOI 10.1006/ijhc.1996.0054.
  - [171] Jason Merrill. *Generic and Gimple: A New Tree Representation for Entire Functions*. In: In Proc. of the 2003 GCC Developers Summit. 2003, p.171–180.
  - [172] Bertrand Meyer. *Applying Design by Contract*. In: IEEE Computer 25.10 (Oct. 1992), p.40–51. ISSN 0018-9162. DOI 10.1109/2.161279.
  - [173] Bertrand Meyer. *Proving Pointer Program Properties – Part 1: Context and overview, Part 2: The Overall Object Structure*. In: ETH Zürich. Journal of Object Technology, 2003.
  - [174] Bertrand Meyer. *Proving Pointer Program Properties – Part 2: The Overall Object Structure*. In: ETH Zürich. Journal of Object Technology, 2003.
  - [175] Bertrand Meyer, Christine Mingins, Heinz W. Schmidt. *Providing Trusted Components to the Industry*. In: IEEE Computer 31.5 (1998), p.104–105.
  - [176] Barton P. Miller, Lars Fredriksen, Bryan So. *An Empirical Study of the Reliability of UNIX Utilities*. In: In Proc. of the Workshop of Parallel and Distributed Debugging. Digital Equipment Corporation, 1990, p.1–22.

- 
- [177] Barton P. Miller, *Nichts dazugelernt – Empirische Studie zur Zuverlässigkeit von Unix-Utilities*. In: Magazin für professionelle Informationstechnik, iX 9 (1995), p.108–121.  
[http://www.heise.de/artikel-archiv/ix/1995/09/108\\_Nichts-dazu-gelernt](http://www.heise.de/artikel-archiv/ix/1995/09/108_Nichts-dazu-gelernt)
  - [178] Robin Milner. *A Theory of Type Polymorphism in Programming*. In: Journal of Computer and System Sciences 17.3 (Dec. 1978), p.348–375. ISSN 0022-0000. DOI 10.1016/0022-0000(78)90014-4.
  - [179] John C. Mitchell. *Foundations for Programming Languages*. 2nd. MIT Press, Cambridge, Massachusetts, London, England, 1996, 846p.
  - [180] Joël Moses. *The Function of FUNCTION in LISP, or Why the FUNARG Problem Should be Called the Environment Problem*. In: (1970).  
<http://hdl.handle.net/1721.1/5854>.
  - [181] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 2007, 856p. ISBN 978-1-55860-320-2.
  - [182] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis. . . . Fern-Universität Hagen, Germany, 2002, 261p.
  - [183] Nomair A. Naeem, Ondrej Lhoták. *Efficient Alias Set Analysis using SSA form*. In: Proc. of the Intl. Symp. on Memory Management. Dublin, Ireland: ACM, 2009, p.79–88. ISBN 978-1-60558-347-1. DOI 10.1145/1542431.1542443.
  - [184] Lee Naish. *Higher-Order logic programming*. Technical Report no.96/2. Dpt. of Computer Science, University of Melbourne, Australia, Feb. 1996, 15p.
  - [185] Aleksandar Nanevski, Greg Morrisett, Lars Birkedal. *Polymorphism and Separation in Hoare Type Theory*. In: Proc. of the 11th ACM SIGPLAN Intl. conf. on Functional Programming. New York, USA: ACM, 2006, p.62–73. ISBN 1-59593-309-3. DOI 10.1145/1159803.1159812.
  - [186] Aleksandar Nanevski, J. Gregory Morrisett, Lars Birkedal. *Hoare Type Theory, Polymorphism and Separation*. In: Journal on Functional Programming 18.5-6 (2008), p.865–911. DOI 10.1017/S0956796808006953.
  - [187] Aleksandar Nanevski, *YNot: Reasoning with the awkward squad*. In: Proc. of 13th ACM SIGPLAN Intl. Conf. on Functional Programming. Victoria, British Columbia, Canada, Sep. 2008.
  - [188] Charles G. Nelson, Derek C. Oppen. *A Simplifier Based on Efficient Decision Algorithms*. In: Proc. of the 5th ACM SIGSOFT-SIGPLAN symp. on Principles of Programming Languages. Jan. 1978, p.141–150. DOI 10.1145/512760.512775.

- 
- [189] Flemming Nielson, Hanne R. Nielson, Chris Hankin. *Principles of Program Analysis*. Springer Berlin, Heidelberg, 1999, 452p. ISBN 978-3-540-65410-0.
  - [190] Peter W. O'Hearn, Hongseok Yang, John C. Reynolds. *Separation and Information Hiding*. In: ACM Transactions on Programming Languages and Systems. 2004, p.268–280. DOI 10.1145/964001.964024.
  - [191] Ross Overbeek. *Book Review on Larry Wos: Automated Reasoning: 33 Basic Research Problems*. In: Journal of Automated Reasoning (1988), p.233–234. DOI 10.1007/BF00244397.
  - [192] Sascha A. Parduhn, Raimund Seidel, Reinhard Wilhelm. *Algorithm Visualization using Concrete and Abstract Shape Graphs*. In: Proc. of the ACM symp. on Software Visualization. 2008, p.33–36. DOI 10.1145/1409720.1409726.
  - [193] Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis. . . . Cambridge University, England, 2005, 169p.
  - [194] Matthew J. Parkinson. *When Separation Logic met Java*. Microsoft Research Cambridge, England, online resource from 13.08.2014.  
<https://www.microsoft.com/en-us/research/video/when-separation-logic-met-java>.
  - [195] Matthew Parkinson, Gavin Bierman. *Separation Logic and Abstraction*. In: SIGPLAN Notes 40.1 (2005), p.247–258. DOI 10.1145/1047659.1040326.
  - [196] Nick Parlante. *Linked List Basics*, document no.103 from the Stanford Computer Science Education Library, 2001. <http://cslibrary.stanford.edu/103>.
  - [197] Terrence Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Programmers, 2012, 328p. ISBN 978-1-93435-699-9.
  - [198] Lawrence C. Paulson. *The Foundation of a Generic Theorem Prover*. Technical Report, Computer Laboratory, University of Cambridge, England, May 1989, p.363–397. DOI 10.1007/BF00248324.
  - [199] Viktor Pavlu. *Shape-Based Alias Analysis — Extracting Alias Sets from Shape Graphs for Comparison of Shape Analysis Precision*. Master thesis. . . . Vienna University of Technology, Austria, Mar. 2010, p.117.
  - [200] Fernando C. N. Pereira, David H.D. Warren. *Definite Clause Grammars for Language Analysis — A Survey of the Formalism and a Comparison with Augmented Transition Networks*. In: Artificial Intelligence 13 (1980), p.231–278.
  - [201] Fernando C.N. Pereira, Stuart M. Shieber. *Prolog and Natural-Language Analysis*. 3rd. Microtome Publishing, Brookline Massachusetts, 2002, 384p. ISBN 0-9719777-0-4.

- 
- [202] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. 1st. Cambridge, Massachusetts, USA: MIT Press, 1991, 114p. ISBN 0-262-660717-0.
  - [203] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, Massachusetts, USA: MIT Press, 2002, 648p. ISBN 0-262-16209-1.
  - [204] Andrew M. Pitts. *Nominal Logic: A First Order Theory of Names and Binding*. In: Information and Computation. Academic Press, 2002, p.165–193.
  - [205] Andrew M. Pitts. *Relational Properties of Domains*. In: Information and Computation 127 (1996), p.66–90.
  - [206] David A. Plaistead. *Theorem Proving with Abstraction, Part I+II*. Technical Report no.UIUCDCS R-79-961. Dpt. of Computer Science, University of Illinois, USA, Feb. 1979.
  - [207] David A. Plaistead. *The Undecidability of Self-Embedding for Term Rewriting Systems*. In: Information Processing Letters 20.2 (Feb. 1985), p.61–64. DOI 10.1016/0020-0190(85)90063-8.
  - [208] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. In: Journal of Logic and Algebraic Programming 60-61 (Dec. 2004), p.17–139. DOI 10.1016/j.jlap.2004.05.001.
  - [209] Gordon D. Plotkin. *LCF Considered as a Programming Language*. In: Theoretical Computer Science 5.3 (1977), p.223–255. DOI 10.1016/0304-3975(77)90044-5.
  - [210] Claude Pommerell, Wolfgang Fichtner. *Memory Aspects and Performance of Iterative Solvers*. In: SIAM Journal on Scientific Computing 15.2 (март 1994), p.460–473. ISSN 1064-8275. DOI 10.1137/0915031.
  - [211] Francois Pottier. *Hiding Local State in Direct Style: A Higher-Order Anti-Frame Rule*. In: Proc. of the 23rd Annual IEEE symp. on Logic in Computer Science. Washington, DC, USA: IEEE Computer Society, 2008, p.331–340. DOI 10.1109/LICS.2008.16.
  - [212] Anthony Preston. *Book review: Automated Reasoning: 33 Basic Research Problems by Larry Wos* (Prentice Hall 1988). In: ACM SIGART Bulletin 105 (1988), 11p. ISSN 0163-5719. DOI 10.1145/49093.1058124.
  - [213] Dick Price. *Pentium FDIV flaw-lessons learned*. In: IEEE Micro 15.2 (1995), p.86–88.
  - [214] Ganesan Ramalingam. *The Undecidability of Aliasing*. In: ACM Transactions on Programming Languages and Systems 16.5 (Sep. 1994), p.1467–1471. ISSN 0164-0925. DOI 10.1145/186025.186041.
  - [215] Tahina Ramananandro, Dos Gabriel Reis, Xavier Leroy. *A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management*. In: 39th symp. Principles of Programming Languages. ACM Press, 2012, p.521–532. DOI 10.1145/2103656.2103718.

- [216] Tahina Ramananandro, Dos Gabriel Reis, Xavier Leroy. *Formal Verification of Object Layout for C++ Multiple Inheritance*. In: 38th symp. on Principles of Programming Languages. ACM Press, 2011, p.67–79. DOI 10.1145/1926385.1926395.
- [217] John H. Reif. *Code Motion*. In: SIAM Journal on Computing 9.2 (1980), p.375–395.
- [218] Greg Restall. *Introduction to Substructural Logic*. Routledge Publishing, 2000, 396p. ISBN 041521534X.
- [219] Greg Restall. *On Logics Without Contraction*. PhD thesis . . . Department of Philosophy, University of Queensland, Australia, 1994, 292p.
- [220] Bernhard Reus. *Class-Based versus Object-Based: A Denotational Comparison*. In: Algebraic Methodology and Software Technology 2422/2002 (2002), p.45–88. DOI 10.1007/3-540-45719-4.
- [221] Bernhard Reus, Jan Schwinghammer. *Separation Logic for Higher-Order Store*. In: Workshop on Computer Science Logic. 2006, p.575–590. DOI 10.1007/11874683\_38.
- [222] Bernhard Reus, Thomas Streicher. *About Hoare Logics for Higher-Order Store*. In: Intl. Colloquium on Automata, Languages, and Programming. Lecture Notes in Computer Science 3580/2005 (2005), p.1337–1348. DOI 10.1007/11523468\_108.
- [223] John C. Reynolds. *An Introduction to Separation Logic*. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2009.  
<http://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr>.
- [224] John C. Reynolds. *Separation Logic: A Logic for Shared Mutable Data Structures*. In: Proc. of the 17th Annual IEEE symp. on Logic in Computer Science. Washington, DC, USA: IEEE Computer Society, 2002, p.55–74. DOI 10.1109/LICS.2002.1029817.
- [225] Laurence Rideau, Bernard P. Serpette, Xavier Leroy. *Tilting at Windmills with Coq: Formal Verification of a Compilation Algorithm for Parallel Moves*. In: Journal of Automated Reasoning 40.4 (2008), p.307–326. DOI 10.1007/s10817-007-9096-8.
- [226] Silvain Rideau, Xavier Leroy. *Validating Register Allocation and Spilling*. In: Compiler Construction. T. 6011. Lecture Notes in Computer Science. Springer, 2010, p.224–243. DOI 10.1007/978-3-642-11970-5\_13.
- [227] Dirk Riehle. *JUnit 3.8 documented using collaborations*. In: SIGSOFT Software Engineering Notes 33.2 (2008), p.1–28.
- [228] *ROSE compiler Infrastructure project*. Lawrence Livermore National Laboratory, California, USA.  
<http://rosecompiler.org>.

- 
- [229] James Rumbaugh, *Object-Oriented Modeling and Design*. Upper Saddle River, New Jersey, USA: Prentice-Hall, Inc., 1991, 528p. ISBN 0-13-629841-9.
  - [230] Jan J. M. M. Rutten. *Elements of Generalized Ultrametric Domain Theory*. In: Theoretical Computer Science 170.1-2 (1996), p.349–381. DOI 10.1016/S0304-3975(96)80711-0.
  - [231] Vladimir O. Safonov. *Trustworthy Compilers*. Wiley Publishing, 2010, 296p.
  - [232] Mooly Sagiv, Thomas Reps, Reinhard Wilhelm. *Parametric Shape Analysis via 3-valued Logic*. In: ACM Transactions on Programming Languages and Systems 24.3 (2002), p.217–298. ISSN 0164-0925. DOI 10.1145/514188.514190.
  - [233] Bernhard Scholz, Johann Blieberger, Thomas Fahringer. *Symbolic Pointer Analysis for Detecting Memory Leaks*. In: SIGPLAN Notes 34.11 (1999), p.104–113. DOI 10.1145/328691.328704.
  - [234] Herbert Schorr, William M. Waite. *An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures*. In: Communications of the ACM 10.8 (1967), p.501–506. DOI 10.1145/363534.363554.
  - [235] Jan Schwinghammer, *Nested Hoare Triples and Frame Rules for Higher-Order Store*. In: Intl. Workshop on Computer Science Logic. 2009, p.440–454. DOI 10.1007/978-3-642-04027-6\_32.
  - [236] Dana Scott. *Data Types as Lattices*. In: SIAM Journal on Computing 5.3 (May 1976), p.522–587.
  - [237] Helmut Seidl, Reinhard Wilhelm, Sebastian Hack. *Compiler Design — Analysis and Transformation*. Saarbrücken, Munich, Germany: Springer, Nov. 2011, p.177. ISBN 978-3-642-17547-3.
  - [238] Ravi Sethi. *Complete Register Allocation Problems*. In: SIAM Journal on Computing (1975), p.226–248.
  - [239] Joseph Sifakis. *A Framework for Component-based Construction*. In: Software Engineering and Formal Methods (2005), p.293–299. DOI 10.1109/SEFM.2005.3.
  - [240] Prokash Sinha. *A Memory-Efficient Doubly Linked List*, online, version from 18.11.2014. <http://www.linuxjournal.com/article/6828>.
  - [241] Daniel Dominic Sleator, Robert Endre Tarjan. *Self-adjusting Heaps*. In: SIAM Journal on Computing 15.1 (1986), p.52–69.
  - [242] Amitabh Srivastava, David W. Wall. *A Practical System for Intermodule Code Optimization at Link-time*. Technical Report no.WRL92-6. Digital Western Research Laboratory, Palo Alto, USA, Dec. 1992.



- 
- [243] Intl. Organization of Standardization. *ISO C++ Standard*, No.4296 from 2014-11-19.  
<https://isocpp.org/std/the-standard>
  - [244] Ryan Stansifer. *Presburger's Article on Integer Airthmetic: Remarks and Translation*. Technical Report no.TR84-639. Computer Science Department, Cornell University, Sep. 1984.  
<http://techreports.library.cornell.edu:8081/Dienst/UI/1.0/Display/cul.cs/TR84-639>
  - [245] *Static Single Assignment Book*, latest from Jul. 2014.  
<http://ssabook.gforge.inria.fr/latest/book.pdf>.
  - [246] Bjarne Steensgaard. *Points-to Analysis in Almost Linear Time*. In: Proc. of the 23rd ACM SIGPLAN-SIGACT symp. on Principles of Programming Languages. St. Petersburg Beach, Florida, USA: ACM, Jan. 1996, p.32–41. ISBN 0-89791-769-3. DOI 10.1145/237721.237727.
  - [247] Joachim Steinbach. *Termination of Rewriting — Extensions, Comparison and Automatic Generation of Simplification Orderings*. PhD thesis. . . . Universität Kaiserslautern, Germany, 1994, 288p.
  - [248] Leon Sterling, Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. 2nd. Cambridge, MA, USA: MIT Press, 1994, 549p. ISBN 0-262-19338-8.
  - [249] Norihisa Suzuki. *Analysis of Pointer Rotation*. In: Communications of the ACM 25.5 (1982), p.330–335. DOI 10.1145/358506.358513.
  - [250] Robert Daniel Tennent. *The Denotational Semantics of Programming Languages*. In: Communications of the ACM 19.8 (Aug. 1976), p.437–453. ISSN 0001-0782. DOI 10.1145/360303.360308.
  - [251] *The Clang Project*, University of Illinois, USA.  
<http://clang.llvm.org>.
  - [252] *The GNU Compiler Collection*.  
<http://gcc.gnu.org>.
  - [253] *The LLVM Project*, University of Illinois, USA.  
<http://llvm.org>.
  - [254] *The Valgrind Project*.  
<http://www.valgrind.org>.
  - [255] Simon Thompson. *Haskell: The Craft of Functional Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, 528p. ISBN 0-201-40357-9.
  - [256] Simon Thompson. *Type Theory and Functional Programming*. Intl. Computer Science Series. Wokingham, England: Addison-Wesley, 1991, 388p. ISBN 0-201-41667-0.

- 
- [257] Mads Tofte, Jean-Pierre Talpin. *Implementation of the Typed Call-by-Value  $\lambda$ -Calculus using a Stack of Regions*. In: Proc. of the 21st ACM SIGPLAN-SIGACT symp. on Principles of Programming Languages. POPL'94. Portland, Oregon, USA: ACM, 1994, p.188–201. DOI 10.1145/174675.177855.
  - [258] Mads Tofte, Jean-Pierre Talpin. *Region-based Memory Management*. In: Information and Computation 132.2 (1997), p.109–176.
  - [259] Paolo Tonella. *Concept Analysis for Module Restructuring*. In: IEEE Transactions on Software Engineering 27.4 (2001), p.351–363. DOI 10.1109/32.917524.
  - [260] Jean-Baptiste Tristan, Xavier Leroy. *Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations*. In: 35th symp. Principles of Programming Languages. ACM Press, 2008, p.17–27.
  - [261] Jean-Baptiste Tristan, Xavier Leroy. *Verified Validation of Lazy Code Motion*. In: Programming Language Design and Implementation. ACM Press, 2009, p.316–326.
  - [262] Anne Sjerp Troelstra, Helmut Schwichtenberg. *Basic Proof Theory*. 2nd. New-York, USA: Cambridge University Press, 2000, 417p. ISBN 0-521-77911-1.
  - [263] Stanford University. *Gottlob Frege*. Stanford Encyclopedia of Philosophy.  
<http://plato.stanford.edu>.
  - [264] Robert A. Wagner, Michael J. Fischer. *The String-to-String Correction Problem*. In: Journal of the ACM 21.1 (Jan. 1974), p.168–173. ISSN 0004-5411. DOI 10.1145/321796.321811.
  - [265] Mitchell Wand. *A New Incompleteness Result for Hoare's System*. In: Proc. of 8th ACM symp. on Theory of Computing. 1976, p.87–91. DOI 10.1145/800113. 803635.
  - [266] David H.D. Warren. *Applied Logic — Its Use and Implementation as a Programming Tool* (also passed as PhD-thesis same year to Edinburgh University, Scotland). Technical Report. no.290. Menlo Park, California, USA: SRI International, Jun. 1983.
  - [267] David Scott Warren. *Efficient Prolog Memory Management for Flexible Control Strategies*. In: Intl. symp. on Logic Programming, New Generation Computing (reprint). T. 2. 4. Atlantic City, New York, USA, Dec. 1984, p.361–369.
  - [268] David Scott Warren. *Programming in Tabled Prolog (Draft)*, Dpt. of Computer Science, Stony Brook, New York, USA, from 31th July 1999.  
<http://www3.cs.stonybrook.edu/warren/xsbbook/book.html>.
  - [269] William E. Weihl. *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables*. In: Proc. of the 7th ACM SIGPLAN-SIGACT symp. on Principles

- of Programming Languages. eds. Paul W. Abrahams, Richard J. Lipton, Stephen R. Bourne. ACM Press, 1980, p.83–94. ISBN 0-89791-011-7.
- [270] Georg Weissenbacher. *Abstrakte Kunst: Fehler finden durch Model Checker*. In: Magazin für professionelle Informationstechnik, iX 5 (2004), p.116–118.  
[http://www.heise.de/artikel-archiv/ix/2004/05/116\\_Abstrakte-Kunst](http://www.heise.de/artikel-archiv/ix/2004/05/116_Abstrakte-Kunst)
- [271] Georg Weissenbacher. *Ohne Beweis – VDM++ Lightweight Formal Methods*. In: Magazin für professionelle Informationstechnik, iX 3 (2001), p.157–161.  
[http://www.heise.de/artikel-archiv/ix/2001/03/157\\_Ohne-Beweis](http://www.heise.de/artikel-archiv/ix/2001/03/157_Ohne-Beweis)
- [272] *Why3 Platform for Verification Systems, Project*.  
<http://why3.lri.fr>.
- [273] Wikipedia. *Funarg Problem*.  
[http://en.wikipedia.org/wiki/Funarg\\_problem](http://en.wikipedia.org/wiki/Funarg_problem).
- [274] Wikipedia. *Region-based Memory Management*.  
[https://en.wikipedia.org/wiki/Region-based\\_memory\\_management](https://en.wikipedia.org/wiki/Region-based_memory_management).
- [275] Wikipedia. *Shape Analysis*.  
[https://en.wikipedia.org/wiki/Shape\\_analysis\\_\(program\\_analysis\)](https://en.wikipedia.org/wiki/Shape_analysis_(program_analysis)).
- [276] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, Massachusetts, USA: MIT Press, Apr. 1993, 384p. ISBN 0-262-23169-7.
- [277] Paul Tucker Withington. *How Real is "Real-Time" GC?* In: Proc. of 6th annual Object-Oriented Programming Systems, Languages and Applications Workshop: Garbage Collection in Object-Oriented Systems. Phoenix, Arizona, USA, Oct. 1991, p.1–8.
- [278] Stephen Wolfram. *A New Kind of Science*. Champaign, Illinois, US, United States: Wolfram Media Inc., 2002, 1197p. ISBN 1-57955-008-8.
- [279] Larry Wos. *Automated Reasoning: 33 BASIC Research Problems*. Prentice-Hall, Inc., 1988, 319p. ISBN 0-13-054552-X.
- [280] Larry Wos. *The Problem of Finding a Restriction Strategy more effective than the Set of Support Strategy*. In: Journal of Automated Reasoning 7.1 (1991), p.105–107.
- [281] Hongseok Yang, Peter W. O’Hearn. *A Semantic Basis for Local Reasoning*. In: Foundations of Software Science and Computation Structures. 2002, p.402–416.

- [282] Ilya S. Zakharov, *Configurable Toolset for Static Verification of Operating Systems Kernel Modules*. In: Programming and Computer Software. T. 41. Proc. of the Institute for System Programming of Russian Academy of Science. Pleiades Publishing, 2015, p.49–64.
- [283] Karen Zee, Viktor Kuncak, Martin C. Rinard. *Full Functional Verification of Linked Data Structures*. In: Programming Language Design and Implementation. 2008, p.349–361. DOI 10.1145/1375581.1375624.
- [284] Kaizhong Zhang, Dennis Shasha. *Simple Fast Algorithms for the Editing Distance between Trees and Related Problems*. In: SIAM Journal on Computing 18.6 (1989), p.1245–1262.
- [285] Кулик, Б.А. *Логика естественных рассуждений* / В.А. Дюк – Санкт-Петербург. Невский Диалект, 2001. – 128с. ISBN 5-7940-0080-5
- [286] Калинина, Т. В. *Абстрактные методы повышения эффективности логического вывода*: дис. ... канд. физ.-мат.наук:05.13.18/ Калинина Татьяна Владимировна. — Санкт-Петербург, 2001. — 185с.
- [287] Левенштейн, В.И. *Двоичные коды с исправлением выпадений, вставок, замещений символов* / В.И. Левенштейн // Докл. АН СССР. – 1965. №4(163). С.845–848.
- [288] Братчиков, И. Л. *Формально-грамматическая интерпретация метода резолюции* / Братчиков Игорь Леонидович // Вестник СПбГУ, труды XXIV-ых научн. конференции процесса управления, устойчивости, Санкт-Петербург. – 1998. №1. p.197-202.
- [289] Верт, Т., Крикуна, Т., Глухих, М. *Обнаружение дефектов работы с указателями в программах на языках Си, Си++ с использованием статического анализа, логического вывода* / Верт Татьяна // Инструменты, методы анализа программ (ТМРА 2013), Кострома. – 2013.
- [290] Лавров, С.С. *Программирование – Математические основы, средства, теория*. / С. Лавров // Санкт-Петербург: БХВ-Петербург, 2001 - 320с. ISBN 5-94157-069-4
- [291] Опалева, Э.А. Самойленко В.П. *Языки Программирование, Методы Трансляции*. / Э.А. Опалева // Санкт-Петербург: БХВ-Петербург, 2005 - 480с. ISBN 5-94157-327-8
- [292] René Haberland, Igor L. Bratchikov. *Transformation of XML Documents with Prolog*. In: Advances in Methods of Information and Communication Technology. Vol. 10. 2008, p.99–111. ISBN 975-5-8021-1020-1, УДК 519.1+681.3.
- [293] René Haberland. *Using Prolog for Transforming XML-Documents*. arXiv:1912.10817 [cs.PL], 2007/2019.

- [294] René Haberland, Kirill Krinkin. *A Non-repetitive Logic for Verification of Dynamic Memory with Explicit Heap Conjunction and Disjunction*. In: International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP). Ed. University of Venice, Italy. Oct. 2016, p.1–9. ISBN 978-1-61208-506-7, ISSN 2308–4498, IARIA XPS Press/ThinkMind.
- [295] René Haberland, Kirill Krinkin, Sergey Ivanovskiy. *Abstract Predicate Entailment over Points-To Heaplets is Syntax Recognition*. In: 18th Conference of Open Innovations (FRUCT), ISSN 2305-7254, ISBN 978-952-68397-3-8. Ed. By T. Tyutina S., Balandin A., Levina. 2016, p.66–74. DOI 10.1109/FRUCT-ISPIT.2016.7561510.
- [296] René Haberland. *A Stricter Heap Separating Points-To Logic*. In: 3rd Intl. Scientific Symposium Sense Enable (SPITSE2016). Ed. by Russia National Research University Moscow. June 2016, p.103–104. ПИИЦ 26444969.
- [297] René Haberland, Sergey Ivanovskiy. *Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog*. In: Proc. of the 8th Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE 2014). Ed. by Alexander Kamkin, Alexander Petrenko, and Andrey Terekhov. 2014, p.46–50. ISBN 978-5-91474-020-4, ISSN 2311–7230, DOI 10.15514/SYRCOSE-2014-8-7.
- [298] René Haberland. *Unification of Template-Expansion and XML-Validation*. In: Proc. of Intl. Conf. on Control Processes and Stability. Vol. 34. Saint Petersburg State University, 2008, p.389–394. ISBN 978-5-288-04680-3, ISSN 2313–7304, ПИИЦ, УДК 517.51:517.9:518.9.
- [299] René Haberland. Narrowing Down XML Template Expansion and Schema Validation, arXiv:1912.10816 [cs.PL], 2007/2019.
- [300] Хаберланд, Р. *Верификация корректности динамической памяти с помощью логического языка программирования*. / Р. Хаберланд // Конференция  $EMC^2$  «Технологии Майкрософт в Теории, на Практике Программирования. Серия: Новые подходы к разработке программного обеспечения», Санкт-Петербургский Политехнический Университет, Санкт-Петербург. – 2014. №3. С.56–57.
- [301] Хаберланд, Р. *Расширяемый Фреймворк для верификации статически типизированных объектно-ориентированных программ, использующих динамическую память*. / Р. Хаберланд // Санкт-Петербургский Электротехнический Университет «ЛЭТИ» (ППС ЛЭТИ), Санкт-Петербург. – дек. 2015. №5. УДК 004.052.2.
- [302] Хаберланд, Р., Ивановский, С. А., Кринкин, К. В. *Верификация Объектно-ориентированных программ с динамической памятью на основе ссылочной модели с помощью Пролога*. / Р. Ха-

- берланд // Известия ЛЭТИ, Санкт-Петербургский Электротехнический Университет, Санкт-Петербург. – 2016. №1. С.14–18. ISSN 2017–8985, УДК 004.052.2.
- [303] Хаберланд, Р. *Сравнительный анализ статических методов верификации динамической памяти.* / Р. Хаберланд // Компьютерные инструменты в образовании, Санкт-Петербург. – 2019. №2. С.5–30. DOI 10.32603/2071-2340-2019-2-5-30, УДК 004.052.2.
- [304] Хаберланд, Р., Кринкин К. В., *Программа для быстрого построения проектов программного обеспечения (Builder), инструментарий для верификации динамической памяти.* Санкт-Петербургский государственный электротехнический университет (ЛЭТИ) — ПрЭВМ в Роспатенте, 18.12.2018г. №2019610438
- [305] Хаберланд, Р., Кринкин К. В., *Программа для динамического сокращения (слайсинга) программы (Shrinker), инструментарий для верификации динамической памяти.* Санкт-Петербургский государственный электротехнический университет (ЛЭТИ) — ПрЭВМ в Роспатенте, 18.12.2018г. №2018664644
- [306] Хаберланд, Р., Кринкин К. В., *Программа для верификации динамической памяти (ProLogika), инструментарий для верификации динамической памяти.* Санкт-Петербургский государственный электротехнический университет (ЛЭТИ). — ПрЭВМ в Роспатенте, 18.12.2018г. №2019610339
- [307] Хаберланд, Р., Кринкин К. В., *Программа для преобразования XML-документов с помощью логического программирования (Prolog-XML).* Санкт-Петербургский государственный электротехнический университет (ЛЭТИ). — ПрЭВМ в Роспатенте, 27.11.2019г. №2019666260
- [308] René Haberland. *Recent Techniques Review on Heap Verification with Class Objects.* St. Petersburg Electrotechnical University (unpublished), 2016
- [309] René Haberland. *Tutorials and Examples.* St. Petersburg Electrotechnical University, 2016. <https://bitbucket.org/reneH123/tutorials>.

# Список определений

1.1	Определение – Тройка Хора . . . . .	13
1.2	Определение – Входной язык программирования . . . . .	17
1.3	Определение – Язык спецификации . . . . .	18
1.4	Определение – Логическое следствие . . . . .	18
1.5	Определение – Доказательство как поиск . . . . .	19
1.6	Определение – Корректность вычисления Хора . . . . .	19
1.7	Определение – Полнота вычисления Хора . . . . .	22
1.8	Определение – Логическая формула предикатов первого порядка . . . . .	26
1.9	Определение – Термы $T_{\lambda 2}$ . . . . .	26
1.10	Определение – Множество термов $\Lambda_{T_{\lambda 2}}$ . . . . .	26
1.11	Определение – Проверка типа . . . . .	27
1.12	Определение – Формальное доказательство . . . . .	29
1.13	Наблюдение – Модель вычисления верификации . . . . .	30
1.14	Наблюдение – Фаза проверки типов . . . . .	37
1.15	Наблюдение – Нередуцируемость верификации к типизации . . . . .	38
3.1	Наблюдение – Организованная память и свежий контекст . . . . .	81
3.2	Наблюдение – Неорганизованная память и единый контекст . . . . .	81
3.3	Наблюдение – Феномен далёкой манипуляции . . . . .	82
3.4	Наблюдение – Интервал видимости переменных . . . . .	82
3.5	Наблюдение – Графовое представление динамической памяти . . . . .	86
3.6	Теорема – Свойства динамических ячеек по Рейнольдсу . . . . .	90
3.7	Определение – Соотношение выполнимости интерпретаций куч . . . . .	94
3.8	Определение – Конечный граф кучи . . . . .	95
3.9	Определение – Терм кучи . . . . .	95
3.10	Определение – Расширение термов куч . . . . .	95
4.1	Определение – Генеричный терм в Прологе . . . . .	99
4.2	Определение – Правило Хорна . . . . .	100
4.3	Определение – Запрос в Прологе . . . . .	102

4.4	Определение – Отсечение решений . . . . .	104
4.5	Тезис – Доказательство куч равно синтаксическому перебору . . . . .	109
4.6	Определение – Двойная семантика предикатов Пролога . . . . .	110
4.7	Наблюдение – Равенство о единицах доказательств . . . . .	111
4.8	Наблюдение – Дедукция с возвратом в доказательствах . . . . .	113
4.9	Наблюдение – Стековая система вызовов . . . . .	114
4.10	Тезис – Выразимость реляций в Прологе . . . . .	116
4.11	Тезис – Упрощение с помощью термового IR . . . . .	119
4.12	Наблюдение – Сравнение декларативных парадигм . . . . .	121
4.13	Наблюдение – Упрощение утверждений равно обобщению . . . . .	121
4.14	Заключение – Минимизация разницы между языками . . . . .	122
4.15	Наблюдение – Сходство языков при верификации . . . . .	123
4.16	Заключение – Минимизация входной программы . . . . .	129
5.1	Наблюдение – Перегрузка оператора . . . . .	133
5.2	Тезис – Ужесточение выразимости . . . . .	133
5.3	Тезис – Упрощение с помощью высчитывания куч . . . . .	134
5.4	Тезис – Неполнота для улучшения полноты . . . . .	134
5.5	Заключение – Ужесточение в моделировании . . . . .	134
5.6	Определение – Выводимая куча по Рейнольдсу . . . . .	137
5.7	Определение – Конъюнкция куч . . . . .	139
5.8	Теорема – Конъюнкция обобщённых куч . . . . .	141
5.9	Соглашение – Локация . . . . .	141
5.10	Лемма – Моноид конъюнкции . . . . .	142
5.11	Теорема – Абелевская группа конъюнкции . . . . .	142
5.12	Соглашение – Обратимость кучи . . . . .	144
5.13	Лемма – Гомоморфизм об обыкновенных кучах . . . . .	145
5.14	Определение – Дизъюнкция кучи . . . . .	146
5.15	Теорема – Моноид дизъюнкции . . . . .	146
5.16	Теорема – Дистрибутивность . . . . .	147
5.17	Соглашение – Моделирование объектных экземпляров . . . . .	149
5.18	Соглашение – Объектные поля . . . . .	151
5.19	Определение – Неполные предикаты . . . . .	152
5.20	Пример – Неполный предикат №.1 . . . . .	152
5.21	Пример – Неполный предикат №.2 . . . . .	153
6.1	Определение – Утверждение о куче . . . . .	158



6.2	Закключение – Корректность кучи . . . . .	160
6.3	Определение – Неформальный граф кучи . . . . .	160
6.4	Определение – Предикатное правило . . . . .	162
6.5	Определение – Набор предиката . . . . .	163
6.6	Закключение – Предикатная среда . . . . .	164
6.7	Лемма – Полнота предикатов . . . . .	164
6.8	Лемма – Предикаты высшего порядка . . . . .	164
6.9	Определение – Свёртывание предиката . . . . .	166
6.10	Наблюдение – Сходство с формальными языками . . . . .	167
6.11	Тезис – Распознавание как доказательство . . . . .	167
6.12	Закключение – Контекст-свободность выражений куч . . . . .	167
6.13	Наблюдение – Редукция куч . . . . .	167
6.14	Лемма – Куча как слово . . . . .	168
6.15	Определение – Преобразование в атрибутируемую грамматику . . . . .	169
6.16	Определение – Обратимость преобразования . . . . .	169
6.17	Закключение – Приостановка преобразований . . . . .	169
6.18	Закключение – Корректность и полнота преобразований . . . . .	170
6.19	Определение – Абстрактное предложение . . . . .	171
6.20	Определение – Множество началов нетерминалов . . . . .	172
6.21	Определение – Множество последующих терминалов . . . . .	172
6.22	Пример – Многозначимость правил . . . . .	173
6.23	Пример – Корректность . . . . .	173

# Список иллюстраций

1.1	Качественная лестница по критериям важности . . . . .	10
1.2	Логические правила вычисления Хора . . . . .	12
1.3	Неполный список правил для верификации . . . . .	14
1.4	Правила циклов заменившие while . . . . .	15
1.5	Пример кода остатка при деление целых чисел . . . . .	16
1.6	Пример отрицательного логического вывода . . . . .	18
1.7	Теорема Чёрча-Россера применена к тройкам Хора . . . . .	20
1.8	Пример полного и корректного набора правил из [67] . . . . .	25
1.9	Теорема Пирса об исключённого третьего в системе « <i>Coq</i> » . . . . .	28
1.10	Индуктивное определение чисел с помощью термов Чёрча . . . . .	29
1.11	Блочная схема с присвоенными значениями . . . . .	32
1.12	Фазы генерации кода . . . . .	36
1.13	Сравнение проблем между проверкой типов и вычислением Хора . . . . .	37
1.14	Виды объектных вычислений . . . . .	40
1.15	Указатель $x$ ссылается на объект, чьё содержимое ссылается на $y$ . . . . .	43
1.16	Специфицируемые этапы объекта . . . . .	44
1.17	Определение объектно-термовых выражений по Абади-Лейно . . . . .	46
1.18	Пример набора правил, которые некорректны . . . . .	47
1.19	Упрощённые объектные по Абади-Лейно . . . . .	48
1.20	Подтип класса . . . . .	48
1.21	Дерева вывода для примера объектного вида программы . . . . .	49
1.22	Пример набора правил для объектного вида объектного вычисления . . . . .	50
1.23	Пример кода ротации указателей по Сузуки . . . . .	56
1.24	Динамическая память при запуске программы ротации указателей . . . . .	57
2.1	Типичное распределение процесса в оперативной памяти . . . . .	67
2.2	Пример стека . . . . .	68
2.3	Пример кучи . . . . .	68
2.4	Пример программы инстанциации объектного экземпляра . . . . .	72

2.5	Пример неверного присвоения объектной ссылки . . . . .	73
2.6	Примеры распределения битовых масок . . . . .	74
2.7	Пример нетерминации кода . . . . .	74
3.1	Пример графа потока данных. . . . .	79
3.2	Пример SSA-присваивания по блокам . . . . .	80
3.3	Видимость локальных переменных в а),b) и динамической в с) . . . . .	82
3.4	Пример конечного автомата $A_1$ . . . . .	83
3.5	Пример конечного автомата $A_2$ . . . . .	84
3.6	Пример конечного автомата $A_3$ . . . . .	84
3.7	Равенства описывающие автомат . . . . .	85
3.8	Пример код Си программа для инверсии списков . . . . .	85
3.9	Пример состояние динамической при выполнение программы . . . . .	86
3.10	Пример кучи с указателями $x, u, y$ . . . . .	92
3.11	Пример кактуса динамической памяти . . . . .	92
3.12	Пример схематической делимости динамической памяти . . . . .	94
3.13	Формальное определение кучи по ЛРП . . . . .	94
4.1	Факты и правила в Прологе на примере предиката Аккерманна . . . . .	101
4.2	Пример кода унификации с проверкой рекурсивного повтора . . . . .	103
4.3	Пример кода унификации списков . . . . .	103
4.4	Дерево вывода для предиката Аккерманна . . . . .	104
4.5	Сопоставления для дерева вывода из рисунка 4.4 . . . . .	105
4.6	Пример код факториал на Прологе . . . . .	105
4.7	Контр-пример код факториал на Прологе . . . . .	106
4.8	Характеристики типизации из пунктов 2,3 . . . . .	110
4.9	Вызов подцелей . . . . .	111
4.10	Пример стековых окон при вызове подцелей . . . . .	111
4.11	Реляционная модель применена к предикатам Пролога . . . . .	117
4.12	Архитектура конвейера верификатора и статических анализаторов . . . . .	120
4.13	Мета-паттерн MVC применена к процессу верификации . . . . .	123
4.14	Количественный анализ при использовании метрик Холстедта [112] . . . . .	124
4.15	Архитектура верификатора динамической памяти . . . . .	128
5.1	Изоморфизмы смежных куч с объектами . . . . .	136
5.2	Пример связанного графа кучи . . . . .	136
5.3	Пример разбитого на две части графа кучи . . . . .	137

5.4	Пример возможного разбиения графа кучи на отдельные части . . . . .	137
5.5	Граф кучи до и после конъюнкции . . . . .	140
5.6	Граф кучи до и после инверсии . . . . .	145
5.7	Упорядоченное множество над графами куч . . . . .	148
5.8	Формы присвоения объектных экземпляров . . . . .	149
5.9	Преобразование схематического графа кучи . . . . .	150
6.1	Пример сложных куч . . . . .	159
6.2	Расширенная форма РФБН прологовских правил . . . . .	162
6.3	Функционал <code>map/3</code> . . . . .	165
6.4	Пример конфигурации кучи . . . . .	174
6.5	Графическая оболочка . . . . .	177
6.6	Перспектива программиста (UML Use Case) . . . . .	178
6.7	Перспектива спецификации программы (UML Use Case) . . . . .	179
6.8	Перспектива верификации (UML Use Case) . . . . .	179
6.9	Пример линейного списка №1 . . . . .	180
6.10	Пример линейного списка №2 . . . . .	180
6.11	Архитектура верификатора . . . . .	182
6.12	Компоненты верификатора . . . . .	183
6.13	Пакет <code>alice.tuProlog</code> . . . . .	183
6.14	Пакет <code>parsers</code> . . . . .	184
6.15	Пакет <code>core</code> . . . . .	185
6.16	Пакет <code>internal.checkers</code> . . . . .	185
6.17	Пакет <code>internal.aps</code> . . . . .	186
6.18	Класс <code>Main</code> . . . . .	187
6.19	Класс <code>Parser</code> . . . . .	188
6.20	Класс <code>SyntaxAnalyzer</code> . . . . .	188
6.21	Пакет <code>frontend</code> . . . . .	189
6.22	Пакет <code>parsers.incoming</code> . . . . .	189
6.23	Пакет <code>parsers</code> . . . . .	190
6.24	Пакет <code>frontend</code> . . . . .	191
6.25	Пакет <code>internal.core</code> . . . . .	192
6.26	Слои пакетов <code>frontend</code> , <code>internal.core</code> , <code>internal.aps</code> . . . . .	193
6.27	Пакет <code>internal.aps</code> . . . . .	194
6.28	Пакет <code>internal.ht</code> . . . . .	195
6.29	Пакет <code>Prolog</code> . . . . .	196

## Приложение: Предметный указатель

- $>:$ , 130
- $\alpha$ -преобразование, 48, 79, 91
- $\beta$ -преобразование, 91
- $\lambda$ -терм, 23, 42, 52, 124
- $\lambda$ -вычисление, 16, 26, 37, 42, 91, 125, 128
- $\lambda$ -выражение, 25
- $\mapsto$ -утверждение, 172, 177
- $\mu$ -рекурсивная схема, 45, 154
- $\mu$ -рекурсивный предикат, 166
- $\phi$ -функция, 80
- $\star$ , 91, 93, 157, 167
- $\cdot$ -оператор, 93
- $\cdot$ -оператор, 92, 100, 131, 141
- $\cdot$ ctor, 43
- $\cdot$ dtor, 43
- $=..$ , 102
- Абельская группа, 142
- Аристотель, 89, 100
- Гегель, 89
- Хаскель, 21, 22
- Канторово множество, 116
- Картезианский продукт, 117
- Платон, 89, 100
- Пролог, 99, 100, 102, 106–109, 111–116, 118–123, 126, 127, 129–131, 153, 154, 158, 159, 169
  - дизъюнкция, 159
  - запрос, 96, 102, 159
  - Definite Clause Grammars, 158
- Прологовская теория, 112
- Прологовское правило, 109, 126, 169
- Тьюринг-вычислимость, 44, 107
- В-дерево, 115
- абдукция, 12, 55, 63, 126
- абсорбция, 148
- абстракция, 24, 29, 50, 51, 75, 92, 109, 156
  - графа, 55
- абстрактная интерпретация, 31, 32, 126
- абстрактная машина, 120, 194
- абстрактный автомат, 20
- абстрактный предикат, 64, 97, 98, 118, 128, 131, 135, 141, 153–157, 160, 174
- абстрактный тип данных, 35, 40, 50, 53
- абстрактное предложение, 173
- адресное пространство, 92, 137
- аксиома, 12, 70
- аксиоматическая семантика, 124
- аксиомы Пиано, 154
- актёр, 41, 51
- алгебраическое поле, 147
- алгебраическое выражение, 161
- алгоритм Кнута-Бендикса, 32
- алгоритм Левенштейна, 195
- алгоритм Рейнольдса, 77
- алгоритм Уэйт-Шора, 78
- анализ образов, 10, 54, 86
- анализ псевдонимов, 11, 58, 59, 69, 77, 128

- анализ зависимости данных, 36, 77
- анализатор Эрли, 171
- анонимная функция, 46, 49, 97
- антецедент, 12, 62
- аппроксимация, 32, 55, 120
- архитектура ЭВМ, 41
- арифметическое выражение, 161
- арифметика, 133, 154
- арифметика Пиано, 32
- арифметика Пресбургера, 32
- арифметика термов по Чёрчу, 96, 100
- арность, 100, 102, 112, 114, 125, 154
- ассемблер, 43, 120
- ассистент верификации, 25
- ассоциативность, 117, 141, 146
- атом, 125, 176
- атомизм, 41, 124
- атомные утверждения, 97
- атрибут, 150
- атрибутируемая грамматика, 169, 194
- автомат, частичных производимых, 158
- автомат, конечный, 83, 118, 158, 195
- автоматизация доказательства, 109, 126–128, 132
- автоматизация тестов, 34
- автоматизация верификации, 25, 194
- байт, 68
- база знаний, 99, 101, 158
- базисный случай, 105, 106
- бесконечная структура данных, 22, 23, 115, 129
- безопасные операции указателей, 76
- быстрая прототипизация, 155
- быстродействие, 66, 69, 114, 115
- биграф, 138
- бинарный оператор, 133, 138
- битовое поле, 81
- битовой вектор, 58
- блок, 79, 127
- блок-схема, 14, 32, 34
- блокировка нити, 51
- булево значение, 23, 26, 34, 62, 94, 107, 159, 205
- целые числа, 97, 105, 133, 137
- частичная корректность, 23
- частичная спецификация, 196
- человеческий фактор, 35
- далёкая манипуляция, 82
- дедукция, 12, 13, 77, 113, 127
- декларативная парадигма, 13, 14, 18, 104, 110, 113, 115, 121, 123, 124, 161
- делегация, 43
- деление, 16
- деление ответственности, 51
- денотационная семантика, 45, 81, 94, 124, 130
- дерево вывода, 18, 49, 104, 105, 109, 110, 113
- деструктор, 66
- детерминизация, 106
- диаграмма Хассе, 148
- диапазон видимости, 17, 54, 66, 79, 100, 121, 127, 161
- динамическая память, 10, 14, 36, 37, 55, 62, 64, 66, 71, 72, 78, 81–85, 88–92, 95, 107, 113, 120, 121, 124, 126–130, 138, 160
- динамический список, 54
- дистрибутивность, 147
- дизъюнкция, 26, 90, 93, 144, 146, 153
- длина программы, 124
- должны-ссылаться, 69
- домен, 19, 126, 137, 163
- домен как реляция, 117
- доверительный компилятор, 52
- дуальная граф памяти, 151
- дуальная операция, 146

- дубликат, 76, 87, 104, 106, 119
- дважды связанный список, 60
- двоичный интерфейс приложения, 41
- двоичное дерево, 88, 93
- двудольный граф, 138
- двусвязный список, 156
- естественные числа, 29
- экслюзив или, 67
- экстремум, 42
- эвристика, 63, 70, 98
- факт, 100, 106, 112, 113, 122, 161
- факториал, 104
- факторизация правил, 166
- фальсификация, 113
- флэш-память, 69
- форма Бэккуса-Наура, 101, 133, 162
- формальная алгебра, 29
- формальная грамматика, 98, 109, 135, 158, 161, 167, 169, 176
- формальная логика, 19, 29
- формальная теория, 25, 29, 33, 76, 112, 122, 127, 134, 139
- формальная верификация, 13
- формальный метод, 12, 31, 33, 51
- формула кучи, 160
- фрагментация памяти, 149
- фрейм, 62, 63, 94, 128
- фреймворк, 52, 77, 81
- функционал, 20, 24, 38, 54
- функциональная парадигма, 23, 54, 81, 108, 121, 123
- функция Аккерманна, 101, 102, 112, 125
- функтор, 26, 97, 100, 102, 103, 111, 125, 128, 131, 162
- генерация кода, 77, 93, 120, 128
- генерация контр-примера, 22, 30, 53, 125, 127, 195
- геометрия, проективная, 16
- геометрия, вычислительная, 156
- гипотеза, 28
- глобальный инвариант, 50
- гомоморфизм, 144, 153
- граф кучи, 81–83, 86, 88, 91, 95, 97, 122, 129, 132, 135, 141, 142, 144, 146, 160
- граф образов, 55
- граф потока управлений, 23, 51, 58, 79, 127
- граф зависимости, 54, 130
- граф, простой, 141
- грамматика, контекст-свободная, 109
- грань графа, 83–87, 92, 98, 129, 156
- группа, 145, 146
- идиома программирования, 41
- иерархия наследования классов, 43, 130, 149
- императивная парадигма, 13, 79, 80, 108, 115, 161
- импликация, 15, 28
- индекс с записями, 50
- индукция, 12, 22, 91, 141, 154
- индуктивно определённая структура, 25, 98, 105
- индуктивное определение, 22, 29, 107
- инфиксная запись, 101
- инфимум, 41, 43
- инъективность, 32
- интеллектуальный уровень языка, 118, 124
- интерпретация формул, 123, 127, 132
- интра-процедуральный анализ, 55
- интроспекция, 11, 63, 70, 102
- интуиционистское суждение, 27
- инвариант, 16, 21, 24, 38, 54, 55, 117, 142
- инверсия, 144, 146, 148
- инверсия контроля, 63
- инверсия списка, 85

- инверсно-польская запись, 119
- исключения, 24, 111
- исключённый третий, 28, 114
- искусственный интеллект, 107
- истина, 12, 160
- изоморфизм Карри-Хауарда, 157
- изоморфизм графов, 174
- изоморфизм изображений, 32, 97, 109, 126
- кактус, 93
- канонизация, 116, 117, 176
- картеж, 130
- кэш, 60, 117, 154, 173
- класс, 43, 45
- классный экземпляр, 52, 148
- классный вид, 128
- классовый тип, 45
- классовое вычисление, 132
- ключевое слово, 27, 177
- ко-область, 126
- коллаборации, 52
- комбинатор плавающей точки, 16, 47, 106
- компиляция, 130
- комплексные числа, 143
- компонентная алгебра, 52
- конфликт наименований, 15, 16, 164
- конъюнкция, 26, 95, 138, 141–143, 146, 148, 153
- конъюнкция реляций, 116
- конъюнкт, 139, 150, 152
- конкретизация, 15, 151
- конкретизация графа, 55
- коннективизм, 41
- коннотация, 90
- консеквент, 12, 106
- константная функция, 95, 97, 100, 152, 160
- конструктор, 43, 44
- контекст-независимость, 134
- контекст-зависимость, 108, 132
- контр-пример, 22, 125, 134, 168, 195
- конвейер, 120, 127
- конверсия типов, 73
- кообласть, 170
- копирование памяти, 114
- копирующий сборщик мусора, 60
- копия кучи, 156
- корректность, 10, 14, 22, 23, 47, 50, 51, 54, 69, 77, 97, 105, 130, 133
- кортеж, 41, 131
- космос выводимости, 13
- красное отсечение, 106, 154
- критерии качества, 9
- критерий минимальности, 129
- куча, 54, 60, 66–68, 81, 87–91, 93, 95, 96, 98, 111, 112, 115, 117, 119, 121, 122, 130, 132, 133, 138, 139, 146, 148
- динамическая, 109, 133
- интерпретация, 82, 128, 159
- инверсия, 143, 147, 148
- изоморфизм, 135
- конечная, 147
- модальность, 90
- неполная, 134
- независимая, 159
- нормализация, 144, 159
- обобщённая, 144, 151
- обратимая, 143
- отрицательная, 144
- положительная, 143
- простая, 113, 118, 126, 150, 156, 159
- пустая, 143, 147, 152, 156, 159, 173
- разделение, 90, 138
- с символами, 161
- слияние, 138



- сложная, 118, 148, 156, 159  
 спецификация, 135  
 связанная, 74, 90, 97  
 утверждение о, 159  
 высчитывание, 106  
 куча Фибоначчи, 88  
 лексема, 158  
 лексика, 107  
 лексикографический порядок, 126, 130, 149, 167  
 лемма, 29, 37, 127, 130  
 лемма Ардена, 83, 84  
 ленивое вычисление, 21, 26, 30  
 лестница качества, 9, 17, 76  
 лево-рекурсия, 106, 158  
 левое свёртывания, 165  
 лимит, 19, 31, 38, 80, 120  
 линейная адресация, 67, 88  
 линейный список, 22, 23, 76, 77, 85, 87, 92–94,  
 100, 101, 111, 115, 126  
 линукс, 78  
 литерал, 107  
 логическая формула, 26, 161  
 логический оператор, 90  
 логический предикат, 15, 121  
 логический вывод, 12, 112, 113, 121, 154, 157  
 логическое правило, 161  
 логическое программирование, 110  
 логическое суждение, 12  
 логика, 29  
 логика предикатов, 26, 32, 42, 135, 156  
 логика распределенной памяти, 194  
 логика распределённой памяти, 62, 63, 90, 93,  
 132, 139  
 логика утверждений, 91  
 логика высшего порядка, 23, 25, 123, 164  
 локация, 95, 141, 144, 164  
 локализация ошибочного кода, 33, 44, 76  
 локализатор, 156, 159  
 локальность, 98, 104, 109, 138, 148, 194  
 локальность спецификации, 53  
 ложь, 160  
 массив, 41, 62, 66, 69, 150  
 машина Тьюринга, 14  
 машина Уоррена, 102, 107, 114, 153  
 мемоизатор, 153, 154, 173  
 метафизика, 29  
 метод, 51  
 метод Кусо, 31  
 метод естественного вывода, 12, 30  
 метод резолюции, 12, 30, 122  
 метод таблиц, 12, 27  
 метод ветвей и границ, 161  
 метрика, 51, 108, 118, 124  
 миф о пещере, 89  
 минимальная программа, 129  
 много-целевая парадигма, 169  
 многоцелевая парадигма, 175  
 многозначимый оператор, 132  
 многозначимость, 90, 158, 173  
 многозначная операция, 133  
 множественный минус, 117  
 множество началов терминалов, 168  
 множество носителя, 165  
 множество последующих терминалов, 168, 172  
 модель памяти по Бурстоллу, 91, 137  
 модель памяти по Рейнольдсу, 93, 137, 157  
 модель вычислимости, 23  
 модульная алгебра, 52  
 модульное программирование, 29  
 модульность, 163  
 модульность спецификации, 129, 133  
 модус толленс, 113

- могут-ссылаться, 69, 77
- монада, 154
- моноид, 142, 146
- морфема, 158
- мост графа, 144
- мульти-парадигмальность, 112, 114, 115, 127
- мусор, 60, 87, 138
- мутация грамматики, 158
- наблюдаемое поведение, 38
- начальная алгебра, 165
- начальное состояние, 49
- надежность, базисная, 9
- надежность, оптимальная, 10
- надежность, полная, 10
- наследственный класс, 149
- натуральные числа, 32, 100, 101, 154
- недетерминированность, 94, 97
- неинициализированное значение, 24, 41, 85
- нейтральный элемент, 142, 147
- неограниченная рекурсия, 171
- неопределённая спецификация, 26
- неопределённое значение, 73
- неорганизованная память, 66
- неповторимость, 90, 91, 142, 153
- нерешимость, 38
- нетерминал, 166, 167
- нетипизируемый терм, 27, 38
- неверный доступ, 69, 73
- незавершение вычисления, 24
- нисходящая цепочка, 19, 21
- нить, 24
- нормализация правил, 122, 128, 163
- нормализация термина, 29, 117
- нормальная форма Сколема, 28
- нормальный вектор, 156
- нумеральная логика, 117
- обыкновенные утверждения, 169
- объектно-ориентированная модель, 10
- объединение реляций, 117
- объект, 42, 43, 49, 50, 88, 90, 93, 95, 100, 110, 112, 118, 125, 130, 132
- объектный экземпляр, 42, 130, 131, 140, 174
- объектный инвариант, 51
- объектный тип, 42, 49
- объектный вид, 130
- объектно-ориентированная модель, 130
- объектно-ориентированная парадигма, 40
- объектное поле, 88, 90, 144
- объектное вычисление, 40, 41, 43
- объём кода, 66
- обобщение правила, 15
- обобщённое утверждение, 77
- обоснованность, 126, 127
- обработка человеческой речи, 107
- обратимый элемент, 143
- обратимость функции, 97, 114, 126, 154
- общая вершина, 146
- обязательно-не-ссылается, 77
- обязательно-ссылается, 77
- очередь с приоритетом, 88
- однозначный оператор, 132
- офсет, 60, 62
- онтология, 51
- операционная память, 14, 42, 66, 87, 88
- операционная семантика, 20, 23, 45, 115, 130, 153, 158
- операционная система, 70–72, 87, 160
- оператор доступа к полям объекта, 92, 163
- оператор последовательности, 93, 165
- опровержение доказательства, 113
- остаток, 16
- открытость, 126

- отмотка стека, 24
- отношение эквивалентности, 97
- отображение, 32, 97, 116
- отрицание предиката, 90, 160
- отрицание утверждения, 106, 107, 160
- отрицание выражения, 15, 107
- отсечение, 102–106, 111, 114, 116
- отсечение параметров, 24
- память
  - недоступная, 69, 70
  - нехватка, 70
  - организованная, 66
- парадигма, 23, 40, 79, 80, 108, 115, 121, 161
- парадокс, 125, 132
- парадокс Расселя, 26
- парадокс типизации, 26, 97
- параметр
  - формальный, 24
  - инкорректный, 24
  - по ссылке, 20
  - по вызову, 20, 110
  - выходной, 112
  - входной, 23, 112, 124
- параметр по вызову, 153
- параметризация предиката, 93, 98, 125, 156
- паросочетание биграфа, 138
- паттерн, 34, 41, 63, 164, 175
- перегруженное значение, 112, 126
- переименование, 48
- переименование реляций, 117
- переменная
  - автоматическая, 21, 79, 127
  - булевая, 26
  - динамическая, 21, 67, 79, 90
  - глобальная, 20, 55, 67, 94, 121
  - квантифицированная, 122, 125, 126, 154
  - логическая, 82
  - локальная, 24, 41, 48, 50, 63, 66, 67, 79, 85, 90, 100, 160, 163
  - неинициализированная, 69
  - символьная, 93, 100, 102, 121
  - статическая, 22, 82
  - свободная, 18, 90, 91, 102
  - типизированная, 161
- переполнение стека, 78
- платформа, 69
- платформа для верификации, 26
- плавление, 59
- побочный эффект, 121, 173
- подцель, 98–105, 109, 111–113, 122, 162, 166, 169, 172
- подграф, 55
- подграмматика, 135, 170
- подход Сузуци, 76
- подкласс, 42, 130, 149
- подкуча, 76, 133, 138
- подпроцедура, 94
- подсказка доказательству, 158
- подструктурная логика, 62, 142, 157
- подтерм, 125, 126
- подтип, 47
- подвыражение, 82, 102, 114
- поиск доказательства, 104
- поиск с возвратом, 113, 114
- покрытие тестов, 69
- поле, 50, 54, 83, 87, 93, 130, 149, 152
  - Галуа, 147
  - арифметическое, 143
  - атрибутное, 42
  - объекта, 22, 41, 43, 93, 140
  - прыжка, 67
  - присвоение, 141

- вещественных чисел, 143
- полиморфизм, 42, 112, 129, 130, 154
- полная абстракция, 45, 130
- полный граф, 87, 148
- полное определение, 90, 133
- полнота, 10, 17, 21, 23, 26, 29, 32, 33, 97, 130, 134
- полнота по Куку, 23, 24
- полугруппа, 142
- полуручной вывод, 157
- порог итераций, верхний, 33, 78
- порядок вычисления, 13, 21, 42, 96, 115, 154, 164
- постоянный накопитель, 70
- постусловие, 15, 128
- пошаговая аппроксимация, 31, 129
- пошаговая верификация, 111, 147
- пошаговое построение графа, 140
- поток токенов, 170
- позднее связывание, 151
- прагматика, 29, 41
- правило
- альтернативное, 103, 126, 160
  - цикла, 15, 16
  - фрейма, 63
  - контракции, 62
  - объектного построения, 48
  - последования, 14
  - присвоения, 15, 16
  - синтаксической ошибки, 170, 195
  - сопоставления, 62
  - сужения, 62, 157
  - верификации, 12, 70
- правило Хорна, 99, 100, 107, 122, 126, 127, 154, 157, 158
- право-рекурсивность, 167
- предикат, 89, 90, 93, 95, 100, 102, 106, 107, 112–115, 118, 121, 161
- голова, 100, 106, 111, 126, 167
- квантифицируемый, 23
- первого порядка, 12, 113
- противоречия, 160
- семантика, 104, 162
- тело, 100, 162
- утверждения, 26, 81
- высшего порядка, 81, 98, 108, 110, 165
- предикатная среда, 164
- предусловие, 15, 32, 128
- префикс, 167
- преобразование, 75
- преобразователь, 169
- приближение указателей, 58
- примитивная рекурсия, 24
- принцип Лискова, 51
- принцип Парнаса, 51
- принцип независимости данных, 81
- принцип скрытия данных, 209
- приостановка программы, 70
- присвоение, 48, 54
- проблема корреспонденции Поста, 168
- проблема приоризации, 106
- проблема приостановки, 97, 106, 127, 150, 164
- проблема выполнимости формулы, 23, 111, 116
- процесс, 66, 122, 123
- процессорное слово, 73, 88, 93
- продолжение, 81, 82, 114
- проекция реляций, 116, 117
- программирование через доказательство, 157
- программный оператор, 13, 14, 17, 23, 24, 31, 32, 36, 76, 82, 85, 87, 89, 90, 111, 119, 120, 129–131, 138
- промежуточное представление, 63, 108, 115, 118–

- 120, 122, 124, 128, 129, 175
- прописная дистанция, 195
- пространственный оператор, 95, 132
- противоречивый пример, 12
- прототипизация, 155
- провал, 107
- проверка моделей, 33, 34
- проверка модели, 34
- проверка типов, 27, 30, 33, 35–37, 46, 48
- псевдоним, 21, 50, 54–56, 59, 69, 74, 75, 81, 82, 97, 117, 120, 127, 128, 140, 151, 154, 155, 160
- пункт синхронизации синтаксиса, 170
- путь доступа, 87, 140
- ранг полинома, 133
- раскрыть, 194
- распознаватель, 119, 158, 175
- распознаватель, нисходящий, 158
- распознаватель, рекурсивный, 158
- распределение куч, 55
- распределение процессорных регистров, 41
- расширение языка, 71, 114, 134
- расширяемость, 112, 115, 119, 126–128
- разделение забот, 163
- разовая функция, 108, 154
- разрядность, 35, 73
- разветвление доказательства, 17
- развёртывание, 29, 55
- редекс, 27, 158
- редукция проблемы, 38
- редукция термов, 27
- регион памяти, 60, 85, 90, 93, 95, 130
- регистр, 14, 41, 77
- регистр специального назначения, 45
- регулярный граф, 87
- регулярное выражение, 83, 84, 90, 108, 119, 177
- реификация, 89
- рекурсивная схема, 101, 113, 115, 154
- рекурсивный подъём, 102, 105
- рекурсивный тип, 42, 98
- рекурсия, 21, 23, 101, 102, 106, 113
- реляционная алгебра, 116
- реляция, 26, 112, 116, 118, 124
- решаемость, 32
- решатель, 33, 64, 98, 109, 127, 134, 141, 145
- решение доказательства, 104
- решётка, 31, 41, 79
- реверс списка, 76, 77
- резолуция, 12
- результат верификации, 17
- ротация указателей, 55, 76
- самообратимая операция, 147
- сбор мусора, 11, 51, 54, 59, 60, 63, 68, 69, 71, 78, 128
- сбора мусора по поколениям, 68
- сборщик мусора, 60
- сдвиг-свёртка, 176
- семантическая функция, 152
- семантический анализ, 41, 83, 128, 131
- семантическое поле, 120
- семантика, 29, 30, 89, 108
- семейство абстрактных предикатов, 63, 165
- семиотика, 29, 90
- сеть многогранника, 156
- сходимость, 13, 19, 22, 32, 121, 139, 142, 145
- сигнатура, 45
- силлогизм, 89
- симптом проблемы, 38
- символ, 18, 34, 90, 93–96, 100, 102, 110, 113, 115–117, 122, 125, 126, 131, 141, 159, 195
- символьная среда, 162
- синтаксический анализ, 108–110, 120, 129, 130,

- 132, 135
- синтаксический анализатор, 126
- синтаксический перебор, 108, 109, 119, 157, 170, 174, 194
- синтаксический сахар, 48, 106, 154
- синтаксическое дерево, 119, 120, 128, 158
- синтаксис, 29, 48, 85, 89, 97, 109, 122
- синтезируемый атрибут, 177
- система переписывания термов, 19, 35
- слабо структурированные данные, 123
- слабоструктурированные данные, 64, 99
- слайсинг программы, 38
- слияние графов, 141
- слово формального языка, 17
- сложность, 51
- смежная запись, 176
- со-процедура, 21
- содержимое указателя, 77, 132
- соотношение графа, 148
- соотношение типов, 47–49
- сопоставление с образцами, 23, 100, 111, 128, 207, 212, 217, 218
- сопряжённое дерево, 158
- состояние кучи, 69, 81
- состояние объекта, 44
- состояние памяти, 17, 49
- состояние вычисления, 12, 13, 17, 108, 121, 127
- совместимость типов, 46, 149
- спецификация, 44, 49, 52, 75, 86, 87, 94, 114, 117, 119, 121, 122, 130
- спецификация, иерархическая, 20
- спецификация, полная, 52, 129
- список разниц, 158
- сравнение, 128
- статический анализ, 31, 33, 36, 43, 120, 128, 151
- статический анализатор, 120
- статическое поле, 63
- стек, 14, 24, 49, 50, 60, 66, 67, 71, 81, 83, 92, 94, 110, 111, 114, 115, 127, 149
- овая архитектура, 14, 153
- овое окно, 20, 54, 66, 79, 103, 110, 111, 114, 115, 153
- степень графа, 87
- стереотип, 41, 51
- стратегия, 30, 60, 104
- стратофикация, 104, 110
- строгий оператор, 139
- строгое вычисление, 23, 125
- структура данных, 87, 88, 92, 101, 125
- структурализм, 88, 139
- супремум, 42
- суждение, 49
- свойство диаманта, 19, 22
- связанный граф, 160, 163
- свёртывание, 55, 157
- свёртывание предиката, 194
- сюрьективность, 32
- сжимаемость, 91
- тактика, 25, 28, 29, 122, 126, 161
- тавтология, 95, 163
- тело правила, 106, 125
- темпоральное утверждение, 51, 52
- теорема, 29, 70, 76, 109, 127
- Биркгоффа, 117
- Чёрча-Россера, 19
- Пирса, 28
- Гёделя о неполноте, 21
- теория целых чисел, 21, 30
- теория объектов, 10, 40
- теория опровергаемости, 12
- терм, 95, 99, 100, 102, 103, 105, 111, 115, 116, 120, 124, 125, 127–129, 161

- овая алгебра, 139
- овое выражение, 133
- дерево, 117
- само-содержащий, 26
- самосодержащий, 19, 125, 171
- выходной, 96, 105, 107, 126, 154, 164
- входной, 96, 105, 107, 114, 126, 154, 164
- входной-выходной, 154
- терминация, 22, 23, 32, 72, 74
- терминал, 167, 173
- тест, 34
- тестирование, вручное, 33
- тетрады, 119, 120
- тезис-антитезис-синтезис, 89
- тип, 38, 94, 95, 102, 125, 129, 130
- тип, идеал, 47
- тип, переменной, 160
- типизация, 21, 26, 36, 38, 42, 47, 62, 73, 95, 98, 127, 128
- типизация по Чёрчу, 23, 93
- типизация высшего порядка, 47
- типизация, слабая, 68, 102
- токен, 169
- токен, предшественник, 158
- тотальность, 23, 100, 101
- трансферная функция, 54, 55
- трансформация графа, 35, 50, 86
- трансформация кода, 119
- трансформация моделей, 53, 118
- транслирующее правило, 177
- треугольник, 156, 174
- триады, 119, 120
- триангуляция, 156
- тройка Хора, 14, 17, 18, 33, 37, 114, 121
- указатель, 20, 42, 52, 55, 62, 67, 69, 73, 80, 82, 83, 85, 87, 88, 90–93, 98, 126, 134, 138, 140, 143, 144, 149, 151, 154, 156, 160
- унификация, 59, 101, 102, 116, 122, 154
- унификация термов, 96, 100, 102, 103, 105, 110, 111, 125, 166, 171
- уплотняющий сборщик мусора, 60
- упорядоченное множество, 19, 43, 46, 88, 148
- упрощение программы, 38
- уровень выразимости, 118
- уровень языковой абстракции, 124
- условный переход, 16, 23, 48, 66, 79, 119
- утечка памяти, 69, 70, 72
- утилитаристский принцип, 135
- утилизация, 44, 66, 72, 73, 76
- утилизация памяти, 80–82, 87, 88, 110
- утверждение, 30, 50, 93, 100, 101, 107, 121, 122, 127, 129–131
- уязвимость, 73
- ужесточение, 95, 132, 134
- вариабельность, 112, 114, 115, 119
- вектор термов, 100, 162
- верификация, 33, 70, 111, 112, 118, 121, 122, 125, 127, 129, 130, 135
- верификация куч, 119
- верификатор, 63
- вершина графа, 86, 87, 90, 98, 156
- вещественные числа, 26, 99
- выбор реляций, 117
- вычисление Абади-Карделли, 40, 42, 45, 128, 130
- вычисление Абади-Лейно, 40, 45, 50, 130
- вычисление Чёрча, 38
- вычисление Хиндлея и Миллнера, 37
- вычисление Хора, 9, 19, 22, 24, 29, 32, 36, 38, 208
- вычисление Карри, 38, 93
- вычисление объектов, 42, 53

- 
- вычисление регионов, 10, 64, 86
  - вычислительное состояние, 36
  - вычисляемость, 120
  - выделение памяти, 80, 82, 87, 88
  - выделение регистров, 69
  - выразимость, 21, 24, 48, 75, 86, 92, 94, 98, 109, 110, 113, 118, 119, 122, 130, 133, 135, 164
  - выталкивание из стека, 66, 71, 152
  - выводимый результат, 112
  - вызывающая сторона, 78, 103
  - вызов по значению, 66, 79, 128
  - вызов предиката, 102, 103, 105, 110, 111, 128, 166
  - вызванная сторона, 103
  - входная программа, 122, 129
  - виртуальная машина, 68
  - виртуальная память, 67, 70
  - виртуальный регистр, 49
  - висячий указатель, 50, 74, 82
  - внутренняя процедура, 22
  - внутренний класс, 63
  - внутренний объект, 149
  - возрастающая цепочка, 106
  - возврат при провале, 114
  - вспомогательный предикат, 76, 92, 122, 129, 154
  - встроенная система, 70
  - встроенный предикат, 63, 102, 107, 112, 126, 164
  - втталкивание в стек, 23, 54, 66, 71, 102, 115, 152
  - взаимная рекурсия, 16, 101
  - ячейка памяти, 66, 83, 87, 88, 95
  - ядро верификации, 25
  - язык
    - овой процессор, 169
    - декларативный, 121, 135
    - естественный, 90, 107, 110, 158
    - формальный, 17, 90, 107, 110, 157, 158
    - функциональный, 33, 96, 108
    - императивный, 13, 96, 108, 110, 114, 130, 161
    - логический, 90, 94, 96, 112
    - моделирования, 35
    - программирования, 13, 17, 42, 52, 115, 158
    - регулярный, 158
    - спецификации, 18, 93, 121, 135, 161
    - утверждений, 24, 158, 195
    - валидации, 118
    - верификации, 121, 195
    - входной, 93, 122, 128, 129
  - язык ACL2, 99
  - язык Baby Modula 3, 45, 46
  - язык Си++, 42
  - язык ISO-C++, 68
  - язык LISP, 82
  - язык Modula 2, 45
  - язык PCF, 16
  - язык XSL-T, 57, 123
  - язык Би, 32
  - язык Паскаль, 45
  - язык Си, 23, 58, 64, 68, 72, 77, 85, 92, 127, 141
  - язык Си++, 43
  - язык Ява, 42, 63, 68, 126, 176
  - замыкание, 23
  - замкнутость, 142
  - запись, 37, 38, 47
  - зависимый тип, 47, 125
  - зависимость данных, 16, 80, 95, 114, 167
  - зелёное отсечение, 106, 154
  - знак, 29
  - звезда Клини, 163
  - \_, 100, 110
  - union, 214



- , 106
- Proof, 27
- Qed, 27
- atom, 102
- calloc, 42
- compound, 102
- concatMap, 108
- concat, 110, 112, 126
- foldl, 108
- is, 101, 154
- list, 102
- malloc, 42
- new, 42, 43
- number, 102
- pack, 42
- register, 77
- self, 43
- super, 43
- take, 23, 115
- tauto, 28
- unify\_with\_check, 125
- union, 68, 75, 149
- unpack, 42
- var, 102
- ABI, 41, 78
- adjoint-trees, 158
- ANTLR, 176, 177
- ARM, 74
- bison, 176
- branch and bound, 161
- bss, 67, 73
- builder, 38
- built-in команда, 158
- clang, 119
- co-domain, 170
- const, 77
- constraint programming, 127
- Coq, 25, 157
- Cyclone, 64
- DCG, 126, 158
- design by contract, 51
- difference lists, 158
- DOT, 125, 127, 128
- doubly-connected edge list, 156
- ElectricFence, 64
- error production rules, 170
- first set, 172
- first-terminal sets, 168
- fold, 163, 165, 171
- follow-set, 168, 172
- footprint, 129
- Gallina, 25
- GCC, 41, 58, 71, 119, 120
- GIMPLE, 63, 119, 120
- GNU, 68, 111
- GNU Linux, 78
- GNU make, 38
- GNU Prolog, 175
- Intel, 74
- JIMPLE, 63
- join, 147
- jStar, 63, 91, 120
- KeY, 64
- LALR-анализатор, 171
- linux, 56
- LL(k)-распознаватель, 173

- 
- LL-анализатор, 108, 171
  - LLVM, 64, 71, 119, 120
  - LLVM биткод, 119, 120
  - mesh, 156
  - Modula, 46
  - name mangling, 176
  - NP-твёрдая проблема, 69
  - Object Constraint Language, 52
  - OCaML, 21, 64
  - occurs-check, 171
  - OCL, 35, 52, 134, 154
  - Pascal, 46
  - PCF, 16, 128
  - poset, 43, 147
  - PowerPC, 74
  - production rules, 173
  - program slicing, 38
  - REDUCE, 171
  - SAFECode, 64
  - satisfiability-modulo-theory solver, 33
  - SATlrE, 64
  - Semantic Web, 109
  - separation of concerns, 51
  - SHIFT, 171
  - shrinker, 38
  - SLR-анализатор, 171
  - Smallfoot, 63, 91
  - SpaceInvader, 63
  - SSA-форма, 59, 79, 80, 95
  - stderr, 38
  - stdout, 38
  - Stratego XT, 35
  - term rewriting system, 35
  - transducer, 169
  - tree graph matcher, 128
  - UML, 35, 52, 64, 134
  - unfold, 163, 171
  - union, 59
  - Unix, 68
  - Valgrind, 64
  - VDM++, 34, 64
  - Verifast, 64, 157
  - Vernacular, 25
  - Why, 33
  - Windows, 73
  - wrong, 46
  - XML, 57, 118
  - xor, 60
  - Y-not, 33
  - yacc, 176
  - Ynot, 64