ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

На правах рукописи

Ушакова Мария Сергеевна

Методы и инструментальные средства формальной верификации функционально-потоковых параллельных программ

2.3.5 — Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель
доктор техн. наук, профессор
Легалов Александр Иванович

Красноярск 2021

Содержание

Введение **5**

1 Методы и средства формальной верификации программ **11**

1.1 Классификация ошибок в программах 11

1.2 Методы верификации программ 12

1.2.1 Классификация формальных методов верификации программ 13

1.2.2 Методы проверки моделей 16

1.2.3 Методы дедуктивного анализа 17

1.2.4 Применимость формальных методов к верификации ФПП программ . . . 21

1.2.5 Методы доказательства завершения программ 23

1.3 Инструментальная поддержка доказательства корректности программ 27

1.3.1 Уточняющие типы 27

1.3.2 Контрактное программирование 28

1.3.3 Верификатор Boogie 29

1.3.4 Верификация Java-программ со спецификацией на JML 31

1.3.5 Средства верификации C-программ 32

1.3.6 Предикатное программирование 34

1.3.7 Обобщённая схема инструментальных средств и её применимость к ФПП программам 35

1.3.8 Инструментальная поддержка доказательства теорем 37

Выводы 40

2 Аксиоматическая семантика ФПП программ **41**

2.1 Семантика языка Пифагор 41

2.1.1 Общие принципы организации модели вычислений 41

2.1.2 Этапы выполнения оператора интерпретации 42

2.1.3 Семантика типов данных языка Пифагор 43

2.2 Спецификация свойств программ на языке Пифагор 46

2.2.1 Система типов языка спецификации 48

2.2.2 Синтаксис языка спецификации 48

2.2.3 Семантика языка спецификации 51

2.2.4 Теории языка спецификации 52

2.2.5 Совместимость языка спецификации с системой HOL 58

2.3 Аксиоматическая теория языка Пифагор 59

2.3.1 Язык аксиоматической теории 59
2.3.2 Аксиомы 61
2.3.3 Правила вывода 63
2.4 Преобразования информационного графа с разметкой 67
2.4.1 Разметка дуг 68
2.4.2 Свёртка 71
2.4.3 Изменение информационного графа 77
Выводы 83

3 Формальная верификация ФПП программ 84
3.1 Доказательство корректности рекурсивных функций 84
3.1.1 Доказательство частичной корректности рекурсивной функции 85
3.1.2 Доказательство завершения рекурсии 89
3.2 Верификация программ со взаимной рекурсией 92
3.2.1 Универсальная рекурсивная функция 94
3.2.2 Объединение функций 95
3.2.3 Алгоритм преобразования произвольной рекурсии в прямую 96
3.2.4 Преобразование рекурсивных функций на языке Пифагор 97
Выводы 110

4 Инструментальная поддержка формальной верификации ФПП программ 112
4.1 Реализация системы 114
4.2 Модуль поддержки РИГ 114
4.2.1 Текстовое и внутренне представление информационного графа программы 115
4.2.2 Основные функции модуля поддержки РИГ 116
4.3 Модуль поддержки термов 117
4.3.1 Текстовое представление термов 117
4.3.2 Внутреннее представление термов 118
4.3.3 Трансляция термов из текстового представления во внутреннее 122
4.3.4 Основные функции модуля поддержки термов 123
4.4 Модуль поддержки ИГР 123
4.4.1 Внутренне представление информационного графа с разметкой 123
4.4.2 Основные функции модуля поддержки ИГР 124
4.5 Модуль поддержки дерева доказательства 127
4.6 Модуль поддержки библиотеки аксиом и теорем 128
4.6.1 Внутренне представление аксиом и теорем 128

4.6.2 Основные функции модуля поддержки библиотеки аксиом и теорем . . . 130
4.7 Интерфейс пользователя . . . 130
4.7.1 Редактор ИГР . . . 131
4.7.2 Редактор узлов дерева доказательства . . . 137
4.7.3 Управление библиотекой аксиом и теорем . . . 137
Выводы . . . 138

Заключение . . . 140

Список литературы . . . 141

Список сокращений . . . 157

Приложение А Акты о внедрении . . . 158

Приложение Б Семантика встроенных функций языка Пифагор . . . 160

Приложение В Стандартные теории языка спецификации . . . 168

Приложение Г Описание условий корректности ФПП программ с помощью логики HOL . . . 174

Приложение Д Аксиомы встроенных функций языка Пифагор . . . 181

Приложение Е Примеры верификации рекурсивных функций . . . 186

Приложение Ж Частный случай рекурсии с явно выраженным рекуррентным соотношением . . . 198

Приложение З Алгоритм работы системы поддержки доказательства . . . 204

Приложение И Пример текстового представления РИГ . . . 205

Приложение К Синтаксис текстового представления языка спецификации 206

Приложение Л Входные данные глобального окружения . . . 209

Приложение М Диаграмма лексического анализа текстового представления термов . . . 212

Приложение Н Диаграмма синтаксического анализа текстового представления термов . . . 213

Приложение О Пример текстового представления ИГР . . . 215

Введение

Актуальность темы и степень её разработанности. Постоянный рост объёма разрабатываемых программных систем и их ориентация на параллельную обработку данных ведут к увеличению сложности программного обеспечения (ПО), что, в свою очередь, повышает вероятность возникновения различных ошибок. Традиционный способ обеспечения надёжности программ путём тестирования не может полностью удовлетворить возрастающие требования практического использования. Альтернативой тестированию являются методы формальной верификации, которые обеспечивают анализ всех возможных вариантов поведения системы. Формальная верификация — это формальное доказательство того, что программа соответствует своей спецификации [1, 2]. Методы формальной верификации позволяют доказать отсутствие ошибок в программе, в то время как тестирование лишь выявляет ошибки, но не даёт гарантии их отсутствия.

Существуют разнообразные подходы к формальной верификации программ [3]. Основными являются методы проверки моделей [4], дедуктивный анализ [1], различные варианты уточнения программ [5, 6].

В настоящее время достигнуты определённые успехи в практическом применении формальной верификации к последовательным программам. Разработан ряд систем для поддержки этого процесса. В качестве примера можно привести верификаторы программ на языке Си: Boogie [7] и система СПЕКТР [8]; системы для верификации объектно-ориентированных программ на Java: LOOP [9] и KeY [10]; системы для верификации функциональных программ: DSolve [11], HSolve [12].

Параллельное программирование позволяет существенно увеличить производительность на современных вычислительных системах. Однако параллелизм приводит к значительному усложнению разработки, а особенно отладки и верификации программ. Непосредственная ориентация современного параллельного программирования на существующие архитектуры затрудняет формальную верификацию, так как наряду с логической корректностью программ приходится исследовать и влияние ресурсных ограничений, порождающих дополнительные ошибки. Ситуация может быть улучшена, если при разработке параллельных программ на первоначальных этапах можно было бы избавиться от ресурсных конфликтов. Одним из таких подходом является архитектурно-независимое параллельное программирование [13, 14, 15, 166]. Представитель данного направления — функционально-потоковая модель параллельных вычислений (ФПМПВ) и созданный на её основе язык программирования Пифагор [16, 17]. Основными идеями подхода являются исключение ресурсных ограничений и неявное управление вычислениями. В функционально-потоковой параллельной (ФПП) программе описываются только информационные связи между выполняемыми функ-

циями. Взаимодействие между функциями при этом осуществляется по готовности данных, что позволяет создавать программы с неограниченным параллелизмом, «сжатие» которого к конкретным вычислительным ресурсам и условиям эксплуатации будет происходить после верификации и отладки написанных программ [166, 167].

Перспективным видится то, что в отсутствии ресурсных ограничений основной упор в разработке параллельных программ и их анализе можно сделать именно на формальную верификацию при отсутствии ресурсных конфликтов. Вместе с тем следует отметить, что вопросы, связанные с формальной верификацией, в рамках данного направления практически не проработаны. Имеются работы, связанные с организацией отладки ФПП программ и использованием методов верификации для анализа корректности данных, не связанных с формальным доказательством логической корректности кода [18, 19, 168, 169].

Поэтому актуальной является разработка формальных методов и средств верификации для языка функционально-потокового параллельного программирования, обеспечивающих проверку логики функционирования программ.

Цель и задачи исследования. Целью работы является повышение надёжности и корректности функционально-потоковых параллельных программ посредством разработки методов формальной верификации.

Для достижения указанной цели в работе решаются следующие задачи:

— проанализировать существующие подходы к формальной верификации и возможности их использования при функционально-потоковом параллельном программировании;

— формально описать семантику языка ФПП программирования Пифагор для формирования основы методов анализа ФПП программ;

— разработать методы, обеспечивающие формальную верификацию ФПП программ;

— разработать инструментальное средство, обеспечивающее поддержку методов формальной верификации ФПП программ.

Объектом исследования является функционально-потоковая модель параллельных вычислений и язык функционально-потокового параллельного программирования.

Предметом исследования являются методы и средства формальной верификации функционально-потоковых параллельных программ.

Научная новизна.

1. Для доказательства корректности функционально-потоковых параллельных программ, написанных на языке Пифагор, впервые разработан метод верификации на базе исчисления Хоара, эквивалентный по сложности методам доказательства корректности для последовательных программ, но позволяющий доказывать корректность программы без огра-

ничения параллелизма, что обеспечивается концепцией неограниченных вычислительных ресурсов.

2. Для доказательства завершения функционально-потоковых параллельных программ впервые предложен метод, использующий ограничивающую функцию, который допускает изменение спецификации программы таким образом, чтобы доказательство частичной корректности одновременно характеризовало и завершение программы. Это позволяет строить инструментальные средства поддержки доказательства на одной общей основе.

3. Для функционально-потоковых параллельных программ впервые предложен метод удаления взаимной рекурсии нескольких функций, позволяющий преобразовывать произвольную функцию в функцию с прямой рекурсией, это повышает эффективность верификации за счёт преобразования программы только к одной функции.

4. Впервые разработана архитектура и реализован прототип инструментального средства, обеспечивающего поддержку верификации функционально-потоковых параллельных программ с помощью предложенных методов. Это позволяет упростить процесс формальной верификации за счёт наглядной визуализации дерева доказательства и автоматизации преобразований графа анализируемой функции.

Положения, выносимые на защиту.

1. Разработанный метод верификации на базе исчисления Хоара для ФПП программ на языке Пифагор позволяет доказывать частичную корректность ФПП программ.

2. Разработанный метод доказательства завершения ФПП программ совместно с методом верификации позволяет доказать тотальную корректность программы.

3. Предлагаемый метод удаления взаимной рекурсии нескольких функций позволяет преобразовать любую рекурсивную функцию ФПП программы в прямую рекурсию, чтобы применить к ней метод верификации.

4. Разработанный прототип инструментального средства для поддержки верификации ФПП программ позволяет визуализировать процесс доказательства и автоматизирует построение дерева доказательства для упрощения процесса формальной верификации.

Теоретическая и практическая значимость работы. Результаты работы могут быть использованы для повышения эффективности разработки параллельных программ. На основе предложенных методов формальной верификации разработано инструментальное средство, обеспечивающее поддержку верификации функционально-потоковых параллельных программ.

Исследование выполнено при финансовой поддержке РФФИ в рамках научных проектов № 13-01-00360, № 17-07-00288 и ФЦП «Научные и научно-педагогические кадры инновационной России» № 14.A18.21.0396.

Полученные научные и практические результаты использованы в учебном процессе на кафедре вычислительной техники и кафедре высокопроизводительных вычислений Института космических и информационных технологий СФУ при изучении дисциплин «Технологии разработки программного обеспечения», «Параллельное программирование», «Высокопроизводительные вычисления на графических процессах» и при выполнении выпускных квалификационных работ.

Результаты исследования использованы при разработке функционально-потоковых параллельных программ для процесса высокоуровневого проектирования цифровых схем на базе функционально-потоковой парадигмы. Для разработки ФПП программ использовалась формализованная семантика языка Пифагор, проводилась формальная верификация ФПП программ перед их преобразованием в программы для сверхбольших интегральных схем.

Результаты практического применения диссертационного исследования подтверждены актами о внедрении (приложение А).

Методы исследования. В работе использовались элементы теории множеств, методы математической логики (различные логики и аксиоматические теории, λ-исчисление, метод доказательства теорем на основе исчисления Хоара, методы доказательства завершения программ на базе трансфинитной индукции), методы теории рекурсивных функций (построение универсальной рекурсивной функции для удаления взаимной рекурсии нескольких функций), элементы теории графов, методы и понятия теории алгоритмов, теории языков программирования, теории языков и формальных грамматик, теория разработки трансляторов. Для описания результатов использовались UML-диаграммы, диаграммы Вирта и формы Бэкуса-Наура.

При создании программных инструментов использовались структурное и объектно-ориентированное программирование, кросс-платформенная библиотека Qt.

Апробация работы. Основные положения диссертации докладывались и обсуждались на 15 конференциях и семинарах, основные из которых: Международная научная конференция «Параллельные вычислительные технологии» (Челябинск, 2013; Ростов-на-Дону, 2018), Parallel Computing Technologies International Conference (Санкт-Петербург, 2013), Международная конференция «IX Сибирский конгресс женщин-математиков» (Красноярск, 2016), Всероссийская научная конференция памяти А.Л. Фуксмана «Языки программирования и компиляторы» (Ростов-на-Дону, 2017), Международная конференция «Суперкомпьютерные дни в России» (Москва, 2018).

Получено два свидетельства о регистрации программного обеспечения (2013, 2021).

По теме диссертации опубликовано двадцать три научные работы; из которых шесть статей в изданиях, рекомендуемых ВАК; три статьи входят в список Web of Science Core

Collection; пять статей индексируется в Scopus.

Личный вклад автора. Основные результаты являются новыми и получены лично автором. Автором проведён значительный объём научных изысканий, подготовлены публикации; предложен метод верификации на базе исчисления Хоара и доказательства завершения ФПП программ на языке Пифагор; предложен метод удаления взаимных рекурсий нескольких функций в программах на языке Пифагор; разработано инструментальное средство поддержки формальной верификации ФПП программ. Научные работы, опубликованные в соавторстве с научным руководителем, заключаются в разработке метода формальной верификации ФПП программ на основе дедуктивного анализа; разработке архитектуры инструментального средства поддержки формальной верификации ФПП программ. Совместно с научным руководителем автор осуществлял постановку целей и задач и анализ полученных результатов. В совместных публикациях автора с Удаловой Ю.В. включен разработанный автором метод доказательства завершения программ на языке Пифагор. В совместных публикациях автора с Непомнящим О.В., Матковским И.В., Васильевым В.М. и Романовой Д.С. научные результаты автора являются дополнением к концепции разрабатываемого ими транслятора функционально-потоковых параллельных программ.

Структура работы. Диссертационная работа состоит из введения, 4 глав, заключения, списка литературы, включающего 188 наименований, списка сокращений и 14 приложений. Работа изложена на 157 листах машинописного текста, содержит 41 рисунок, 6 таблиц.

Во введении приводится общая характеристика работы и даётся краткий обзор содержания диссертации.

В первом разделе проводится анализ существующих подходов к формальной верификации программ. На основе проведённого анализа выбираются наиболее подходящие методы для верификации ФПП программ. Проводится анализ инструментальных средств поддержки формальной верификации программ. На основе анализа предлагается общая схема архитектуры инструментального средства, подходящая для верификации ФПП программ.

Во втором разделе анализируются и предлагаются методы формальной верификации ФПП программ. Разрабатывается аксиоматическая система для верификации ФПП программ на языке Пифагор, являющегося практическим воплощением ФПМПВ. Для этого проводится формализация семантики ФПП программ, выбирается язык спецификации для формулировки свойств ФПП программ, разрабатываются аксиомы и правила вывода аксиоматической системы. Предлагается наглядный способ проведения доказательства на информационном графе программы.

В третьем разделе рассматривается доказательство корректности функций, содержа-

щих рекурсию. Процесс доказательства разделяется на два этапа: доказательство частичной корректности и завершения программы. Предлагается метод доказательства завершения программы, расширяющий исчисление Хоара для доказательства тотальной корректности, и метод удаления взаимной рекурсии нескольких функций.

В четвёртом разделе представлена архитектура системы, обеспечивающей поддержку доказательства корректности ФПП программ, написанных на языке Пифагор. Описан прототип системы, построенный на базе данной архитектуры. Данная система визуализирует процесс доказательства. Часть этапов доказательства выполняется автоматически; для другой части этапов, проходящих в интерактивном режиме, система предоставляет всевозможные варианты пути доказательства, из которых пользователь выбирает допустимые.

В заключении формулируются основные результаты работы.

Автор благодарен научному руководителю профессору Легалову Александру Ивановичу за постановку задачи и внимание к работе. Признателен сотрудникам кафедры вычислительной техники Института космических и информационных технологий СФУ за хорошие условия работы над диссертацией.

1 Методы и средства формальной верификации программ

В главе 1 проводится анализ литературных источников. Классифицируются ошибки, возникающие в программах и выделяются ошибки, характерные для ФПП программ. Приводится классификация методов верификации программ, на её основе проводится сравнительный анализ методов и исследуется их применимость к верификации ФПП программ. Анализируется архитектура существующих инструментальных средств поддержки доказательства корректности программ, что позволяет выбрать общую схему архитектуры инструментального средства верификации ФПП программ.

1.1 Классификация ошибок в программах

Классифицируем ошибки, которые возникают в программах, в частности, в ФПП программах. Считается, что корректная программа должна, проработав, завершиться и вернуть в качестве результата ответ. Исходя из этого, ошибки программы можно разделить на две группы [1].

1. Ошибки, в результате которых программа не завершается. Они, в свою очередь, могут быть вызваны:

   1.1. вызовом частично определённых функций со значением аргумента вне области определения (аварийное завершение);

   1.2. выполнением бесконечной последовательности операторов в цикле (зацикливание).

2. Ошибки в семантике программы, в результате которых программа, если завершается, то возвращает неправильный ответ.

Существуют программы, для которых предполагается бесконечная работа в цикле, например, программа, описывающая работу некоторого прибора [20]. Для таких программ зацикливание не является ошибкой.

Параллельное программирование вносит в программы дополнительные ошибки, связанные с его спецификой [21]. Перечислим наиболее распространённые из них.

1. Неверное взаимодействие процессов (mismatched communications), связанное с потерей или неправильным обменом сообщениями, когда не те данные передаются не тому процессу.

2. Взаимная блокировка (deadlocks) — ситуация, при которой несколько процессов находятся в состоянии бесконечного ожидания ресурсов, захваченных самими этими процессами. Это приводит к зависанию или аварийному завершению программы.

3. Состояние гонки (race conditions) — ошибка, при которой работа системы зависит от того, в каком порядке выполняются части кода. В частности, оно возникает, когда несколько

процессов неконтролируемо обращаются к одной и той же разделяемой переменной. В этом случае значение переменной будет зависеть от порядка, в котором процессы получают к ней доступ.

4. Неверное разделение ресурсов (false sharing). Оно возникает в системах с разделяемой памятью, когда два процесса пытаются изменить различные участки кэша, в результате происходит его повторная загрузка из основной памяти.

В основном данные ошибки не связаны с логикой выполнения программы и с некорректным использованием памяти, а являются следствием взаимодействия ограниченных вычислительных ресурсов.

Ошибки функционально-потоковых параллельных программ. Ошибки в ФПП программах определяются спецификой ФПМПВ. Их также можно разделить на две группы [170]: ошибки в семантике программы и невозможность завершения программы. Однако ошибки второй группы могут быть вызваны только зацикливанием программы в результате бесконечной рекурсии, потому что в языке нет частично определённых функций, и все функции возвращают результат для любого аргумента.

В ФПП программах отсутствуют ошибки, обусловленные взаимодействием ограниченных ресурсов, это определяется спецификой модели вычислений [16].

1. Архитектурная независимость, достигаемая за счёт описания в программе только информационных связей.

2. Асинхронный параллелизм, поддерживаемый выполнением операций по готовности данных.

3. Принцип единственного присваивания, обусловленный прямым взаимодействием функций через информационные связи, что позволяет избежать многократного изменения памяти.

4. Принцип единственного использования вычислительных ресурсов, позволяющий не рассматривать ошибки, связанные с конфликтом ресурсов.

5. Отсутствие операторов цикла, что позволяет избежать конфликтов при обработке различных данных в одних и тех же фрагментах параллельной программы.

Таким образом, анализ корректности функционально-потоковых параллельных программ в целом сводится к анализу ошибок, аналогичных ошибкам, возникающим в последовательных программах.

1.2 Методы верификации программ

Основными понятиями в области формальной верификации программ являются понятия критериев качества, спецификации и верификации. Самым важным критерием качества

программ является *надёжность* (reliability) — способность ПО поддерживать определенную работоспособность в заданных условиях [22]. Надёжность сочетает в себе два фактора: корректность (correctness) и устойчивость (robustness) [23, 24]. *Устойчивость* — это способность ПО соответствующим образом реагировать на аварийные ситуации [25]. *Корректность* — это способность ПО выполнять задачи в строгом соответствии со спецификацией [25]. *Спецификация программы* — свойства, условия или требования, которым должна удовлетворять программа.

Выделяют два свойства корректности: частичная корректность и завершение программы [1]. *Завершение программы* — достижение в процессе выполнения программы, начатой в допустимом (для входной спецификации) состоянии, выхода программы. *Частичная корректность* — способность программы вернуть правильный результат (согласно заданной спецификации), если она начата в допустимом состоянии, при условии, что она завершается. При выполнении обоих свойств программа является *полностью (тотально) корректной.*

Верификация (в широком смысле) — это вид деятельности, направленный на контроль качества программного обеспечения и обнаружение в нём ошибок. К верификации относят следующие методы: экспертиза, статический анализ, формальные, динамические (тестирование) и синтетические методы [22].

В данной работе исследуются методы формальной верификации. *Формальная верификация* (верификация в узком смысле) — это доказательство корректности программы, которое заключается в установлении соответствия между программой и её спецификацией, описывающей цель разработки [1]. Далее термин верификация будет использоваться в узком смысле.

1.2.1 Классификация формальных методов верификации программ

Существует множество различных методов верификации программ [3]. Чтобы выбрать подходящий для верификации ФПП программ, рассмотрим несколько классификаций данных методов. В работе [1] представлена классификация методов в зависимости от этапа разработки, на котором они применяются.

1. Аналитический подход, при котором проводится верификация готовой программы или её фрагмента (рисунок 1.1а). В данном случае программирование предшествует доказательству свойств программы. Доказательство заключается в установлении соответствия между формализованной моделью программы и спецификацией с помощью формальных методов.

2. Синтетический (конструктивный) подход, при котором пошагово конструируется программа совместно с доказательством её корректности (рисунок 1.1б). Первоначально

имеется спецификация всей программы или её отдельных фрагментов, далее спецификация поэтапно преобразуется в программу с помощью формальных преобразований. Преобразования производятся через переходные состояния, которые представляют собой комбинацию программы и спецификации.

3. Верификация динамических свойств вычислений, при которой обнаруживаются такие ошибки, как незавершение циклов, некорректность операций (деление на ноль, выталкивание из пустого стека и др.), выход за границы массива, тупики в параллельных программах и др. Отличительной особенностью верификации этих свойств является то, что она может производиться при частичном или даже полном отсутствии спецификаций.

4. Верификация формальных спецификаций проблемных областей, представляющих аксиоматизацию основных понятий этих областей.

5. Верификация правил преобразований для трансформационного синтеза программ.

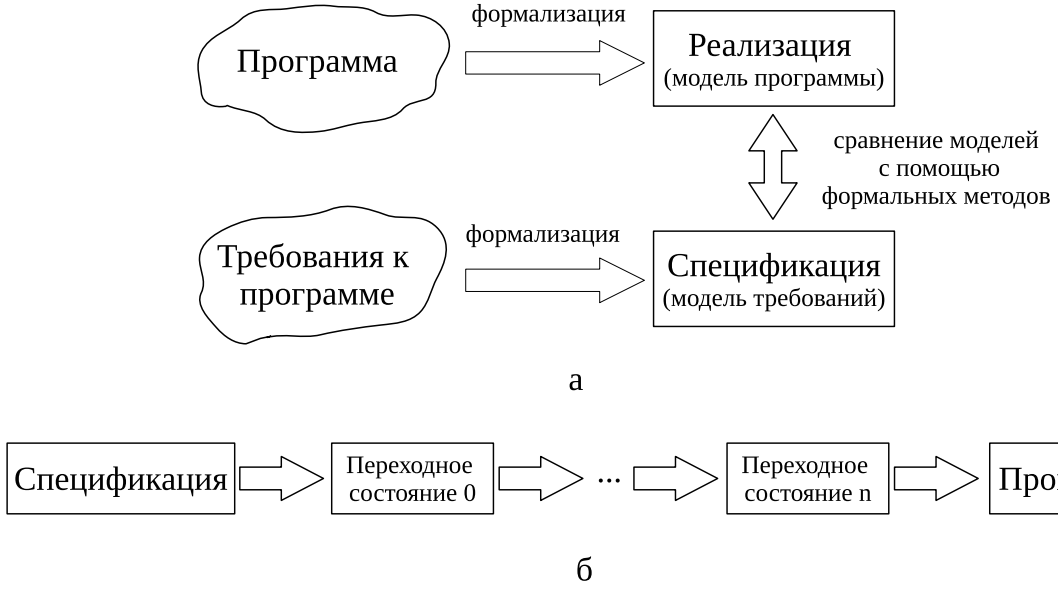


Рисунок 1.1 — Схема двух подходов к верификации программ; а — аналитический подход; б — конструктивный подход

Методы первых двух групп принципиально отличаются подходом к процессу верификации. В первом случае программа считается уже написанной, а во втором она формируется из спецификации. В целом оба этих подхода могут быть применимы к верификации ФПП программ. Часть методов третей группы могут быть применимы к ФПП программам. К ним относятся методы доказательства завершения программы, так как в ФПП программах возможно зацикливание в результате бесконечной рекурсии. Последние две группы методов исключаются из рассмотрения, так как не относятся к задачам данной работы. В работе не ставится цели доказательства корректности спецификации программы, спецификация считается заданной правильно изначально. После верификации ФПП программы предполагается

преобразовывать её в программы для реальных архитектур. В этом случае саму программу можно рассматривать как спецификацию для программ реальных архитектур (ФПП программа будет стоять на первой позиции схемы рисунка 1.1б). Для доказательства корректности этого процесса возможно применение методов верификации правил преобразования. Однако этот вопрос также не рассматривается на данном этапе исследования.

Другая классификация формальных методов верификации основывается на формальных моделях требований и свойств программ [2, 22]. Формальные модели подразделяют на логико-алгебраические, исполнимые и промежуточные. Логико-алгебраические модели (логические или алгебраические исчисления) при моделировании ПО описывают некоторый набор его свойств, возможно, изменяющийся со временем, но не дают точного представления о том, за счёт чего изменяются эти свойства. Примерами логических моделей является исчисление высказываний, исчисление предикатов, λ-исчисление, модальные и темпоральные логики; примеры алгебраических моделей — реляционные алгебры и алгебры процессов. Исполнимые модели характеризуются тем, что их можно каким-то образом выполнить, чтобы проследить изменение свойств моделируемого ПО. К исполнимым моделям относят конечные автоматы, сети Петри и др. К промежуточным моделям относят логики Хоара, программные контракты.

Поскольку, реализация и спецификация могут быть моделями двух основных видов, логико-алгебраическими или исполнимыми, возможно четыре разных комбинации этих видов моделей [22]. Однако на практике не используют для спецификации исполнимую модель в сочетании с логико-алгебраической для реализации, потому что в такой комбинации невозможно обеспечить бóльшую абстрактность спецификации по сравнению с реализацией. Остаются три случая.

1. Спецификация S и реализация I представлены как логико-алгебраические модели. В этом случае выполнение специфицированных свойств в реализации моделируется отношением *выводимости* (записывается как $I \vdash S$). Чаще всего для проверки используется метод *дедуктивного анализа* (theorem proving).

2. Спецификация S является логико-алгебраической моделью, а реализация I — исполнимой. Выполнение специфицированных свойств в реализации в этой ситуации называется отношением *выполнимости* (записывается как $I \models S$). Наиболее проработанный и распространённый — *метод проверки моделей* (model checking).

3. Спецификация S и реализация I представлены как исполнимые модели. В этом случае общепринятого названия нет, используются термины «симуляция» или «моделирование» (simulation), «сводимость» (reduction), «соответствие» или «согласованность» (conformance). Для методов проверки тоже нет общепринятого названия. Отнесем к этому пункту *имита-*

ционное моделирование (simulation modeling).

Последняя группа методов подразумевает, что программа заменяется некоторой исполнимой моделью, однако ФПП программа сама может рассматриваться как модель в рамках конструктивного подхода. Кроме того, данные методы позволяют лишь выявить отдельные ошибки в программе, но не дают формальное доказательство её корректности. Ввиду этих причин, в дальнейшем методы проверки согласованности не рассматриваются.

Для выбора подходящих методов для верификации ФПП программ рассмотрим и сравним основные методы двух первых групп.

1.2.2 Методы проверки моделей

Проверка моделей (model checking) — это автоматический метод верификации параллельных программ с конечным числом состояний [2, 4]. *Состояние* — это моментальный снимок или мгновенное описание системы, в котором зафиксированы значения переменных в конкретный момент времени. Метод проверки моделей направлен на систематическое исследование пространства состояний программы, начиная с начального состояния и следуя по всем возможным путям выполнения, пока не будут пройдены все достижимые состояния или не будет достигнуто состояние ошибки.

Чаще всего для описания проверяемых свойств в методе проверки моделей используется некоторая временная (темпоральная) логика [26], а в качестве модели выступает конечный автомат, состояния которого соответствуют наборам значений элементарных формул в проверяемых свойствах, обычно он называется моделью Крипке. Проверку модели выполняет специализированный инструмент, который либо подтверждает, что модель действительно обладает заданными свойствами, либо выдаёт сценарий её работы, в конце которого эти свойства нарушаются, либо не может прийти к определенному вердикту, поскольку анализ модели требует слишком больших ресурсов [22].

Такой подход применим только к программам с конечным числом состояний. Если программа оперирует неограниченно большими структурами данных, такими как списки и деревья, то набор возможных состояний программы будет бесконечным, и исследование пространства состояний может не завершиться. Поэтому, чтобы эффективно проверять программы с бесконечным числом состояний, создается абстракция пространства состояний, которая разбивает бесконечное пространство состояний на подмножества, представленные конечным числом абстрактных элементов. Далее можно смоделировать проверку соответствующей абстрактной программы с конечным числом состояний, которая имитирует исходную программу, но оперирует абстрактными, а не конкретными состояниями [27].

1.2.3 Методы дедуктивного анализа

Дедуктивная верификация — это проверка правильности программы, которая сводится к доказательству теорем в подходящей логической системе. Впервые формально поставили проблему доказательства корректности программ Роберт Флойд и Энтони Хоар [28, 29]. Верификация программ проводится дедукцией на основе аксиом и правил вывода. Эта достаточно сложная процедура не может быть полностью автоматизирована, она требует участия человека, действующего на основе предположений и догадок, использующего интуицию при построении инвариантов и нетривиальном выборе альтернатив [2]. В данном случае под *инвариантом* понимается некоторая логическая формула, которая сохраняет истинность в некоторой точке программы вне зависимости от пути по которому система приходит в данную точку [1].

Методы дедуктивного анализа достаточно разнообразны. Однако можно выделить два основных типа логических систем, которые лежат в их основе. Первый тип — классические логические системы, к которым относятся исчисления высказываний и предикатов различных порядков [30, 31, 32], второй тип — лямбда-исчисление (λ-исчисление) [33, 34, 35].

Ниже приведено краткое описание основных групп методов дедуктивного анализа, необходимое для их сравнения и анализа применимости к верификации ФПП программ.

Исчисление Хоара. Одно из направлений дедуктивного анализа основано на логике Хоара (Hoare logics) [29], которая предоставляет один из наиболее наглядных способов представления спецификации и реализации. Логики Хоара являются специфическим видом логик, утверждения которых состоят из формул логики некоторого вида и программных инструкций. В простейшем виде это *тройки Хоара* — формулы вида

$$\{\phi\}\texttt{Prog}\{\psi\}, \tag{1.1}$$

где `Prog` — часть программы на определённом языке программирования, а φ и ψ — формулы некоторой логики, зависящие от переменных, входящих в `Prog`. Формула φ интерпретируется как условие, выполненное перед началом исполнения `Prog` и называется *предусловием*, а ψ — как условие, которое должно быть выполнено после этого исполнения, называется *постусловием*. Тройка Хоара является истинной, если ψ всегда истинно после исполнения `Prog`, начинающегося из состояния, в котором истинно φ. Формально, тройка Хоара (1.1) определяет предикат

$$\forall s_0 \in S.\ (\varphi(s_0) \wedge fin(\texttt{Prog}, s_0)) \Rightarrow \psi(s_0, f_{\texttt{Prog}}(s_0)), \tag{1.2}$$

где S — множество всех состояний программы `Prog`, s_0 — начальное (входное) состояние программы, $fin(\texttt{Prog}, s_0)$ — предикат завершения выполнения программы, $f_{\texttt{Prog}}$ — функция,

вычисляемая программой Prog.

Данная интерпретация тройки Хоара позволяет рассматривать её как некоторый аналог формулы, то есть объект, принимающий значение true или false. Тройка Хоара является элементарной единицей рассуждений о свойствах программ. При использовании троек Хоара оказываются выразимыми многие свойства. Однако существуют и невыразимые свойства, например, свойство завершения. Поэтому (1.2) содержит предикат завершения fin и выражает свойство частичной корректности. Свойство завершения программы можно определить как $\forall s_0.\, \varphi(s_0) \Rightarrow fin(s_0)$. Тогда свойство полной (тотальной) корректности выражается как

$$\forall s_0.\, \varphi(s_0) \Rightarrow \Big(\psi(s_0, f_{\texttt{Prog}}(s_0)) \wedge fin(\texttt{Prog}, s_0)\Big).$$

Одним из первых методов доказательства частичной корректности программ, основанных на логики Хоара, является метод индуктивных утверждений [1, 28]. Он позволяет свести доказательство свойства частичной корректности к доказательству некоторого конечного числа утверждений. Вся программа разбивается на трассы вычислений. Для разделения на трассы циклических вычислений в программу включаются дополнительные индуктивные утверждения, называемые инвариантами цикла [1, 36]. Каждой трассе ставится в соответствие своя тройка Хоара. Чтобы программа была корректна, истинными должны быть все полученные тройки.

Однако метод индуктивных утверждений является неформальной аксиоматической теорией [37]. Более высокий уровень абстракции предоставляет аксиоматический подход, предложенный Хоаром [29]. Общую идею доказательства корректности программ на некотором языке P с помощью исчисления Хоара можно представить следующим образом. Вначале выбирается некоторая аксиоматическая теория $\mathfrak{T}$ с языком L. На этом языке формулируются требования к программе, то есть L — это язык спецификации. Обычно в теорию $\mathfrak{T}$ входят общелогические аксиомы (например, логики первого или высшего порядков) и аксиоматические теории, описывающие семантику типов данных языка программирования (например, аксиоматическая теория для целых, действительных чисел, списков и других встроенных типов языка как простых, так и составных) [37, 38]. При этом, чем более выразительно исчисление, тем удобнее с его помощью формулировать необходимые утверждения, но тем более сложной задачей является проверка их доказуемости [22].

Далее теория $\mathfrak{T}$ расширяется с помощью троек Хоара. Также добавляются аксиомы, характеризующие семантику встроенных операторов языка программирования P, и правила вывода, позволяющие выражать свойства композиции операторов через свойства её составных частей. Получается новая расширенная теория $\mathfrak{H}$ — исчисление Хоара для языка программирования P. Процесс доказательства корректности некоторой программы $prog$ в

данном исчислении схематично приведён на рисунке 1.2. Для программы *prog* формулируется предусловие и постусловие на языке L и строится тройка Хоара. По правилам вывода и аксиомам расширения $\mathfrak{H}$ исходная тройка Хоара преобразуется в формулы (на языке L) аксиоматической теории $\mathfrak{T}$. Строится вывод этих формул из аксиом теории $\mathfrak{T}$ с использованием правил вывода теории $\mathfrak{T}$. Если такой вывод удаётся построить, то формулы принадлежат теории $\mathfrak{T}$ и являются тождественно истинными, поэтому истинна и тройка Хоара программы *prog*, откуда следует, что программа корректна.

Рисунок 1.2 — Схема доказательства корректности тройки Хоара

Типизированное лямбда-исчисление. *Лямбда-исчисление* (λ-исчисление) — формальная система, разработанная Алонзо Чёрчем для формализации и анализа понятия вычислимости [33]. Лямбда-исчисление рассматривает функции как правила. Термы теории строятся из переменных с помощью операций абстракции и аппликации, а формулы теории — равенства между λ-термами [35]. Особую роль в программировании и доказательстве корректности программ играет *типизированное λ-исчисление*. Это версия λ-исчисления, в которой λ-термам приписываются специальные синтаксические метки, называемые типами.

Хенк Барендрегт классифицировал типизированные λ-исчисления и предложил идею лямбда-куба [34]. *Лямбда-куб* (λ-куб) — наглядная классификация восьми типизированных λ-исчислений с явным приписыванием типов. Каждое измерение в λ-кубе соответствует одному из вариантов зависимости между типами и термами. В наиболее бедной вершине куба находится просто типизированное λ-исчисление (в котором термы зависят от термов, а типы в зависимостях не участвуют), а в наиболее богатой — исчисление конструкций [39].

В данной классификации используется понятие *зависимого типа* (dependent type), то есть типа, который зависит от терма или другого типа. Это понятие было введено Мартином-Лёфом в интуиционистской теории типов [40]. Например, тип, описывающий n-кортежи действительных чисел, является зависимым, так как он «зависит» от величины n. Функция, чья область значений варьируется в зависимости от её аргумента, называется *зависимой функ-*

цией (dependent function). Тип этой функции называется *зависимым произведением типов* (тип зависимого произведения, dependent product type), пи-типом или просто зависимым типом (dependent function type). Записывается зависимый тип как $\Pi\, x : A.\, B(x)$, где $A : U$ — тип из вселенной типов U, B — семейство типов, сопоставляющее каждому терму $a : A$ тип $B(a) : U$ (в данном случае, символ «:» имеет тот же смысл, что и символ «$\in$» и обозначает принадлежность к типу). Зависимый тип может рассматриваться как декартово произведение типов.

Между системами λ-куба и логическими системами (исчисление высказываний, многосортное исчисление предикатов первого, второго и высшего порядков) существует изоморфизм, его называют изоморфизмом или соответствием Карри-Ховарда [34, 41]. Теория типов и изоморфизм Карри-Ховарда используются для верификации программ. С одной стороны, термы и типы λ-исчисления рассматриваются как программы и спецификации программ соответственно. С другой стороны, тип можно рассматривать как высказывания (формулы), а терм — как доказательство этого высказывания. Таким образом, наблюдается структурная эквивалентность между математическими доказательствами (записанными в форме естественного вывода) и программами. Когда тип соответствует высказыванию, то для доказательства этого высказывания необходимо построить терм этого типа. Если это удаётся сделать, то про тип говорят, что он обитаем или населён, и это доказывает, что высказывание является теоремой.

Соответствие Карри-Ховарда позволило создать целый класс функциональных языков программирования, среда выполнения которых одновременно является системой автоматического доказательства, таких как Coq [41] и Agda [42] (см. раздел 1.3.8).

Уточнение программ. *Уточнение программ* (program refinement) — это методология программирования, в которой формальное описание того, что должна делать программа (спецификация) постепенно трансформируется (уточняется) в исполняемую программу, удовлетворяющую этой спецификации [5] (рисунок 1.1б).

Выделяют два направления уточнения программ, в зависимости от того, как рассматривается спецификация [43]: спецификация рассматривается как тип в некоторой выразительной теории типов [44], или спецификация — это выражение (высказывание) программной логики со свободными переменными на языке предикатов первого порядка [6].

В первом случае используется уточнение данных. *Уточнение данных* (data refinement) — преобразование абстрактной модели данных в реализуемые структуры данных. В работе [44] уточнение данных осуществляется с помощью расширенного исчисления конструкций (Extended Calculus of Constructions) [45]. При уточнении данных, программа, написанная с использованием абстрактных типов данных (например, стека), переписывается

так, чтобы использовать более конкретный тип (например, массив). Разработка корректных программ с помощью поэтапного уточнения (stepwise refinement) заключается в построении уточняющих отображений (refinement maps), сохраняющих корректность программы. Отображение называется абстрактной реализацией (abstract implementation) и строится между типом реализующей спецификации и типом исходной абстрактной спецификации.

Второй вариант — это «классическое» *уточняющее исчисление*, которое является формализацией и расширением метода слабейшего предусловия, предложенного Дейкстрой [46, 47]. *Слабейшее предусловие* для программы `Prog` и любой формулы (предиката) Q, описывающей ожидаемый результат выполнения программы `Prog`, — это формула $wp(\texttt{Prog}, Q)$, которая представляет множество всех состояний, для которых выполнение `Prog`, начавшееся в таком состоянии, обязательно закончится через конечное время в состоянии, удовлетворяющем Q. То есть выполнено $\{wp(\texttt{Prog}, Q)\}\ \texttt{Prog}\ \{Q\}$.

Уточняющее исчисление позволяет вывести императивную программу из спецификации [6, 48]. Например, в работе [6] считается, что не существует различия между спецификацией и программой (программным кодом), всё это программы. Уточнение в данном случае — это отношение между программами: для любых двух программ $prog_1$ и $prog_2$, говорят, что $prog_1$ уточняется программой $prog_2$, если для любого постусловия A имеем $wp(prog_1, A) \Rightarrow wp(prog_2, A)$, где $wp(prog, A)$ — слабейшее предусловие, обеспечивающее завершение в состоянии, описанном A.

Недостаток первого подхода к уточнению программ заключается в том, что он подходит только для интуиционистской логики. Второй подход также имеет ряд недостатков, основной из которых — недостаточная структурированность спецификации, которая необходима для объединения верификации и разработки программы [43]. Поэтому в [5, 43] предложен третий подход, названный уточняющими типами (refinement types), в котором логика программы сочетается с теорией типов языка программирования. *Уточняющий тип* (refinement type) — это тип с приписанным предикатом, который принимает истинное значение для всех элементов этого уточняемого типа [43, 49]. В работе [5] строится уточняющее исчисление (refinement calculus), в котором неопределённые термы имеют уточняющие типы и последовательными преобразованиями по правилам вывода уточняются до программного кода.

1.2.4 Применимость формальных методов к верификации ФПП программ

Основная особенность функционально-потоковой парадигмы параллельного программирования — возможность писать программы с неограниченным параллелизмом без учета

ресурсных ограничений. При этом саму ФПП программу можно рассматривать как спецификацию для программ реальных архитектур. Классические языки спецификации, используемые для описания логики функционирования программы, не позволяют описать параллельность выполнения операций. Поэтому методы конструктивного подхода, а именно различные варианты уточнения программ, плохо подходят для верификации ФПП программ. Если изначально спецификация формулируется на языке, не отражающем параллельность выполнения операций в описываемом алгоритме, то при преобразовании спецификации в параллельную программу потребуется вносить в неё данную информацию. Это означает, что, во-первых, необходимо разработать правила распараллеливания спецификации. И, во-вторых, от разработчика потребуется продумывание механизмов применения данных правил для распараллеливания изначально последовательного алгоритма. В результате теряется наглядность и простота написания параллельной программы, что в корне противоречит ФПП парадигме.

Так как нет простого способа описать параллельность ФПП программы с помощью спецификации, то для верификации ФПП программ будет рассматриваться применение аналитических методов верификации. В этом случае считается, что ФПП программа уже написана. А для её верификации необходимо задать формальную спецификацию и провести сравнение этой спецификации с программой.

Рассмотрим основные характеристики, которыми должен обладать метод верификации ФПП программ. Наиболее важной характеристикой является возможность использования языка спецификации, позволяющего формулировать достаточно сложные утверждения, полностью описывающие логику функционирования программы. Например, недостаточно просто указывать допустимые типы, которые может иметь результат, или формулировать инвариантные утверждения о корректности отдельных фрагментов программы. Из требования выразительности языка спецификации вытекает то, что разрешимость разрабатываемой логической системы в общем случае является недостижимой характеристикой. Под *разрешимостью* понимается существование универсального алгоритма решения поставленной проблемы [50], то есть, в данном случае, алгоритма автоматического доказательства корректности программы. К другим значимым характеристикам относится применимость к широкому классу задач, простота метода и возможность автоматизации процесса доказательства.

Результат сравнения данных характеристик для метода проверки моделей и дедуктивного анализа приведён в таблице 1.1.

Можно сделать следующие выводы. Дедуктивный анализ сложнее, чем метод проверки моделей, но он применим к большему классу задач. Возможность использования произвольного языка спецификации в дедуктивном анализе удовлетворяет главному требованию

Таблица 1.1 — Сравнение методов проверки моделей и дедуктивного анализа

	Проверка моделей	**Дедуктивный анализ**
1	Применяется для верификации систем с конечным числом состояний.	Может быть применим к системам с бесконечным числом состояний.
2	Проверка может быть осуществлена полностью автоматически. Можно использовать для незавершающихся программ.	Большинство систем построения доказательств не могут быть полностью автоматизированы из-за проблемы разрешимости формальных аксиоматических теорий. В частности, правильность завершения работы программы в общем случае нельзя проверить автоматически.
3	Существует проблема «комбинаторного взрыва» в пространстве состояний, что увеличивает время анализа или делает его невозможным из-за ограничений ресурсов.	Анализ требует много времени и может быть осуществлен только экспертами, обладающими знаниями в области логического вывода и имеющими практический опыт.
4	Применим для недетерминированных и параллельных программ, при этом усложняется этап построения модели.	Применим для недетерминированных и параллельных программ, при этом усложняется модель и процесс доказательства.

выбора метода. Преимуществом же метода проверки моделей является возможность автоматического доказательства корректности программы.

В данной работе для верификации ФПП программ будет использоваться более универсальный метод — метод дедуктивного анализа. Из всех методов данной группы наиболее подходящим для верификации ФПП программ является метод, основанный на исчислении Хоара. Методы, использующие изоморфизм Карри-Ховарда, менее подходят для верификации ФПП программ, так как они предполагают, что логика функционирования программы должна быть переформулирована на языке λ-исчисления. Это требует разработки (и верификации) правил преобразования ФПП-программы на язык λ-исчисления, и последующее доказательства принадлежности этой программы типу, описывающему спецификацию, также сформулированную на языке λ-исчисления. Это значительно сложнее, чем простое преобразование одной заданной тройки Хоара к формулам на языке спецификации. Однако язык λ-исчисления достаточно выразителен и хорошо подходит для спецификации свойств функциональных программ, которыми и являются ФПП программы, потому он может быть использован в качестве языка спецификации свойств ФПП программ.

1.2.5 Методы доказательства завершения программ

В ФПП программах могут присутствовать ошибки, приводящие к зацикливанию из-за вызова бесконечной рекурсии. В результате таких ошибок программа не завершается.

Выбранный метод верификации ФПП программ на основе исчисления Хоара позволяет доказать только частичную корректность программы, так как тройки не позволяют выразить свойство завершения. Поэтому необходимо выбрать метод для доказательства завершения ФПП программ.

Доказательство завершения программы, хоть и является частью доказательства тотальной корректности программы, часто рассматривается как самостоятельная задача (можно отнести к задачам верификации динамических свойств программы), без привязки к верификации логики функционирования программы. В общем случае формальный анализ завершения представляет весьма сложную проблему, так как задача является неразрешимой.

Основной причиной незавершения программы является её зацикливание в результате выполнения бесконечной итерации (цикла) или рекурсии (рекурсивной функции) [51, 52]. Ввиду того, что множество вычислимых функций эквивалентно множеству рекурсивных функций, любой итерационный процесс может быть заменен на рекурсивный и наоборот [53, 54]. Отсюда, методы доказательства завершения циклов и рекурсий имеют одинаковую суть. Поэтому можно рассматривать применение обоих типов методов к доказательству завершения ФПП программ.

В основе любого метода доказательства завершения программы лежит применение *трансфинитной индукции* (индукции на фундированном множестве). Задаётся некоторое *фундированное множество* S — частично упорядоченное множество, любое непустое подмножество которого имеет минимальный элемент [31]. С каждой циклической конструкцией (циклом или рекурсивной функцией) связывается частичная функция, отображающая значения переменных программы в элементы фундированного множества. Эту функцию называют ограничивающей [1, 46, 47], функцией декремента [36], мерой (maesure function) [55, 56], функцией завершения (termination function) или сходимости (convergence function) [58]. В данной работе используется термин *ограничивающая функция*. Если при каждом последующем повторении циклической конструкции происходит уменьшение значения ограничивающей функции, то программа завершается.

Самыми простыми методами логического анализа завершения циклических вычислений для императивных программ являются метод Флойда и метод счётчиков.

Метод Флойда. Этот метод может рассматриваться как дальнейшее развитие метода индуктивных утверждений, также предложенного Флойдом [28]. Основная идея метода состоит в том, что с каждой контрольной точкой программы, лежащей на циклическом пути, связывается ограничивающая функция. Она определена на значениях переменных программы и является верхней границей числа шагов, которые должны быть выполнены в дальнейшем. При каждом шаге цикла значение функции должно уменьшаться, по крайней мере,

на единицу, до тех пор, пока выполнение цикла ещё не завершилось. Поскольку функция ограничена снизу, то цикл должен завершиться.

Метод счётчиков. Идея этого метода состоит во введении в программу новых переменных-счётчиков, добавляемых по одной в каждый цикл программы [1]. Переменная-счётчик должна инициализироваться перед входом в цикл и увеличивать своё значение при каждом прохождении по циклу. В преобразованной таким образом программе изменённые инварианты циклов представляются в виде, содержащем условие ограниченности значений счётчиков функциями, которые зависят только от входных переменных. Это указывает на существование верхней грани значений счётчиков. Для преобразованной программы доказательство частичной корректности одновременно характеризует завершение программы, то есть ограниченность значений счётчиков. Однако по сути ограничивающая функция и переменные-счётчики эквивалентны.

Методы доказательства завершения рекурсивных функциональных программ. Доказательство завершения функциональной программы сводится к построению порядка полной фундированности на множестве допустимых аргументов функции и проверке того, что аргумент каждого рекурсивного вызова меньше, чем входной аргумент [59, 57]. Порядок $\succ$ является *отношением полной фундированности*, если не существует бесконечной убывающей цепи $x_1 \succ x_2 \succ \ldots$

Существуют различные методы доказательства завершения рекурсивных функций: принцип изменения размера (size-change principle) [60, 61]; аннотирование типов специальными индексами (size indices) и последующей проверкой того, что аргументы рекурсивного вызова имеют меньшие индексы [6, 62, 63]; использование зависимых типов [12, 64].

Предложено множество подходов по автоматизации процесса доказательства. Одной из основополагающих работ в данной области является работа Бойера и Мура [55]. В системе NQTHM они реализовали метод автоматизированного доказательства завершения функций, написанных на языке LISP. Метод доказательства Бойера и Мура очень эффективный, так как работает с произвольной ограничивающей функцией. Однако у него низкий уровень автоматизации ввиду того, что пользователь должен иметь представление о том, почему алгоритм завершается, и сформулировать это системе в виде индукционных лемм. Также для обеспечения надёжности (soundness) эти леммы должны быть проверены (верифицированы) системой, что может быть трудоёмко и обычно требует доказательства с помощью индукции [56].

Альтернативные методы доказательства предложены в работах [58, 65, 66]. Они являются полностью автоматическими, но используют одно фиксированное упорядочение, поэтому класс решаемых этими методами задач ограничен. Так, метод Вальтера позволяет авто-

матически генерировать индукционные леммы для некоторого класса задач, а надёжность этих лемм вытекает из их построения. Ограничением в методе Вальтера является использование одной фиксированной ограничивающей функции, которая учитывает «размер» объектов (например, стеки сравниваются по глубине, списки по длине, а деревья по количеству узлов). Гизль расширил метод Вальтера так, чтобы можно было использовать произвольную ограничивающую функцию, удовлетворяющую определённым требованиям [56]. Для этого используются методы теории переписывания термов, позволяющие автоматически доказать полиномиальные неравенства (polynomial inequalities), в качестве ограничивающей функции используется норма полинома (polynomial norm) [67].

В последующих работах, при доказательстве завершения функциональных программ, стали активно применяться достижения теории переписывания термов: различные методы автоматической генерации упорядочений термов (term orderings) [59]. Серьёзным прорывом в теории переписывания термов стало появление критерия зависимых пар (dependency pairs) [68], это позволило доказывать завершение большего класса задач [69]. Разработанные методы переписывания термов легли в основу инструментальных средств автоматизированного доказательства завершения программ, в частности системы AProVE [70, 71, 72].

В целом у всех рассмотренных методов одна основа, поэтому в выборе метода доказательства завершения учитываем особенности ФПП программ, используемый метод доказательства частичной корректности и возможность последующей модификации и автоматизации метода.

Отличительная черта ФПП программ состоит в возможности написания программ не только с последовательной рекурсией, но и параллельной. Под *параллельной рекурсией* в данном случае понимается функция f, в которой содержится вызов некоторой функции g, не менее двух аргументов которой являются рекурсивными вызовами функции f [73]. Учитывая то, что при доказательстве частичной корректности используется исчисление Хоара, выберем метод, позволяющей расширить исчисления Хоара так, чтобы доказательство корректности тройки Хоара являлось доказательством тотальной корректности программы. В дальнейшем это позволит строить инструментальные средства на общей базе. Для того, чтобы сосредоточиться на исследовании доказательства тотальной корректности ФПП программ, задачу выбора ограничивающей функции оставим за пользователем. Это значительно упростит метод, а в дальнейшем такой подход может быть модифицирован за счёт автоматизированного выбора некоторого подходящего типа ограничивающей функции, позволяющего автоматически формировать условие завершения и даже его доказательство, или заменён на метод другого типа, например, основанный на переписывании термов. Таким образом, если пользователь предоставляет ограничивающую функцию, то при использовании

λ-исчисления в качестве языка спецификации, определение ограничивающей функции можно добавить в предусловии программы, а в постусловие добавляется условие уменьшения значения ограничивающей функции при каждом последующем рекурсивном вызове. Тем самым доказательство истинности тройки Хоара будет доказательством тотальной корректности программы.

1.3 Инструментальная поддержка доказательства корректности программ

Для того, чтобы разработать систему инструментальной поддержки доказательства корректности ФПП программ, рассмотрим и сравним существующие средства поддержки доказательства корректности программ.

1.3.1 Уточняющие типы

Современные функциональные языки программирования, такие как OCaml и Haskell, имеют строгую статическую типизацию. Основателем данного направления, в котором впервые была реализована система типов, считается язык ML [74]. Для увеличения выразительности системы типов предложено использовать зависимые типы [49, 75]. Для сохранения возможности автоматического вывода типов при использовании уточняемых типов в работе [11] предлагается использовать ликвид-типы (liquid-types — аббревиатура от английского **L**og**i**cally **Qu**alified **D**ata Types — типы данных с логическими квалификаторами или **лог**и**ч**ески **ква**л**и**фицированные типы **д**анных). Это система для автоматического вывода зависимых типов, достаточная для доказательства различных свойств надёжности, не требующая значительного ручного аннотирования программы.

Система получает на вход программу и набор логических квалификаторов [11] и делает вывод ликвид-типов. Алгоритм, по которому работает система, состоит из трёх этапов. На первом этапе используется вывод Хиндли-Милнера [76]. На втором этапе происходит генерация ликвид-ограничений. На третьем этапе проводится решение ликвид-ограничений [77]. Однако существуют корректные программы, которые нельзя правильно типизировать в системе либо из-за того, что неправильно выбраны квалификаторы, либо ввиду ограниченности определения подтипов [78].

На основе ликвид-типов разработано несколько инструментальных средств для верификации функциональных программ. Например, DSolve для программ на языке OCaml [11, 27], HSolve (LiquidHaskell) для программ на языке Haskell [12, 79]. Специальные ликвид-типы низкого уровня (low-level liquid types) используются для верификации императивных программ. Например, система CSolve разработана для программ на языке Си [27, 80].

Все системы автоматической верификации на основе ликвид-типов имеют схожий ин-

терфейс. Они принимают на вход программу и множество квалификаторов и выдают ответ, удовлетворяет программа условиям корректности или нет. Также они возвращают зависимые типы и множество примитивных операций для которых статически не может быть доказана корректность (в идеале это множество пустое).

В качестве примера на рисунке 1.3 приведена схема работы системы LiquidHaskell.

Рисунок 1.3 — Схема работы системы LiquidHaskell

Система за раз верифицирует содержимое одного файла. Вначале исходный код на языке Haskell передаётся GHC (Glasgow Haskell Compiler) [81], который переводит программу в промежуточное представление. На втором этапе программа, сигнатуры типов для функций модуля (которые должны быть верифицированы) и сигнатуры типов внешних функций (которые считаются верными) передаются генератору ограничений, который генерирует ограничения для подтипов. Полученные ограничения упрощаются до простых логических импликаций. На последнем этапе эти ограничения, вместе с логическими квалификаторами, предоставленными пользователем и извлечённые из сигнатур типов, передаются SMT-решателю, который определяет, являются ли они выполнимымы.

Haskell отличается от других функциональных языков тем, что в нём есть ленивые вычисления, и он работает в режиме вызова по необходимости (call-by-need). Поэтому для верификации программ используется расслоённая система типов (stratified type system), которая помечает связи как потенциально сходящиеся или нет. Понятие сходимости в данном случае связано с завершением функции. При этом уточняющие типы могут использоваться для доказательства завершения функции [12].

1.3.2 Контрактное программирование

Термин «контрактное программирование» (Design by Contract) предложил Бертран Мейер при разработке языка объектно-ориентированного программирования Eiffel [82]. Это один из методов проектирования программного обеспечения, цель которого — повышение надёжности (reliability) [23, 25]. Данный метод предполагает, что проектировщик должен определить формальные, точные и верифицируемые спецификации интерфейсов для компонентов системы. При этом используются предусловия, постусловия и инварианты. Данные

спецификации называются «контрактами».

Средства для контрактного программирования поддерживают многие языки программирования, например Eiffel, Spec#, C#, Java [25]. Для программ на языке Eiffel разработана автоматическая система верификации, которая может доказывать корректность уже написанного кода без необходимости добавления дополнительных утверждений. Инструментальное средство EVE Proofs [83] (в последствие AutoProve [84]) — статический верификатор (static verifier) для программ на Eiffel. Он транслирует программы на Eiffel в программы для Boogie (см. раздел 1.3.3) и запускает полностью автоматическое средство доказательства теорем для проверки корректности кода. Схема работы системы приведена на рисунке 1.4.

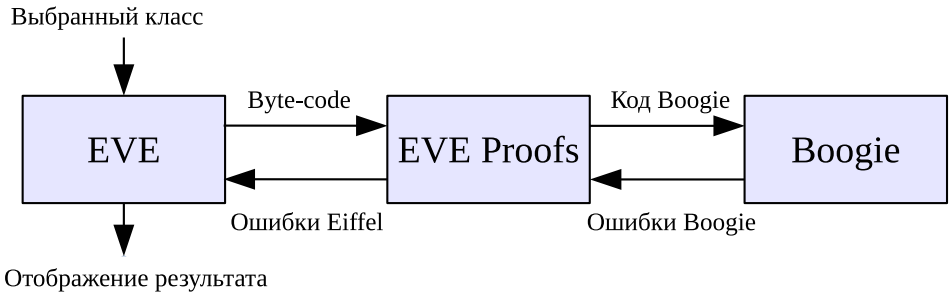


Рисунок 1.4 — Процесс доказательства с помощью EVE Proofs

При доказательстве корректности больших систем используется модульный принцип, который заключается в том, что корректность различных частей доказывается независимо. В EVE Proofs корректность доказывается независимо для каждого метода класса. EVE Proofs интегрирован в EVE (Eiffel Verification Environment) — исследовательское ответвление интегрированной среды разработки EiffelStudio [85].

В контрактном программировании также рассматриваются вопросы параллельных вычислений. SCOOP (Simple Concurrent Object-Oriented Programming) — объектно-ориентированная модель многопоточности [25, 86, 87, 88].

1.3.3 Верификатор Boogie

Изначально верификатор Boogie разрабатывался для верификации программ Spec# в среде разработки .NET. Верификатор использует язык BoogiePL [89] (в последствие названный «язык Boogie» [90]) — промежуточный язык, разработанный для описания условий корректности императивных и объектно-ориентированных программ. Внутри Boogie имеет структуру конвейера, совершающего серию преобразований от исходного кода программы к условиям корректности и сообщениям об ошибке. Основные элементы конвейера приведены на рисунке 1.5.

Входным языком системы является Spec# — расширение языка C# посредством кон-

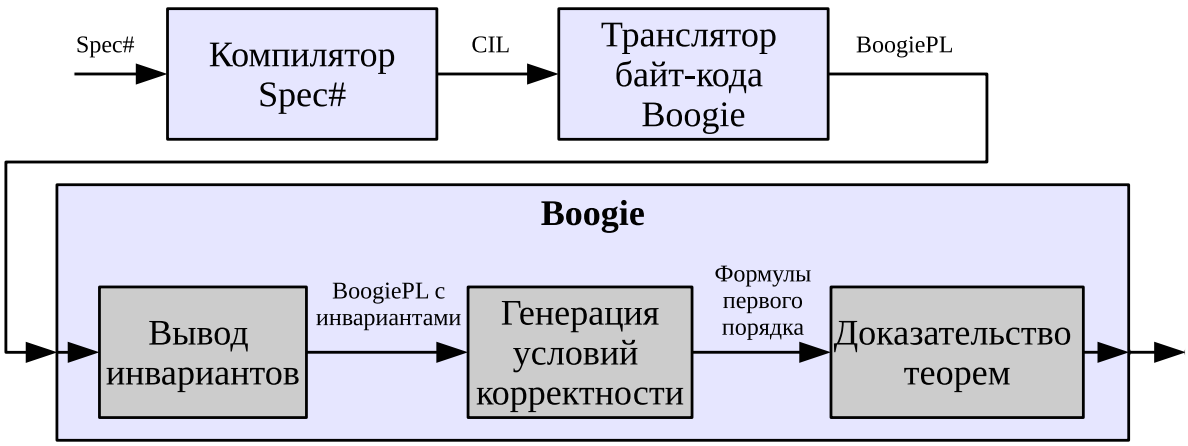


Рисунок 1.5 — Конвейер Boogie

трактов [23]. Компилятор Spec# транслирует код в CIL (Common Intermediate Language) — «высокоуровневый ассемблер» виртуальной машины .NET. На следующем этапе представление CIL переводится на язык BoogiePL [89]. Для этого в условиях корректности требуются инварианты циклов. Инварианты могут быть написаны пользователем на BoogiePL или Spec#, а могут выводиться из исходного кода программы. После этапа вывода инвариантов Boogie генерирует условия корректности, которые являются формулами на языке логики первого порядка и арифметики. При генерации используется техника построения слабейших предусловий [46].

Формулы передаются системе автоматического поиска доказательства для определения их истинности, и следовательно, корректности программы. Boogie использует систему доказательства теорем Simplify [91]. Если система доказательства теорем не может доказать истинность формулы, то это свидетельствует либо о наличие ошибки в программе, либо об отсутствии какого-либо условия в контракте (ошибка в контракте). Также ввиду неполноты (incompleteness) процесса доказательства сообщение об ошибке может быть ложным, если система поиска доказательства вышла за рамки имеющихся ресурсов, таких как время или дисковое пространство.

Отделение верификатора Boogie от транслятора и генератора байт-кода посредством промежуточного языка позволяет использовать его для верификации программ на других языках программирования. Для этого достаточно построить транслятор с исходного языка на язык BoogiePL. В результате верификатор Boogie используется также при верификации программ на языках: C, Dafny, байт-кода Java с помощью BML (Bytecode level Specification Language) и Eiffel [90].

1.3.4 Верификация Java-программ со спецификацией на JML

JML (Java Modeling Language) [92] — это язык спецификации поведения интерфейсов (BISL, Behavioral Interface Specification Language), который может использоваться для спецификации поведения модулей Java. Язык объединяет в себе подход проектирования по контракту Eiffel (см. раздел 1.3.2) и модульную спецификацию языков семейства Larch [93], а также содержит элементы уточняющего исчисления (см. раздел 1.2.3). Языки спецификации поведения интерфейсов позволяют писать формальные аннотации на уровне кода. К ним относятся предусловия, постусловия, инварианты и утверждения, которые позволяют выражать предполагаемое поведение программных модулей [94, 95]. В JML используется синтаксис языка Java для записи предикатов в утверждениях, а также расширения языка Java с помощью различных спецификационных конструкций, например, кванторов [96, 97, 98].

JML разработан так, чтобы использоваться с широким набором инструментов. Инструменты, проверяющие, что JML-аннотированные Java-программы соответствуют своей спецификации, можно разделить на две группы: инструменты проверки утверждений во время выполнения и инструменты статической верификации. При статической верификации производится анализ кода без запуска программы, для этого используется техника логических доказательств. К инструментам верификации программ, поддерживающим JML, относятся: KeY [99, 100], Krakatoa [101, 102], LOOP [9, 103], ESC/Java2 [104].

ESC/Java (Extended Static Checker for Java) — система расширенной статической проверки Java-программ, использующая JML. Эта система пытается при компиляции найти распространённые ошибки, возникающие при выполнении программы. Данный подход называется расширенной статической проверкой и может рассматриваться как расширенная форма проверки типов. Для такой проверки в ESC/Java используется автоматическое средство доказательства теорем Simplify. Система ESC/Java не является ни надёжной, ни полной. Система OpenJML [105] является преемником ESC/Java.

Krakatoa — инструмент статической верификации, основанный на платформе для верификации Why [106], использующий систему поддержки доказательства Coq.

Система **LOOP** принимает программу и контракт, написанный на JML, и использует интерактивную систему доказательства теорем PVS, для которой она генерирует условия корректности, похожие на тройки Хоара. Она также обеспечивает некоторую автоматизацию с помощью тактики слабейшего предусловия.

Система **KeY** используется для формальной верификации Java-программ. Это интегрированная, автономная система, для работы которой не нужны внешние компоненты. Общий процесс доказательства с использованием системы KeY показан на рисунке 1.6. Поль-

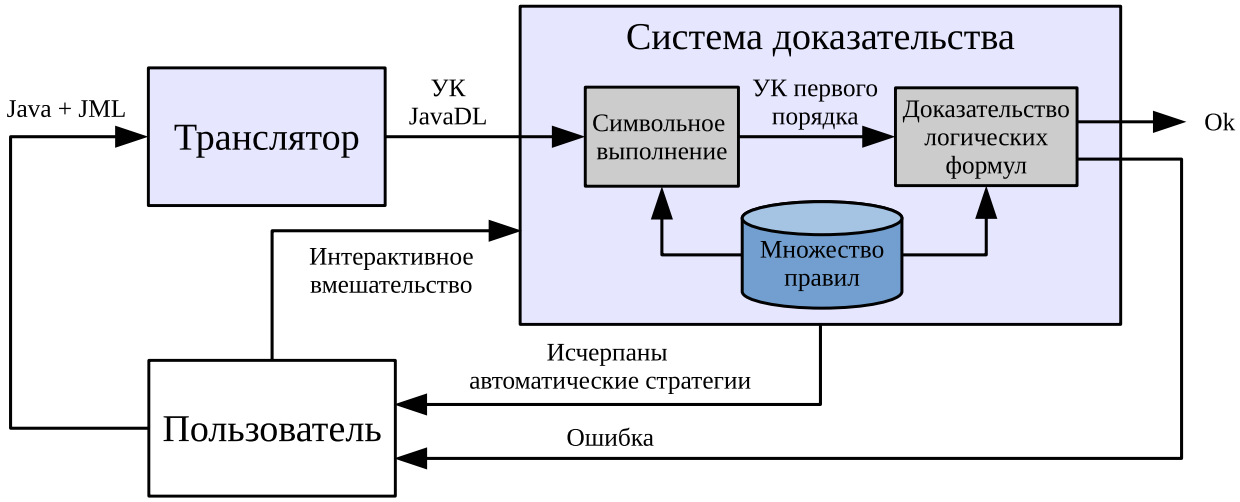


Рисунок 1.6 — Схема процесса верификации в системе KeY

зователь предоставляет системе исходный код Java с аннотациями, написанными на JML. Из полученной программы и спецификации система автоматически формирует условия корректности (УК) на языке Java Dynamic Logic (JavaDL) — версии динамической логики первого порядка, адаптированной для программ на Java Card (диалект Java для мобильных устройств).

Динамическая логика может рассматриваться как расширение логики Хоара. Особенностью KeY является то, что формулы включают исходный код программы, а правила вывода представлены в виде так называемых таклетов (taclets), которые являются схемами правил типизированной логики первого порядка, сформулированными на простом предметно-ориентированном языке. В основе системы лежит эффективный интерпретатор, который применяет таклеты к последовательностям доказываемых целей и тем самым создает деревья доказательств [99]. Правила вывода для программных формул в KeY разработаны таким образом, чтобы осуществлять символьное выполнение программы, которое проходит программу начиная от входных данных (прямое прослеживание). Совместно с принципом индукции (инвариантами циклов), символьное выполнение программы позволяет полностью верифицировать программу: свести любую корректную программу к конечному набору истинных формул логики первого порядка с арифметикой. Однако при верификации реальных программ число ветвей в дереве символического выполнения растет экспоненциально с количеством точек ветвления в программе. В этом случае применяется принцип модульности.

1.3.5 Средства верификации C-программ

Существует несколько проектов по верификации C-программ.

Проект **Verisoft** [107], ориентирован в основном на встраиваемые системы. Одна из

целей проекта — верификация ядра операционной системы для простого, верифицированного процессора. Используется простое подмножество языка C — язык C0, семантика которого моделируется в системе доказательства теорем Isabelle/HOL.

Why [106, 108] — это платформа, предназначенная для верификации многих императивных языков. В ней определён промежуточный язык под тем же названием Why, в который транслируются верифицируемые программы. Цель трансляции — генерация условий корректности в виде, не зависящем от системы доказательства теорем. Платформа Why служит основой системы Frama-C, которая предоставляет статический анализ для полного языка C и дедуктивную верификацию для ограниченного подмножества.

В проекте **VCC** (A Verifier for Concurrent C) [109] аннотированные программы транслируются в логические формулы с помощью верификатора Boogie.

Система верификации **CSolve** основана на уточняющих типах (см. раздел 1.3.1).

Мультиязыковая система СПЕКТР. Цель проекта СПЕКТР — разработка нового подхода к верификации императивных программ, который позволяет интегрировать, унифицировать и комбинировать методы и техники верификации императивных программ, накапливать и использовать знания о них [8]. Система СПЕКТР является мультиязыковой системой анализа и верификации программ. В настоящее время она применяется для верификации C-программ, но открытая архитектура системы обеспечивает добавление новых языков программирования.

Основной цикл работы СПЕКТР состоит в последовательном выполнении заданий, поступающих от пользователя, анализатором программных моделей (АПМ), и возвращении результатов анализа обратно пользователю (рисунок 1.7). Задание содержит аннотированную программу и спецификацию анализа на языке Atoment [8], определяющую применяемые к ней техники анализа и верификации. Перед входом в АПМ аннотированная программа преобразуется построителем программных моделей в программную модель (ПМ). ПМ является некоторым внутренним представлением аннотированной программы в системе СПЕКТР, доступ к которому осуществляется через конструкции языка Atoment. Использование ПМ позволяет унифицировать формат данных для АПМ. Аннотированные программы, использующие различные языки программирования и различные языки аннотации, приводятся к единому формату ПМ. Это обеспечивает мультиязыковость системы СПЕКТР. Перед возвращением пользователю результаты анализа обрабатываются интерпретатором результатов анализа.

Логика работы АПМ основана на последовательных трансформациях ПМ. Для выполнения трансформаций АПМ использует интерпретатор трансформаций, подавая ему на вход ПМ и спецификацию трансформации (составную часть спецификации анализа) для неё.

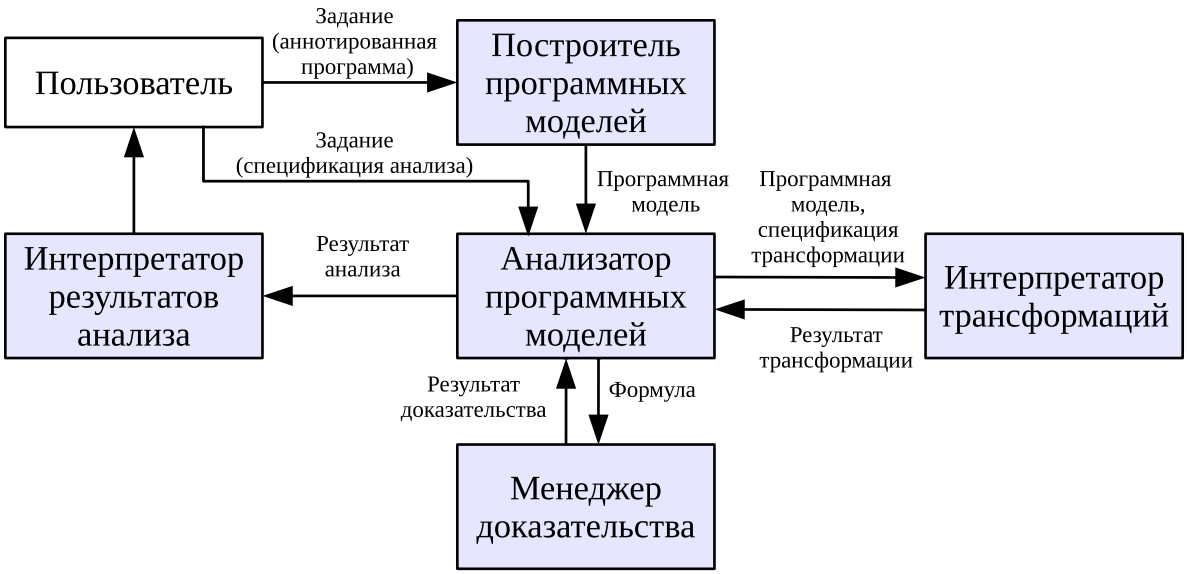


Рисунок 1.7 — Архитектура системы СПЕКТР

Интерпретатор трансформаций возвращает результат трансформации [110].

Для доказательства формул АПМ использует менеджер доказательства, подавая ему на вход формулы, появляющиеся в результате трансформаций. В версии системы из работы [8] поддерживается адаптер к решателю Z3. Менеджер возвращает результат доказательства формулы, включающий статус доказательства со значениями «истинна», «ложна», «не доказана» и, возможно, контрпример. Контрпример вместе с обратными зависимостями используется интерпретатором результатов анализа.

1.3.6 Предикатное программирование

Шелеховым В.И. разрабатывается язык предикатного программирования Р [111, 112], который позволяет сопоставить каждой исполнимой конструкции языка формулу на языке исчисления предикатов. Это позволяет доказывать корректность предикатных программ с помощью метода дедуктивной верификации [113]. Язык предикатного программирования Р относится к классу языков функционального программирования, при этом он сочетает функциональный и операторный (предикатный) стили записи алгоритмов. Язык является универсальным и может использоваться для разработки широкого класса программ [114]. В частности, он может быть расширен средствами для описания процессов, передачи сообщений и порождения процессов, работающих параллельно с процессом-родителем. Спецификация предикатных программ реализуется с помощью предусловия, постусловия и функции меры, используемой для доказательства завершения рекурсий [115]. На базе аппарата формальной операционной семантики для языка Р разработан метод дедуктивной верификации предикатных программ. Система верификации разрабатывалась как back-end в системе пре-

дикатного программирования [113]. Данная система осуществляет трансляцию программы с языка P на язык C++. Синтаксический анализатор производит разбор кода программы на языке P и формирует её образ во внутреннем представлении. Система дедуктивной верификации (рисунок 1.8) включает: генератор формул корректности, транслятор формул во внутреннее представление SMT-решателя CVC3, транслятор формул на язык спецификации системы PVS. Генератор формул корректности строит формулы корректности программы, используя систему правил. Для гарантии корректности программы необходимо проводить полный контроль типов. В трансляторе с языка P реализована статическая проверка совместимости типов. Условия совместимости типов, которые транслятор не может проверить статически, добавляются к формулам корректности программы. Все формулы проходят проверку в SMT-решателе CVC3. Формулы, которые решатель не смог доказать, оформляются в виде теории для системы интерактивного доказательства PVS.

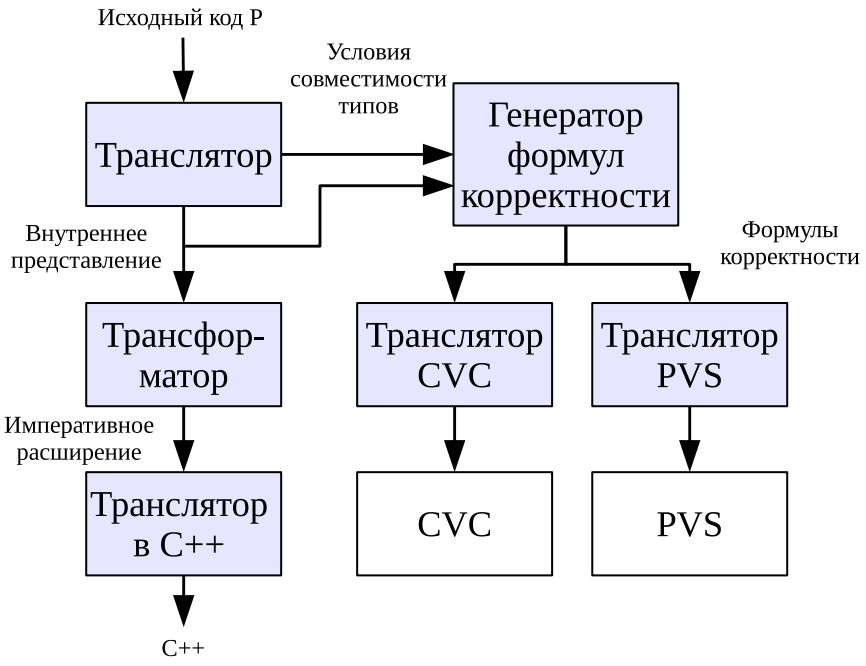


Рисунок 1.8 — Система верификации предикатных программ

1.3.7 Обобщённая схема инструментальных средств и её применимость к ФПП программам

Все рассмотренные системы инструментальной поддержки верификации программ имеют схожую архитектуру, схема которой приведена на рисунке 1.9. На вход системе поступает код программы со спецификацией. Спецификация может передаваться независимо от кода или интегрировано, например, в виде комментариев в файле исходных кодов. Для ФПП программ применимы оба варианта. Также, чтобы упростить процесс доказательства,

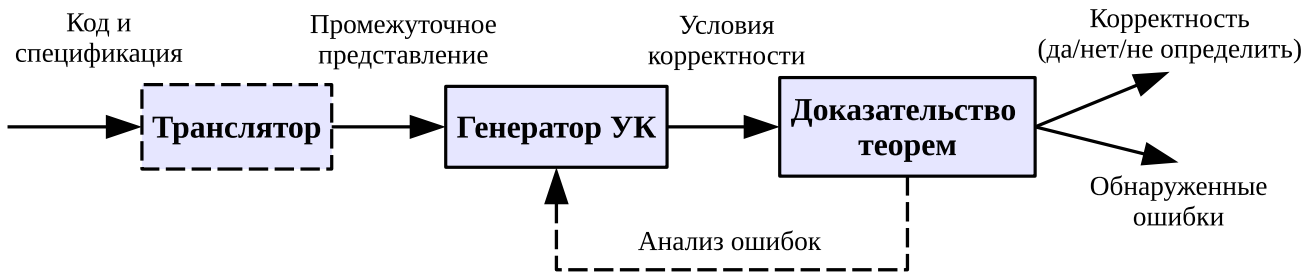


Рисунок 1.9 — Обобщённая схема систем поддержки верификации программ

в системах используется принцип модульности: одномоментно доказывается корректность только одной функции. Используем этот принцип и для верификации ФПП программ.

Модуль трансляции программы в некоторое промежуточное представление присутствует только у некоторых систем, например, работающих с верификатором Boogie. Для ФПП программ не найдено подходящего формата, в который возможно без потерь трансформировать программу, чтобы передать её на верификацию существующей системе доказательства корректности программ, поэтому данный модуль не требуется.

Главным в системе является модуль генерации условий корректности. Этот модуль уникальный у каждой системы и фактически определяется используемым исчислением. Он зависит от таких ключевых характеристик, как используемый язык программирования, язык спецификации, алгоритм и правила преобразования. Данный модуль может работать как автоматически, так и интерактивно. Одна из основных задач данной работы состоит в разработке модуля генерации УК для ФПП программ.

Получаемые от модуля условия корректности передаются модулю доказательства. Практически всегда это сторонний модуль, не связанный с языком программирования и разработанный для доказательства утверждений на определённом языке логики. Порядок логики определяет, автоматически или интерактивно проводятся доказательства формул. Для верификации ФПП программ мы также будем предполагать использование сторонних средств доказательства теорем. Это позволит провести декомпозицию задачи и использовать имеющиеся достижения в области автоматического и автоматизированного доказательства теорем. На данном этапе работы сгенерированные условия корректности передаются пользователю, который может использовать для их доказательства одну из существующих систем.

В результате работы система поддержки верификации программ выдаёт сообщение о корректности программы. В некоторых системах также предполагается наличие обратной связи, позволяющей анализировать причины возникших ошибок.

В целом все рассмотренные системы дедуктивной верификации можно разделить на два типа: автоматические и интерактивные. Автоматические системы требуют от програм-

миста наименьших трудозатрат: достаточно написать спецификацию программы, а доказательство корректности система полностью берёт на себя. Но это не отменяет необходимости анализа и исправления найденных ошибок в программе или спецификации, и итерационного повторения процесса до тех пор, пока все несоответствия программы и спецификации не будут найдены. Однако разрешимость формул корректности достигается за счёт использования простого языка спецификации, который не позволяет сформулировать достаточно сложные утверждения о корректности программы. Поэтому о полной верификации алгоритма речи не идет. Например, для функции сортировки элементов в массиве (для простоты фиксированного размера) можно задать «простую» спецификацию: после сортировки каждый последующий элемент в массиве больше либо равен предыдущему. Но такой спецификации всегда удовлетворяет выходной массив из нулей, вне зависимости от того, какой массив был на входе. Поэтому для «большей» корректности программы в спецификации нужно указать, что в исходном и конечном массиве должны содержатся одни и те же элементы, а различаться может только их порядок. Для формулировки такого утверждения требуется более сложный язык. Поэтому автоматические системы часто используются как помощники при разработке программ, не позволяя программисту допустить многие распространённые ошибки, как, например, ошибки с типами, обращение по индексу за пределами массива и др.

Интерактивные системы имеют более выразительный язык спецификации, который позволяет сформулировать более сложные требования к программе, но и доказательство требует участия программиста, а система помогает ему, например, с выбором правил, автоматическим проведением части доказательства, проверкой правильности доказательства.

Для программ на языке Пифагор необходимо использовать достаточно выразительный язык спецификации и интерактивное доказательство корректности, так как целью является получение библиотеки корректных неограниченно параллельных программ. В данном случае требуется верификация всего алгоритма, а не отдельных утверждений о программе. Однако использование сложного языка всегда позволяет брать его разрешимое подмножество, если это допускает решаемая задача, что позволит проводить автоматические доказательства.

1.3.8 Инструментальная поддержка доказательства теорем

Рассмотрим существующие системы доказательства теорем и выберем наиболее подходящие для доказательства условий корректности ФПП программ. При разработке языка спецификации будем опираться на логики выбранных систем. Это в будущем упростит применение данных систем для доказательства условий корректности. Вместе с тем, не будем делать жесткую привязку к языку какой-либо системы, чтобы не ограничить выразитель-

ность языка спецификации ФПП программ, а также сохранить возможность смены системы интерактивного доказательства.

В целом инструменты построения доказательств можно разделить на две группы.

1. Инструменты автоматического построения доказательств (используемые названия: provers — система поиска доказательства, пруверы; solvers — решатели).

2. Инструменты автоматизации доказательства (используемые названия: automated theorem provers — система автоматизированного доказательства теорем, поиска доказательства; verification system — средства верификации; proof assistant — средства поддержки доказательства; interactive theorem prover — система интерактивного поиска доказательства).

Решатели. Инструменты первой группы работают полностью автоматически: получая формулу, они дают ответ «истинна», «ложна» или «неизвестно». Работа этих систем основана на расширениях пропозициональной логики или логик первого порядка.

Задача выполнимости формул в теориях (satisfiability modulo theories, SMT) — это задача разрешимости для логических формул с учётом лежащих в их основе теорий [116]. Формально, SMT-формула — это формула в логике первого порядка, в которой некоторые функции и предикатные символы имеют дополнительную интерпретацию. Задача состоит в том, чтобы определить, выполнима ли данная формула. SMT-решатели — это программы, которые принимают на входе вопрос, сформулированный в рамках какой-либо теории и требующий ответа «Да» или «Нет», и на выходе формируют ответ. Поддерживаемые теории: булевы числа, целые числа, действительные числа, числа с плавающей точкой, массивы и битовые векторы. Примеры SMT-решателей: CVC3 [118], CVC4 [119], Z3 [120], Yices [121].

Системы интерактивного поиска доказательства. Данные системы помогают проводить формальные доказательства при взаимодействии человека и машины, совмещая в себе преимущества ручного и автоматического подхода [122]. Данные инструменты основаны на логиках высших порядков. В работе [123] проводится сравнение основных систем интерактивного поиска доказательства. Основными отличиями между сравниваемыми системами являются: используемый формализм, уровень достоверности, степень автоматизации, имеющиеся библиотеки, язык написания доказательств, интерфейс пользователя. На основе данного сравнения можно выделить несколько наиболее универсальных и выразительных систем: HOL [124, 125], Isabelle [126, 127], Coq [41, 128], PVS [129, 130].

HOL (Higher Order Logic) — семейство систем интерактивного поиска доказательства, имеющих схожую реализацию и использующих похожую логику высшего порядка [125]. Логика системы HOL является вариантом простой теории типов Чёрча [125], она эквивалентна теории множеств. Системы семейства HOL используют LCF-подход (Logic for computable functions), который заключается в применении метаязыка логики вычислимых функций для

доказательства теорем. Данные системы реализованы как библиотеки на функциональном языке программирования ML (Meta Language). К основным встроенным теориям относятся: логика, арифметика, теории примитивной рекурсии, списков, деревьев. На данный момент разрабатываются четыре системы HOL: HOL4 [124], HOL Light [131], ProofPower [132], HOL Zero [133]. HOL также является предшественником системы Isabelle [126, 127].

Coq — интерактивное программное средство доказательства теорем, использующее собственный язык функционального программирования с зависимыми типами Gallina и язык команд Vernacular [128]. Система позволяет записывать математические утверждения, помогает найти их формальные доказательства, автоматически проверяет доказательства на правильность, а также позволяет сгенерировать корректную программу по формальной спецификации конструктивного доказательства. Теоретической базой Coq является исчисление индуктивных конструкций. Стандартная библиотека включает в себя следующие теории: классическая логика с равенством, арифметика Пеано, списки, множества, действительные числа и др.

PVS (Specification and Verification System) — интерактивная среда для написания формальной спецификации и проверки формальных доказательств [129, 130, 134]. PVS имеет выразительный язык спецификации, который является языком логики высшего порядка, расширенной сложной системой типов, включающей подтипы на основе предикатов (predicate subtypes) и зависимые типы, параметризованные теории и механизм определения абстрактных типов данных. К стандартным типам PVS относятся числа (действительные, рациональные, целые, натуральные, ординальные), записи (records), кортежи (tuples), массивы, функции, множества, последовательности, списки, деревья и т.д. Система типов удобна для написания спецификации, но проверка типов становится неразрешимой задачей. Эта проблема решается с помощью генерации условий корректности и последующем их доказательстве с помощью системы доказательства теорем PVS [135].

Основным критерием выбора языка спецификации ФПП программ является его выразительность, при этом критерий разрешимости становится недостижимым. При выборе системы доказательства условий корректности, полученных для ФПП программ, можно воспользоваться тем же подходом, что и в предикатном программировании (см. раздел 1.3.6).

Истинность простых утверждений доказывается с помощью решателя. В работе [171] рассматривается применение для этих целей решателя CVC4 [119], который использует язык SMT-LIB версии 2.0 [116]. Решатель CVC4 выбран по причине того, что поддерживает все теории SMT-LIB: целые и действительные числа, массивы, битовые векторы.

Для более сложных условий корректности приходится использовать логику высшего порядка и системы интерактивного доказательства. Все рассмотренные системы интерактив-

ного поиска доказательства похожи по функциональности, а основные отличия затрагивают интерфейс. Поэтому при выборе верификатора для системы доказательства корректности программ на языке Пифагор основными критериями является наличие хорошей документации, множества примеров и широта использования. Этим критериям в большей степени удовлетворяет система HOL.

Выводы

Проведён анализ литературных источников, проанализированы существующие подходы к формальной верификации и возможности их использования для ФПП программ. Получены следующие результаты.

1. Классифицированы ошибки, которые могут присутствовать в ФПП программах. Основным преимуществом ФПМПВ является отсутствие ошибок, обусловленных взаимодействием ограниченных ресурсов.

2. Рассмотрены существующие методы формальной верификации программ. Для верификации ФПП программ выбран метод, основанный на исчислении Хоара.

3. Рассмотрены методы доказательства завершения программ. Для ФПП программ выбран метод, расширяющий исчисление Хоара и требующий определения ограничивающей функции.

4. Проведено сравнение существующих систем поддержки формальной верификации программ на основе дедуктивного анализа. Определена общая схема архитектуры данной системы для поддержки формальной верификации ФПП программ.

5. Рассмотрены существующие системы инструментальной поддержки доказательства теорем. Логика системы HOL выбрана как основа для разработки языка спецификации ФПП программ.

2 Аксиоматическая семантика ФПП программ

В главе 2 формализуется семантика языка Пифагор, задаётся язык спецификации свойств ФПП программ. Эти результаты используются для последующего построения аксиоматической теории на базе исчисления Хоара для верификации ФПП программ. Программа на языке Пифагор наглядно представляется в виде информационного графа. Вводится понятие информационного графа с разметкой. Рассматривается возможность отображать на информационном графе с разметкой преобразования программы в процессе доказательства корректности, для этого описываются преобразования, которые претерпевает граф во время доказательства.

2.1 Семантика языка Пифагор

2.1.1 Общие принципы организации модели вычислений

Пифагор является языком функционально-потокового параллельного программирования [16]. В его основе лежит модель функционально-потоковых параллельных вычислений. В данной модели программа — это функция. Каждая функция представляется *информационным графом*, в котором узлами являются программо-формирующие операторы, а дугами — информационные связи между операторами. Любая связь может быть размечена значением, которое одновременно является выходным значением оператора, из которого выходит связь, и входным значением оператора, в который связь направлена. Процесс выполнения программы представляется с помощью разметки её информационного графа. Поступление входных аргументов соответствует начальной разметке графа. Выполнение любого оператора начинается, как только размечены все его входные дуги. После выполнения оператора его выходная дуга помечается получаемым значением. Выполнение программы завершается, когда размеченной оказывается дуга результата.

Существуют следующие типы програмно-формирующих операторов:

1. оператор интерпретации — осуществляет применение функции к аргументу;

2. константный оператор — не имеет входов и имеет только один выход, на котором постоянно находится разметка, определяющая заданное значение константы;

3. оператор копирования данных — обеспечивает размножение данных;

4. оператор группировки в список — обеспечивает структуризацию и упорядочение данных;

5. оператор группировки в параллельный список — осуществляет объединение данных, на выходе получается множественная связь с кратностью, равной количеству входов в оператор;

6. оператор группировки в список задержанных вычислений — операторы, сгруппированные в этот список, будут выполнены только по необходимости, при подаче задержанного списка на вход оператора интерпретации, в этом случае происходит снятие задержки, также называемое раскрытием задержанного списка.

На рисунке 2.1 приведено графическое обозначение операторов.

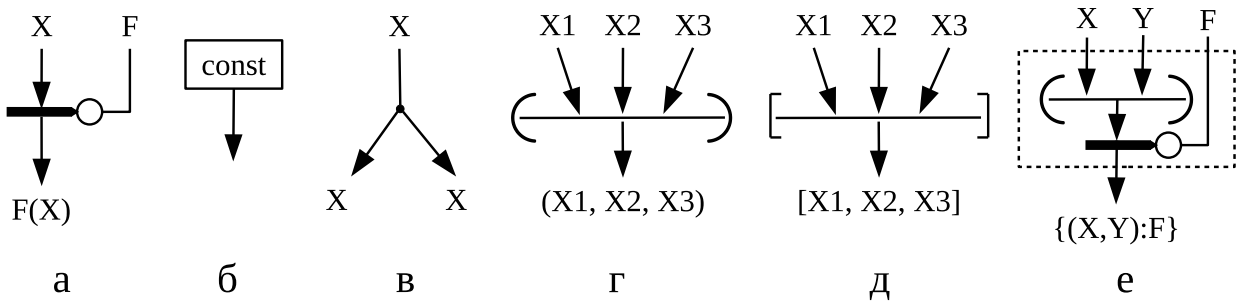


Рисунок 2.1 — Графическое обозначение программо-формирующих операторов; а — оператор интерпретации; б — константный оператор; в — оператор копирования; г — оператор группировки в список данных; д — оператор группировки в параллельный список; е — задержанный список

2.1.2 Этапы выполнения оператора интерпретации

В языке Пифагор имеется только один оператор, применяющий функцию к аргументу — оператор интерпретации [166, 172]. Этот оператор имеет два входа (операнда): один принимает функцию `F` (функциональный вход), другой — аргумент функции `X` (вход для данных). После выполнения, на единственный выход оператора интерпретации передаётся результат применения функции к аргументу `F(X)`. Для достижения более наглядного текстового представления программ оператор интерпретации имеет постфиксную и префиксную формы записи, что позволяет выражение `F(X)` представить двумя способами: «`X:F`» и «`F^X`».

Процесс выполнения оператора интерпретации можно разделить на два этапа [173]:

1. анализ и преобразование операндов;
2. выполнение операции.

Первый этап заключается в преобразовании операндов по правилам эквивалентных преобразований [16, 173]. Правила преобразований приведены в таблице 2.1. В данной таблице пустой элемент обозначается символом «•», для того, чтобы не спутать его с точками в многоточии. Правила преобразований применяются в указанном порядке, и после применения одного из правил следующее ищется с начала списка. Переход ко второму этапу выполнения оператора интерпретации происходит только в том случае, если ни одно из правил нельзя применить. Второй этап — выполнение операции, которое заключается в применении функции к аргументу.

Таблица 2.1 — Анализ и преобразование операндов

1	Эквивалентность элемента и параллельного списка кратности один $[x] \to x$
2	Раскрытие задержанных списков 2.1 $\{X\} : F \to [X] : F$ 2.2 $X : \{F\} \to X : [F]$ 2.3 $\{X\} : \{F\} \to [X] : [F]$
3	Раскрытие параллельных списков 3.1 $[x_1, x_2, \ldots, x_n] : F \to [x_1 : F, x_2 : F, \ldots x_n : F]$ 3.2 $X : [f_1, f_2, \ldots, f_n] \to [X : f_1, X : f_2, \ldots, X : f_n]$ 3.3 $[x_1, x_2, \ldots, x_m] : [f_1, f_2, \ldots, f_n] \to$ $\to [[x_1 : f_1, x_1 : f_2, \ldots, x_1 : f_n], [x_2 : f_1, x_2 : f_2, \ldots, x_2 : f_n], \ldots, [x_m : f_1, x_m : f_2, \ldots, x_m : f_n]]$
4	Преобразования списка данных 4.1 Слияние параллельных списков в списке данных $(x_1, x_2, \ldots, x_{i-1}, [\mathbf{X_i}], x_{i+1}, \ldots, x_n) \to (x_1, x_2, \ldots, x_{i-1}, \mathbf{X_i}, x_{i+1}, \ldots, x_n)$ 4.2 Удаление пустых элементов «•» $(x_1, x_2, \ldots, x_{i-1}, \bullet, x_{i+1}, \ldots, x_n) \to (x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ 4.3 $X : (F) \to (X : [F])$

2.1.3 Семантика типов данных языка Пифагор

Описание типов и семантики встроенных функций языка Пифагор на естественном языке приведено в [16], в работе [173] рассматривается метод формализации семантики встроенных функций языка.

Рассмотрим математическую семантику типов данных языка Пифагор [174]. Математический подход к семантике типов данных — это описание типов на основе теории множеств [136]. Семантика всякого простого (конкретного) типа с точки зрения теории множеств — это множество значений этого типа вместе с совокупностью частичных или тотальных (всюду определённых) операций, в которых эти значения могут быть аргументами или результатами, и совокупность отношений, в которых эти значения могут участвовать. Семантика составного (родового) типа — это (частичная) функция, которая по конкретным типам строит новый тип, то есть по множествам значений с допустимыми операциями и отношениями строит «новое» множество значений со своими операциями и отношениями, в которых могут использоваться эти «новые» значения.

Простые (атомарные) типы языка.

1. Множество целых чисел int, элементами которого являются целые числа из множества $\{x \in \mathbb{Z} \mid \text{MinInt} \leqslant x \leqslant \text{MaxInt}\}$, где MinInt и MaxInt — некоторые заданные константы.

2. Множество действительных чисел float задается диапазоном от минимального отрицательного MinFloat до максимального положительного MaxFloat с указанием точности перечисления PrecFloat. Однако в данной работе точность задания действительных чисел учитываться не будет, поэтому действительные числа рассматриваются как элементы множества $\{x \in \mathbb{R} \mid \text{MinFloat} \leqslant x \leqslant \text{MaxFloat}\}$.

3. Булевское множество bool={true, false}.

4. Конечное множество символов char. Каждый символ — это видимый знак или управляющий символ используемой кодовой таблицы, например ASCII, ограниченный одинарными кавычками.

5. Конечное множество констант ошибок
error={BASEFUNCERROR, BOUNDERROR, INTERROR, REALERROR,
ZERODIVIDE, TYPEERROR}.

6. Множество сигналов signal, содержащее один элемент «пустое значение», обозначаемое точкой «.», signal={.}.

7. Бесконечное множество задержанных списков delaylist — множество констант, каждая из которых соответствует допустимому подграфу программы на языке Пифагор.

Составные типы.

1. Список данных datalist — конечный список, элементы которого могут иметь произвольный тип.

2. Параллельный список parlist отличается от списка данных тем, что для него возможна параллельная обработка всех элементов одной функцией.

Список длины n можно представлять с помощью декартова произведения конечного числа ранее определённых типов языка Пифагор, элементы этого типа — кортежи (обозначаются перечислением элементов в угловых скобках):

$$T_1 \times T_2 \times \cdots \times T_n = \prod_{i=1}^{n} T_i = \{< t_1, t_2, \ldots, t_n > \mid t_i \in T_i,\ i = 1, 2, \ldots, n\},$$

где $n \in \mathbb{N}$ — длина списка, T_i — ранее определённые типы языка Пифагор. Тип списка данных длины n обозначим как datalist$\prod_{i=1}^{n} T_i$, а параллельного списка — parlist$\prod_{i=1}^{n} T_i$. В первом случае элементы ограничиваются круглыми скобками $(t_1, t_2, \ldots, t_n)$, а во втором — квадратными $[t_1, t_2, \ldots, t_n]$.

Функции. Все функции языка Пифагор можно разделить на два подмножества: множество предопределённых (встроенных) функций языка (например, арифметические функции, функции сравнения и др.) и множество функций, порождаемых при программировании (пользовательские функции). В языке функции имеют тип func.

Синтаксически все функции либо не имеют аргумента, либо принимают только один

аргумент. Однако этот аргумент может являться списком данных произвольной длины, что позволяет реализовать любую операцию. Другая особенность функций состоит в том, что они являются полиморфными (ad-hoc-полиморфизм [137]), в том смысле, что они могут принимать аргумент любого типа и возвращать результат любого типа (в зависимости от типа и значения входного аргумента).

Все встроенные функции являются полностью определёнными для любых типов аргументов. Пользовательские функции получаются при «комбинации» встроенных функций. Поэтому в программах не возникают ошибки, связанные с вызовом частично определённых функций со значением аргумента вне области определения.

Для примера рассмотрим семантику функции выбора элемента из списка. Вызов этой функции в синтаксисе языка записывается как `p:n`, где `p` — аргумент (предполагается, что это список), `n` — целое число, направленное на функциональный вход оператора интерпретации (будем ещё называть `n` селектором). Данная запись является некоторым «синтаксическим сахаром» и может быть представлена как `(p,n):select`, где `select` — некоторое обозначение для функции выбора элемента. На естественном языке семантика функции описывается следующим образом [16]:

> Целочисленная константа может интерпретироваться как функция выбора элемента из списка. В качестве аргумента функция принимает список любой размерности, содержащий элементы любого типа. Результат зависит от значения константы. Если она положительна и находится в диапазоне от единицы до величины равной длине списка, то результат — элемент с соответствующим порядковым номером. Если значение константы превышает длину списка, то выдаётся ошибка `BOUNDERROR`. В случае отрицательной константы происходит исключение элемента с соответствующим номером из списка или выдаётся ошибка, если абсолютное значение константы превышает длину списка. Нулевое значение константы возвращает пустое значение «.». Если аргумент не является списком, то возвращается константа ошибки `BASEFUNCERROR`.

Для проведения формальной верификации требуется формальное описание семантики, гарантирующее отсутствие неучтенных вариантов значений входных данных. Выбор того или иного варианта выполнения функции зависит от «свойств» аргумента. Эти «свойства» можно охарактеризовать с помощью значений конечного числа определённых выражений. Последовательно вычисляя значения этих выражений, мы либо сразу получаем вариант выполнения функции, либо приходим к необходимости вычислить значение следующего выражения. Такое представление семантики приведено в работе [173].

Чтобы не вводить новый язык, выражения для аргумента записываются на языке Пифагор, с использованием трех основных функций: определения типа аргумента (`p:type`), получения i-го элемента списка (`p:i`) и определения длины списка (`p:|`). Выражения записываются через эти функции и функции, которые определены раньше рассматриваемой.

В качестве примера в таблице 2.2 приведено описание функции выбора элемента из списка. Считаем, что операции сравнения и проверки на равенство уже были определены в предыдущих правилах.

Замечание. При определении равенства и операций сравнения можно не использовать функцию выбора элементов из списка, если представить аргумент как список из двух элементов `(p1,p2)`, поэтому ситуации циклической зависимости определяемых понятий не возникнет.

В таблице используется обозначение констант из внутреннего представления языка, как пары тип-значение: `<тип, значение>`. Поэтому функция выбора элемента из списка записана как `p:<int,b>`, где `p` — аргумент, `<int,b>` — целочисленная константа со значением `b`. Переменные `p` и `b` участвуют в выражениях и определяют результат выполнения функции. В таблице приводятся все возможные варианты выполнения функции. Выбор того или иного варианта зависит от значений определённых выражений. Во втором столбце таблицы приведены выражения (в первом — их номера), в третьем столбце — их возможные значения, а в четвёртом столбце — результат выполнения функции при указанных значениях выражений. Запись «$\to n$» в четвёртом столбце обозначает, что для определения результата необходимо рассмотреть возможные значения выражения с номером n. Для описания результата используются обычные логические ($\neg, \wedge, \vee$), арифметические операции ($+, -, \times, /$) и знаки сравнения ($=, \neq, >, <, \leqslant, \geqslant$). При необходимости вставляются поясняющие комментарии на естественном языке.

Семантика всех встроенных функций языка Пифагор приведена в приложении Б.

2.2 Спецификация свойств программ на языке Пифагор

Одной из наиболее важных и сложных задач при формальной верификации является описание спецификации программы [3]. Для этого требуется специальный язык спецификации, на котором формулируются требования к программе.

В качестве языка спецификации используем логику высшего порядка, а именно вариант типизированного лямбда-исчисления, называемый исчислением конструкций (система λC) [34, 39]. Этот выбор обусловлен несколькими причинами. Логика должна быть типизированной, потому что, во-первых, функции являются полиморфными, и в спецификации требуется указывать, какой тип аргумента приведёт к какому результату; во-вторых, списки

Таблица 2.2 — Семантика функции выбора элемента из списка (`p:<int,b>`)

№	**Выражение**	**Значения выражения**	**Результат**
1.	`p:type`	`datalist`	$\to 2$
		else	`BASEFUNCERROR`
2.	`(b,0):=`	`true`	. (пустое значение)
		else	$\to 3$
3.	`( (p:|,b):>=, (b,0):>, (p:|,b:-):>=, (b,0):<)`	`(true, true, false, false)` что обозначает: b не превышает длины p и $b > 0$	p_b, где $p = (p_1, \ldots, p_n)$, $n \geqslant b$
	это обозначает следующее: абсолютное значение b не превышает длину списка p	`(false, false, true, true)` что обозначает: $(-b)$ не превышает длины p и $b < 0$	$(p_1, \ldots, p_{d-1}, p_{d+1}, \ldots, p_n)$, где $p = (p_1, \ldots, p_n)$, $d = -b$, $n \geqslant d$
		else	`BOUNDERROR`

могут содержать элементы разного типа, что также требуется описывать с помощью языка спецификации. Порядок логики обуславливают списки языка. В самом общем случае список может иметь неизвестную длину и содержать элементы неизвестного типа. Для того, чтобы описать это в спецификации, требуется использовать кванторы не только для переменных (логика 1-го порядка) и функций (логика 2-го порядка), но и типов (логика высшего порядка). Порядок логики можно понизить, если ограничить списки: либо запретить им иметь произвольную, заранее неизвестную длину, либо заранее неизвестный тип. В самом ограниченном случае можно запретить использовать списки неизвестной длины и неизвестного типа. Такой вариант рассматривается в языке Smile — типизированной версии языка Пифагор [141, 175], в котором вводится несколько различных контейнеров для представления списков совместно с использованием статической типизации и фиксации размерности списковых и контейнерных структур данных. Также введение статической типизации рассматривается в работах [138, 139] при трансляции программ с языка Пифагор на реальные архитектуры.

Язык спецификации вводится как аксиоматическая теория: задаётся язык, аксиомы и правила вывода выражений. Вначале вводится система типов, необходимая для описания типов объектов языка. С помощью правил конструирования описывается синтаксис термов. Далее описывается семантика, правила вывода выражений и теории для разных типов языка. Каждая теория задаётся с помощью сигнатуры и аксиом.

2.2.1 Система типов языка спецификации

Выражениями в языке спецификации являются термы, каждый терм имеет тип. Если M — терм, а A — тип этого терма, то выражение «M типа A» записывается как $M : A$ (это стандартное обозначение λ-исчисления, которое можно понимать как $M \in A$).

Тип тоже является выражением языка и должен иметь тип. Типы типов называются сортами, которые, в свою очередь, тоже термы языка и имеют тип. Таким образом, образуется бесконечная иерархия сортов. В основе иерархии находится сорт Set, далее по иерархии идут типы Type(i), $\forall i \in \mathbb{N}$, при этом выполняются следующие отношения:

$$\mathrm{Set} : \mathrm{Type}(i) \quad (\forall i \in \mathbb{N}),$$

$$\mathrm{Type}(i) : \mathrm{Type}(j) \quad (\text{если } i < j).$$

Согласно этой иерархии, множество термов разделяется на уровни (категории). Каждая категория имеет свои переменные, константы и функции. Тип каждого терма на уровне i — это терм уровня $i + 1$. Кроме того, терм уровня i имеет множество типов уровня j, для $j > i$. Множество всех сортов $\mathcal{S} = \{\mathrm{Set}, \mathrm{Type}(i) \mid i \in \mathbb{N}\}$. Далее для Type(1) используется обозначение Type.

Приведём несколько примеров.

Уровень 0. Исходные величины и функции. Примерами констант данной категории являются целые и действительные числа, например, «$1 : \mathbb{N}$» и «$2.5 : \mathbb{R}$». Функциями являются операции сложения «$+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$», унарный минус «$- : \mathbb{Z} \to \mathbb{Z}$». Также сюда можно отнести функцию суммирования $\sum_{i=0}^{n} f(i)$, принимающую целое число и функцию : «$\sum : \mathbb{N} \to (\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$».

Уровень 1. Типы данных и функций. Простые типы являются константами. Например, множество натуральных чисел «$\mathbb{N} : \mathrm{Set}$», множество целых чисел «$\mathbb{Z} : \mathrm{Set}$», множество целочисленных функций от двух переменных «$\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} : \mathrm{Set}$». Составные типы (операторы типа) являются функциями, например, декартово произведение двух типов «$\times : \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set}$», список элементов заданного типа «$\mathrm{list} : \mathrm{Set} \to \mathrm{Set}$».

Уровень 2. Сорт Set. Примеры констант: «$\mathrm{Set} : \mathrm{Type}$», «$\mathrm{Set} \to \mathrm{Set} : \mathrm{Type}$».

2.2.2 Синтаксис языка спецификации

Множество *псевдотермов* $\mathfrak{I}$ системы λC имеет следующий абстрактный синтаксис:

$$\mathfrak{I} = V \mid C \mid \mathfrak{I}\,\mathfrak{I} \mid \lambda V : \mathfrak{I}.\,\mathfrak{I} \mid \Pi\, V : \mathfrak{I}.\,\mathfrak{I}$$

где V — бесконечное множество переменных (обозначаются буквами x, y, и т.д.);

C — бесконечное множество констант (обозначаются буквами c, d и т.д.), среди которых выделены константы, называемые сортами: Set, Type(i), $i \in \mathbb{N}$;

$\mathfrak{I}\,\mathfrak{I}$ — применение (application) первого терма ко второму;

$\lambda V : \mathfrak{I}.\, \mathfrak{I}$ — λ-абстракция (abstraction), терм $\lambda x : T.\, u$ является функцией, отображающей элементы множества T в выражение u.

$\Pi\, V : \mathfrak{I}.\, \mathfrak{I}$ — зависимое произведение типов (dependent product), если в терме $\Pi\, x : T.\, U$ переменная x не присутствует в U (U не зависит от x), то независимое произведение записывается как $T \to U$.

Пусть M, N — псевдотермы, тогда «$M \equiv N$» означат синтаксическую эквивалентность термов (обе буквы обозначают один и тот же терм). Если псевдотерм M содержит свободную переменную x, то это выражается с помощью записи $M[x]$ ($M \equiv M[x]$).

Для псевдотермов определена операция подстановки «:=». Выражение $M[x := N]$ означает, что все свободные вхождения переменной x в псевдотерме M заменяются на N.

Считаем, что в терме все связанные переменные отличаются от свободных, это достигается переименованием связанных переменных (α-конверсия), например $\lambda x.x$ становится $\lambda y.y$. Таким образом, псевдотермы, различающиеся только связанными переменными, также эквиваленты (α-эквивалентны): $\lambda x.x \equiv \lambda y.y$. Поэтому каждой подстановке предшествует переименование связанных переменных так, чтобы их имена отличались от имён свободных переменных «подставляемого» псевдотерма.

Функцию от нескольких аргументов представляем с помощью вложенных лямбда-функций и последовательного применения к ним нескольких аргументов. Например, для функции от двух аргументов $f(x, y)$ определим $F \equiv \lambda x.\, \lambda y.\, f(x, y)$, тогда $(F\, x)\, y \equiv f(x, y)$.

Используя левую ассоциативность для применения функции к аргументу введём следующее обозначение:

$$(\ldots((F\, M_1)\, M_2) \ldots\, M_n) \equiv F\, M_1\, \ldots\, M_n \equiv F(M_1, \ldots, M_n).$$

Аналогично λ-абстракция (и зависимое произведение типов) имеет правую ассоциативность, поэтому:

$$\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.\, f(x_1, \ldots, x_n))\ldots)) \equiv \lambda x_1\, \ldots\, x_n.\, f(x_1, \ldots, x_n);$$

$$\lambda x_1 : A\, \ldots\, x_n : A\,.f \equiv \lambda x_1\, \ldots\, x_n : A\,.f \equiv \lambda(x_1, \ldots, x_n : A)\,.f.$$

Для псевдотермов $\mathfrak{I}$ определены понятия β-конверсии с помощью следующего правила:

$$(\lambda x : A.\, B)\, C =_\beta B[x := C].$$

Также для типов определено отношение «являться подтипом» $\leq$, учитывающие иерархию типов (см. раздел 2.2.1):

1. если $T =_\beta T'$, то $T \leq T'$;

2. $\forall T : \mathrm{Set}.\, T \leq \mathrm{Set}$;

3. $\mathrm{Type}(i) \leq \mathrm{Type}(j),\ \forall i \leqslant j,\ i, j \in \mathbb{N}$.

Термы имеют вид: $M : A$, где $M, A \in \mathfrak{I}$.

Контекст (context) — конечная, линейно упорядоченная последовательность термов вида $x : A$, где x — переменная, а $A \in \mathfrak{I}$, при этом все переменные попарно различные. Для обозначения контекста используются буквы Γ, Δ и т.д., «<>» обозначает пустой контекст. Добавление переменной y в контекст обозначают: $\Gamma, y : B =< x_1 : A_1, \ldots, x_n : A_n, y : B >$, где $\Gamma =< x_1 : A_1, \ldots, x_n : A_n >$. Терм-в-контексте — это контекст Γ вместе с термом $t \equiv t_1 : t_2$, удовлетворяющий следующим условиям: Γ содержит все свободные переменные из t_2 (тип для t_1) и все свободные переменные переменные из t_1, в контексте отсутствуют связанные переменные t. Контекст может содержать переменные, отсутствующие в t. Из этих условий следует, что переменная в t не может быть одновременно свободной и связанной. Для любого терма всегда найдётся α-эквивалентный терм, удовлетворяющей этому условию, полученный посредством переименования необходимых связанных переменных.

Тип терма зависит от контекста, поэтому правила присваивания типа аксиоматизируют понятие выводимости терма из контекста: $\Gamma \vdash A : B$, где A и B — псевдотермы, а Γ — контекст. Пусть $s, s_1, s_2 \in \mathcal{S}$, A, B, C, F, a, b — произвольные псевдотермы, x, y — произвольные переменные, тогда понятие выводимости $\vdash$ определяется следующими аксиомами и правилами вывода [34, 39]:

$$<> \vdash \mathrm{Set} : \mathrm{Type}\,; \qquad \text{(axiom-set)}$$

$$<> \vdash \mathrm{Type}(i) : \mathrm{Type}(i+1)\,; \qquad \text{(axiom-type)}$$

$$\frac{\Gamma \vdash A : s}{\Gamma,\ x : A \vdash x : A},\ x \notin \Gamma\,; \qquad \text{(start)}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma,\ x : C \vdash A : B},\ x \notin \Gamma\,; \qquad \text{(weakening)}$$

$$\frac{\Gamma \vdash F : (\Pi\, x : A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B[x := a]}\,; \qquad \text{(application)}$$

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi\, x : A.\, B) : s}{\Gamma \vdash (\lambda x : A.\, b) : (\Pi\, x : A.\, B)}\,; \qquad \text{(abstraction)}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B \leq B'}{\Gamma \vdash A : B'}\,; \qquad \text{(convertion-type)}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\Pi\, x : A.\, B) : s_2}\,. \qquad \text{(product)}$$

Для того, чтобы ввести новую константу c типа A, добавляют правило вывода:

$$\frac{\Gamma \vdash A : s}{\Gamma \vdash c : A}, \qquad \text{(constant)}$$

При этом имена всех констант должны быть различны, нельзя дважды определить одну константу с разными типами.

2.2.3 Семантика языка спецификации

Рассмотренный язык позволяет описывать объекты и их тип, однако не позволяет формулировать свойства объектов. Свойства описываются с помощью логических формул. Для их представления необходимо ввести атомарный тип bool, обозначающий множество из двух элементов {true, false}. Далее логические формулы отождествляются с термами типа bool. Необходимость расширения языка новыми константами формализуется с помощью понятия стандартной сигнатуры.

Сигнатура Σ — множество выделенных констант (константы, которым даны имена). Семантика термов определяется моделью. *Модель* — это функция, которая сопоставляет каждому типу некоторое непустое множество (тогда переменные могут принимать значения любого элемента из множества, соответствующего их типу), а каждой константе сопоставляет элемент множества, соответствующий её типу.

Введём стандартную сигнатуру (подобную [125]). Сигнатура Σ *стандартная*, если она содержит атомарный тип $\mathrm{bool} : \mathrm{Set}$, тип $\mathrm{ind} : \mathrm{Set}$ и три примитивные константы:

$\Rightarrow \ : \mathrm{bool} \to \mathrm{bool} \to \mathrm{bool}$,

$= \ : (\Pi\, A : \mathrm{Type}.\ A \to A \to \mathrm{bool})$,

$\varepsilon : (\Pi\, A : \mathrm{Type}.\ (A \to \mathrm{bool}) \to A)$.

В данном случае используется теоретико-множественная интерпретация, в которой типы (термы, имеющие тип Set) обозначают множества, а термы с этим типом соответствуют элементам множества. В стандартной модели M сигнатуры Σ предполагаемая интерпретация констант такая: bool — множество булевских констант, ind — бесконечное множество элементов (individuals).

Константа $\Rightarrow$ обозначает импликацию:

$$(b \Rightarrow b') = \begin{cases} \text{false}, & \text{если } b = \text{true и } b' = \text{false}\ ; \\ \text{true}, & \text{иначе.} \end{cases}$$

Константа $=$ означает проверку равенства на множестве, обозначенном A. Если X — множество, x, $x' \in X$, то

$$(x = x') = \begin{cases} \text{true}, & \text{если } x = x'; \\ \text{false}, & \text{иначе.} \end{cases}$$

Константа ε — функция выбора на множестве, обозначенном A. Если X — множество, $f \in (X \to \text{bool})$, то

$$\varepsilon(f) = \begin{cases} \text{ch}(f^{-1}\{\text{true}\}), & \text{если } f^{-1}\{\text{true}\} \neq \varnothing; \\ \text{ch}(X), & \text{иначе;} \end{cases}$$

где $\text{ch} : \Pi\, X : U.\, X$ — функция выбора элемента из множества (получает непустое множество из вселенной (universe) U в качестве аргумента), $f^{-1}\{\text{true}\} = \{x \in X \mid f(x) = \text{true}\}$. Получаем, что f задаёт подмножество множества X, и, если оно не пустое, то ε возвращает его элемент, иначе возвращается произвольный элемент X.

Используем следующие обозначения. Если $t_1 : \text{bool}$ и $t_2 : \text{bool}$, то $(\Rightarrow t_1\ t_2) \equiv (t_1 \Rightarrow t_2)$. Если A — тип, t_1 и t_2 — элементы этого типа, тогда $(=\ A\ t_1\ t_2) \equiv (=\ t_1\ t_2) \equiv (t_1 = t_2)$. В последних двух случаях аргумент A становится неявным (определяется по типам элементов). Если A — тип, $t : \text{bool}$, то $(\varepsilon\ A\ (\lambda x : A.\, t)) \equiv (\varepsilon\ x : A.\, t)$.

2.2.4 Теории языка спецификации

Для того, чтобы построить аксиоматическую теорию, в которой возможно формулировать и доказывать утверждения о свойствах объектов, необходимо расширить имеющуюся систему логическими аксиомами и правилами вывода. Также необходима возможность проводить доказательства при наличии некоторых предпосылок (предположений, assumptions) — утверждений, которые считаются истинными (но необязательно являются тождественно истинными в модели). Следствия предпосылок будут истинны в том случае, если удастся доказать истинность их предпосылок.

Пусть Δ — контекст, $\Gamma = \{t_1, t_2, \ldots t_n\}$ — множество формул в контексте Δ (термы типа bool), называемых предпосылками, тогда $\Delta.\Gamma \vdash t$ обозначает (логическую) выводимость формулы t в контексте Δ из формул $t_1, t_2, \ldots t_n$ с помощью аксиом и правил вывода.

Используем следующие сокращения записи: в случае, если контекст может быть произвольным, то он не указывается: $\Delta.\Gamma \vdash t \equiv \Gamma \vdash t$; если контекст важен, а предпосылки отсутствуют, то $\Delta.\Gamma \vdash t \equiv \Delta.\ \vdash t$.

Если t — произвольная формула, t_1, t_2 — произвольные термы с подходящими типами, то схемы дополнительных (логических) правил вывода будут следующими:

$$\frac{}{t \vdash t}\ ; \qquad \text{(assume)}$$

$$\frac{}{\vdash t = t}\ ; \qquad \text{(reflexivity)}$$

$$\frac{}{\vdash (\lambda x.\, t_1)\, t_2 = t_1[x := t_2]}\ ; \qquad \text{(conversion)}$$

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\, t_1) = (\lambda x.\, t_2)}\,, \qquad \text{(abstraction-term)}$$

где переменная x не встречается в свободном виде в Γ;

$$\frac{\Gamma \vdash t_2}{\Gamma \setminus \{t_1\} \vdash t_1 \Rightarrow t_2}\,; \qquad \text{(discharge-assumption)}$$

$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}\,, \qquad \text{(modus ponens)}$$

где символ $\setminus$ обозначает разность множеств, а $\cup$ — объединение множеств.

Терм $M : A$ может иметь два вида переменных: обычные переменные (переменные из M) и переменные типа (переменные из A), поэтому следует рассмотреть два вида подстановок: подстановка переменных терма и подстановка переменных типа.

$$\frac{\Gamma_1 \vdash t_1 = t_1' \quad \cdots \quad \Gamma_n \vdash t_n = t_n' \quad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1', \ldots, t_n']}\,, \qquad \text{(substitution)}$$

где $t[t_1, \ldots, t_n]$ обозначает терм t со свободным вхождением подтермов $t_1, \ldots, t_n$, $t[t_1', \ldots, t_n']$ — результат замены каждого вхождения t_i на t_i' для $1 \leqslant i \leqslant n$;

$$\frac{\Gamma \vdash t}{\Gamma \vdash t[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]}\,, \qquad \text{(instantiation-type)}$$

при этом ни одна из переменных типа $\alpha_1, \ldots, \alpha_n$ не должна присутствовать в Γ, а также различные переменные в t не должны отождествляться в результате подстановки.

Остаётся дополнить систему необходимыми константами типов языка Пифагор, элементами этих типов (задать сигнатуру) и аксиомами, характеризующими свойства этих констант.

Теория стандартной модели (тип bool и ind).

Кроме констант bool, ind, $\Rightarrow$, $=$ и ε, описанных выше, теория содержит следующие константы: true, false, $\forall$, $\exists$, $\exists!$, $\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, OneOne, Onto. Семантика данных констант стандартная: true и false — элементы булевского множества; $\forall$, $\exists$, $\exists!$ — кванторы всеобщности, существования и существования и единственности; $\wedge$, $\vee$, $\Leftrightarrow$ $\neg$ — операции конъюнкции, дизъюнкции, эквиваленции и отрицания; OneOne и Onto описывают свойства функции быть инъективной и сюръективной соответственно. Подробное описание и аксиоматическое определение данных констант приведено в приложении В [125].

Для операций $\wedge$, $\vee$ и $\Leftrightarrow$ далее используем инфиксную форму.

Теория натуральных чисел (с нулём).

Несмотря на то, что в Пифагоре отсутствует тип натуральных чисел, он вводится в языке спецификации и далее используется для описания целых чисел. Данная теория является расширением теории стандартной модели. Её сигнатура $\Sigma_{\mathbb{N}}$ включает следующие константы: $\mathbb{N}$, 0, SUC, 1, 2, 3, и т.д., $+$, $\cdot$, где SUC — функция следования, 1, 2, 3, . . . — константы,

обозначающие числа, а $+$, $\cdot$ — символы операций. Подробное описание и аксиоматическое определение констант приведено в приложении В [146, 136, 145, 125].

Далее в выражениях символы бинарных операций и отношений будут записываться в инфиксной форме.

Теория целых чисел.

Аксиоматическая система для целых чисел строится на основе аксиоматики натуральных чисел. Сигнатура $\Sigma_{\mathbb{Z}}$ включает следующие константы: $\mathbb{Z}$, $0, 1$, $-$ (унарный минус), $+$, $\cdot$, $-$, $\mod$, $/$, $<$, $>$, $\leqslant$, $\geqslant$. Подробное описание и аксиоматическое определение данных констант приведено в приложении В [136, 142, 144].

Теория действительных чисел.

Теория действительных чисел является расширением теории натуральных чисел. Сигнатура $\Sigma_{\mathbb{R}}$ содержит следующие константы: $\mathbb{R}$, $0, 1$, $+$, $\cdot$, $-$, $/$, $-$ (унарный минус), $<$, $>$, $\leqslant$, $\geqslant$, NR, ZR, RZ. Функции NR, ZR, RZ осуществляют преобразование типов из $\mathbb{N}$ в $\mathbb{R}$, из $\mathbb{Z}$ в $\mathbb{R}$ и из $\mathbb{R}$ в $\mathbb{Z}$ соответственно.

Замечание. Далее считаем, что преобразование типов происходит по умолчанию и используем упрощенную записью формул, когда при совершении операций над натуральными или целыми и действительными числами опускаются функции преобразований, если их применение понятно из контекста. Например,

$$\forall(x:\mathbb{N})(r:\mathbb{R}).\,\mathrm{NR}(x)\cdot r \equiv \forall(x:\mathbb{N})(r:\mathbb{R}).\,x\cdot r.$$

Подробное описание и аксиоматическое определение констант теории действительных чисел приведено в приложении В [142, 143, 125].

Множество символов.

Тип символов обозначается константой типа $\mathrm{char}:\mathrm{Set}$. Элементами типа являются символы одной из кодовых таблиц, например ASCII: ‘A’, ‘B’, ‘C’, . . . , ‘0’, ‘1’, ‘2’, . . . , ‘\n’, ‘\\’ и т.д. Также имеется две функции, которые по кодовой таблице преобразуют символ в целое число и наоборот: $\mathrm{CharInt}:\mathrm{char}\to\mathbb{N}$, $\mathrm{IntChar}:\mathbb{N}\to\mathrm{char}$.

Множество констант ошибок.

Тип обозначается константой $\mathrm{error}:\mathrm{Set}$, элементы — имена ошибок в языке Пифагор.

Множество имён функций языка Пифагор.

Тип обозначается константой $\mathrm{func}:\mathrm{Set}$, элементы — константы, обозначающие имена встроенных и пользовательских функций в языке Пифагор. Подмножество, состоящее из имён встроенных функций имеет вид:

$\mathrm{spec}=\{\mathrm{type}_p,\ |_p,\ +_p,\ -_p,\ *_p,\ /_p,\ \%_p,\ =_p,\ !=_p,\ <_p,\ >_p,\ <=_p,\ >=_p,\ ?_p,\ \#_p,\ \mathrm{datalist}_p,\ \mathrm{parlist}_p,$

$\ldots_p,\ \mathrm{dup}_p,\ \mathrm{int}_p,\ \mathrm{float}_p,\ \mathrm{char}_p,\ \mathrm{bool}_p,\ \mathrm{signal}_p\}.$

Индекс «p» введён для того, чтобы не путать константы из множества spec и идентификаторы функций языка спецификации.

Множество сигналов.

Тип $\mathrm{signal} : \mathrm{Set}$ содержит один элемент «$\bullet$».

Множество задержанных списков.

Обозначается $\mathrm{delaylist} : \mathrm{Set}$. Элементами типа являются строки особого вида. Строка является списком символов $\mathrm{list\,char}$ (тип list описывается ниже). Каждая строка — это код, соответствующий некоторому допустимому подграфу программы на языке Пифагор.

Теория списков.

Списки являются составными (полиморфными) типами [41, 125]. Они представляют собой последовательность элементов некоторого типа. Тип элементов списка является параметром. Сигнатура Σ_L содержит следующие символы: list, nil, cons, head, tail, length, elem. Подробное описание и аксиоматическое определение констант теории списков приведено в приложении В [125, 147, 148].

Далее аргумент A зависимого произведения типов у nil, cons и функций списка будет неявным, то есть не записывается, а определяется по типу списка. Также введём следующее обозначение для списков:

$$\mathrm{cons}(a_1, \mathrm{cons}(a_2, \dots \mathrm{cons}(a_n, \mathrm{nil}) \dots) \equiv (a_1, a_2, \dots, a_n),$$

$$\mathrm{elem}(n, l) \equiv l[n].$$

В языке Пифагор списки могут содержат элементы разных типов, поэтому дополним теорию списков «гетерогенными» списками.

Тип list позволяет формировать списки типов $\mathrm{list\,Set}$, например, $(\mathbb{N}, \mathbb{Z}, \mathbb{N}{\to}\mathbb{R})$. Введём новый тип datalist — элементами которого являются списки с элементами разного типа. Сигнатура «гетерогенных» списков следующая:

$\mathrm{datalist} : (\mathrm{list\,Set}){\to}\mathrm{Set}$,

$\mathrm{hnil} : \mathrm{datalist\,nil}$,

$\mathrm{hcons} : \Pi(A : \mathrm{Set})(L : \mathrm{list\,Set}).\, A{\to}\mathrm{datalist}\, L{\to}\mathrm{datalist}\,(\mathrm{cons}\, A\, L)$.

$\mathrm{hhead} : \Pi(A : \mathrm{Set})(L : \mathrm{list\,Set}).\, \mathrm{datalist}\,(\mathrm{cons}\, A\, L){\to}A$,

$\mathrm{htail} : \Pi(A : \mathrm{Set})(L : \mathrm{list\,Set}).\, \mathrm{datalist}\,(\mathrm{cons}\, A\, L){\to}\mathrm{datalist}\, L$,

$\mathrm{hlength} : \Pi\, L : \mathrm{list\,Set}.\, (\mathrm{datalist}\, L){\to}\mathbb{N}$,

$\mathrm{helem} : \Pi(A : \mathrm{Set})(L : \mathrm{list\,Set}).\, \mathbb{N}{\to}\mathrm{datalist}\, L{\to}A$.

Введём следующее обозначение для списков типа $\mathrm{datalist}(A_1, A_2, A_3, \dots A_n)$:

$$\mathrm{hcons}(A_1, (A_2, A_3, \dots, A_n), a_1, \mathrm{hcons}(A_2, (A_3, \dots, A_n), a_2, \dots \mathrm{hcons}(A_n, \mathrm{nil}, a_n, \mathrm{hnil})) \dots) \equiv$$
$$\equiv (a_1 : A_1, a_2 : A_2, \dots, a_n : A_n) \equiv (a_1, a_2, \dots, a_n),$$

последнее обозначение используется в случае, если типы элементов понятны из контекста.

Аксиомы типа datalist.

1. Аксиомы конструирования списка datalist (аналогичны соответствующим аксиомам для list).

2. Аксиомы, определяющие функцию, возвращающую первый элемент списка hhead, хвост списка htail, длину hlength (аналогичны соответствующим аксиомам для list); аксиома извлечения i-го элемента списка helem:

$$\vdash \forall(A, B : \mathrm{Set})(L : \mathrm{list\,Set})(a : A)(b : B)(l : \mathrm{datalist}\, L).$$
$$(\mathrm{helem}(A, \mathrm{cons}(A, L), 1, \mathrm{hcons}(a, l)) = a) \wedge (\forall n : \mathbb{N}.\, (n > 1) \Rightarrow$$
$$\Rightarrow (\mathrm{helem}(A, \mathrm{cons}(A, \mathrm{cons}(B, L)), \mathrm{SUC}\, n, \mathrm{hcons}(a, \mathrm{hcons}(b, l))) =$$
$$= \mathrm{helem}(B, \mathrm{cons}(B, L), n, \mathrm{hcons}(b, l)))),$$

где в функции hcons два первых аргумента неявные.

3. Аксиомы преобразования списка:

$\vdash \mathrm{hnil} = (\bullet : \mathrm{signal})$,

$\vdash \forall(L : \mathrm{list\,Set})(l : \mathrm{datalist}\, L).\, \mathrm{hcons}(\bullet, l) = l.$

Аналогично введём тип parlist с соответствующими функциями phnil, phcons, и др. Аксиомы parlist совпадают с аксиомами datalist, за исключением последней группы, так как для параллельных списков аксиомы эквивалентных преобразований свои и строятся на основе таблицы 2.1.

Введём следующее обозначение для списков типа $\mathrm{parlist}(A_1, A_2, A_3, \ldots A_n)$:

$$\mathrm{phcons}(A_1, (A_2, A_3, \ldots, A_n), a_1, \mathrm{phcons}(A_2, (A_3, \ldots, A_n), a_2, \ldots \mathrm{phcons}(A_n, \mathrm{nil}, a_n, \mathrm{phnil}))\ldots) \equiv$$
$$\equiv [a_1 : A_1, a_2 : A_2, \ldots, a_n : A_n].$$

Подмножества и принадлежность ко множеству.

Зададим функцию, определяющую принадлежность элемента ко множеству:

$\in : \Pi(A : \mathrm{Set})(B : \mathrm{Set}).B \to \mathrm{bool}.$

Аксиома, характеризующая эту функцию:

$\vdash \forall(A : \mathrm{Set})(B : \mathrm{Set})(x : B).\ \in (A, B, x) = (A = B).$

Пусть аргумент B определяется по умолчанию, тогда используем следующие обозначение:

$$\in (A, B, x) \equiv x \in A.$$

Подмножество некоторого множества можно охарактеризовать с помощью предиката, принимающего истинное значение, если элемент принадлежит подмножеству. Например, A — тип, тогда его подтип будет иметь вид $\{x : A \mid P(x)\}$, где $P : A \to \mathrm{bool}$ — предикат. Будем

использовать следующее обозначение:

$$a \in A \wedge P(a) \equiv a \in \{x : A \mid P(x)\}.$$

Зададим функцию, определяющую множество, которому принадлежит элемент:

$\mathrm{type} : \Pi(A : \mathrm{Set}).A \to \mathrm{Set}$.

Аксиома, характеризующая эту функцию:

$\vdash \forall(A : \mathrm{Set})(x : A).\, \mathrm{type}(A, x) = A$,

далее аргумент A определяется по умолчанию.

Дополнительные обозначения.

Для сокращения записи введём несколько обозначений.

$$(a_1 \text{ op } b) \wedge (a_2 \text{ op } b) \wedge \ldots, (a_n \text{ op } b) \equiv (a_1, a_2, \ldots, a_n \text{ op } b),$$

где op — некоторая бинарная функция. Например, данное обозначение используется в следующих формулах: $(a, b > 0)$, $(z, y \in int)$ и др.

$$(a_1 \text{ op } a_2) \wedge (a_2 \text{ op } a_3) \wedge \cdots \wedge (a_{n-1} \text{ op } a_n) \equiv (a_1 \text{ op } a_2 \text{ op } \ldots \text{ op } a_n),$$

где оператор $\text{op} \in \{<, >, \leqslant, \geqslant, =\}$ или другой транзитивный оператор.

$$(a \in A_1) \vee (a \in A_2) \vee \cdots \vee (a \in A_n) \equiv a \in (A_1 \mid A_2 \mid \cdots \mid A_n),$$

$$x \in \mathbb{Z} \wedge (\mathrm{MinInt} \leqslant x \leqslant \mathrm{MaxInt}) \equiv a \in \mathrm{int},$$

$$x \in \mathbb{R} \wedge (\mathrm{MinFloat} \leqslant x \leqslant \mathrm{MaxFloat}) \equiv a \in \mathrm{float},$$

$$\neg(a \in A) \equiv a \notin A.$$

Также используем сокращения для связанных переменных:

$$\forall x : \mathbb{Z}.\, (\mathrm{MinInt} \leqslant x \leqslant \mathrm{MaxInt}) \Rightarrow g(x) \equiv \forall x \in \mathrm{int}.\, g(x).$$

Непротиворечивость и надёжность системы. Рассмотренная система является надёжной для теоретико-множественной семантики [125]. Каждый тип строится на основе уже существующего типа. Например, натуральные числа определяются как счётное подмножество типа ind. Целые числа можно представить как пары двух натуральных чисел вида $(n, 0)$ или $(0, n)$, где первая пара соответствует положительному целому, а вторая — отрицательному. Действительные числа строятся на основе рациональных, а те, в свою очередь, на основе целых. Списки можно представить как пару, первый элемент которой есть функция, а второй — целое число, равное размеру списка. Каждая аксиома истинна в модели, и все правила вывода надёжны. То есть, если гипотезы правила вывода истинны в (стандартной)

модели, то заключение этого правила тоже истинно в модели. Значит, утверждение истинно в модели, если у него есть формальное доказательство из аксиом по правилам вывода.

Рассмотренная теория является непротиворечивой, так как она имеет стандартную модель [125]. Логические противоречия не удовлетворяют ни одной модели, это и гарантирует непротиворечивость теории, имеющей модель. Надёжность гарантирует выводимость только истинных утверждений, поэтому логическое противоречие не может быть формально выведено из аксиом.

Однако, теория не является полной (по теореме Гёделя о неполноте), так как в ней определены основные арифметические понятия: натуральные числа, сложение и умножение.

2.2.5 Совместимость языка спецификации с системой HOL

Язык спецификации ФПП программ разрабатывался с учётом логики системы интерактивного поиска доказательства HOL. Поэтому большинство теорий, в частности, теория стандартной модели и теории чисел, полностью совместимы с системой HOL. В приложении Г приведена таблица соответствия типов языка спецификации ФПП программ и встроенных типов логики HOL.

Часть типов, специфичных для языка Пифагор, отсутствуют в логике HOL, однако теории этих типов простые и могут быть доопределены в системе HOL. Например, множество сигналов и констант ошибок, которые являются простыми типами с конечным числом элементов. Другая часть специфичных для языка Пифагор типов — списки. Несмотря на то, что строки в языке Пифагор являются списком данных, для их описания в логике HOL можно использовать тип string. Тогда задержанные списки представляются как «обертка» над строкой string. Остаётся определить списки данных datalist и параллельные списки parlist.

Рассмотрим список данных (для параллельных списков всё аналогично). В общем случае, список данных в языке Пифагор может:

1. иметь произвольную длину,

2. содержать элементы различных типов,

и в этом случае он не аксиоматизируется встроенными теориями системы HOL (и другими подобными системами интерактивного доказательства). Однако если случаи 1 и 2 не имеют место одновременно, то при переводе условий корректности в систему HOL тип datalist представляется одним из встроенных типов HOL.

Если длина списка — константа, то он может быть представлен типом prod, который описывает декартово произведение двух типов, элементами типа являются упорядоченные пары значений.

Если список имеет элементы одинакового типа (тип может быть заранее неизвестен), то он может быть представлен типом list.

Для того, чтобы была возможна такая замена, достаточно, чтобы одно из указанных ограничений следовало из спецификации к программе. Если программа всё же предполагает использование списка datalist в общем виде, то предусловие в такой спецификации не будет слабейшим.

В приложении Г описана теория, дополняющая логику HOL недостающими типами языка Пифагор и функциями. Также в приложении приведено несколько примеров условий корректности ФПП программы, записанных на языке логики HOL.

2.3 Аксиоматическая теория языка Пифагор

Для выполнения формальной верификации на основе аксиоматического подхода строится аксиоматическая теория для языка ФПП программирования Пифагор, в рамках которой и будут проводиться формальные доказательства корректности программ [176, 177, 178]. Аксиоматическая теория строится по аналогии с теорией для императивных языков программирования [1]: задаётся язык теории, аксиомы и правила вывода.

2.3.1 Язык аксиоматической теории

Основными объектами аксиоматической теории для языка программирования Пифагор являются тройки Хоара. В традиционном виде тройка Хоара записывается в виде

$$\{\varphi\}\ \texttt{Prog}\ \{\psi\},$$

где `Prog` — программа на языке Пифагор, а φ и ψ — предусловие и постусловие на языке спецификации. Для того, чтобы фигурные скобки тройки не совпадали с фигурными скобками языка программирования или языка логической спецификации, для тройки Хоара вводится следующее обозначение:

$$\boxed{\varphi(x)}\ \texttt{Prog(x)}\rightarrow r\ \boxed{\psi(r)}\ ,$$

где `Prog(x)` — код функции с входным аргументом x, r — результат вычислений функции, $\varphi(x)$ — предусловие для `Prog`, зависящее от аргумента x, $\psi(r)$ — постусловие для `Prog`, зависящее от результата выполнения функции (и от входного аргумента, так как от него зависит результат).

Например, рассмотрим следующую функцию на языке Пифагор:

```
Fun << funcdef arg {
   arg:F >>  return
}
```

Пусть P и Q — предусловие и постусловие для этой функции. Тогда тройка Хоара будет иметь вид:

$$\boxed{P(arg)}\ \texttt{arg:F}\rightarrow r\ \boxed{Q(r)}\ .$$

В связи с тем, что программу на языке Пифагор удобно отображать в виде информационного графа, возможна непосредственная привязка предусловия и постусловия к дугам этого графа. Пример подобной привязки представлен на рисунке 2.2. Для операторов используются условные обозначения, введённые на рисунке 2.1. Входной аргумент и возвращаемое значение обозначаются пятиугольником с записанным в нём идентификатором.

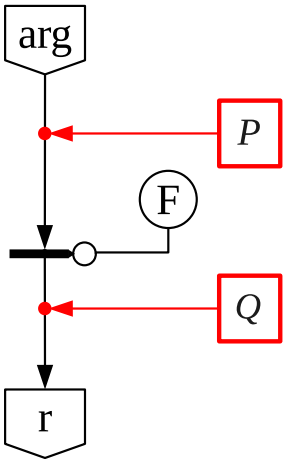


Рисунок 2.2 — Тройка Хоара для функции `Fun` в виде информационного графа

Язык Пифагор имеет неявную типизацию, а формулы на языке спецификации — явную, поэтому для правильной записи предусловий и постусловий необходимо указать «ожидаемый» тип аргумента и результата. Кроме того, в языке Пифагор передача нескольких аргументов в функцию осуществляется с помощью списка данных, который обозначается идентификатором, не отражающим его внутреннюю структуру. Для устранения несоответствий между языками, формула предусловия (постусловия) обязательно должна включать утверждение о типе аргумента (результата) или описание его структуры с указанием типов элементов списка данных datalist. На основе этого утверждения однозначно должен восстанавливаться контекст формулы, при этом для предусловия и постусловия контекст один и тот же. Различные варианты указания типов и структуры аргумента (результата) на языке спецификации и восстанавливаемый по ним контекст приведены в таблице 2.3. Также для удобства в каждом случае вводится сокращённая форма записи.

Если аргумент (результат) имеет вид списка данных, то этот список обязательно конечен и описывается в общем виде или с помощью перечисления всех элементов. При этом допускается использовать вложенные списки, например: $x = ((a_1 : A_1, \ldots, a_m : A_m), t_2 : T_2, \ldots, t_n : T_n)$.

Таблица 2.3 — Способы описания типа и структуры аргумента (результата) в формулах на языке спецификации

Сокращённая форма	**Формула на языке спецификации**	**Контекст**
$x \in T$	$x \in T$	$x : T$
$x \notin T$	$(x \in T_1) \wedge (T \neq T_1)$	$T : \mathrm{Set}, T_1 : \mathrm{Set}, x : T_1$
$x \in (T_1 \mid T_2 \mid \cdots \mid T_n)$	$x \in T \wedge ((T = T_1) \vee \cdots \vee (T = T_n))$	$T_1 : \mathrm{Set}, \ldots, T_n : \mathrm{Set}, T : \mathrm{Set}, x : T$
$x \notin (T_1 \mid T_2 \mid \cdots \mid T_n)$	$x \in T \wedge ((T \neq T_1) \wedge \cdots \wedge (T \neq T_n))$	$T_1 : \mathrm{Set}, \ldots, T_n : \mathrm{Set}, T : \mathrm{Set}, x : T$
$x = (t_1 : T_1, \ldots, t_n : T_n)$	$x \in \mathrm{datalist}(T_1, \ldots, T_n) \wedge$ $x = (t_1 : T_1, \ldots, t_n : T_n)$	$T_1 : \mathrm{Set}, \ldots, T_n : \mathrm{Set}, t_1 : T_1, \ldots, t_n : T_n,$ $x : \mathrm{datalist}(T_1, \ldots, T_n)$
$x \neq (t_1 : T_1, \ldots, t_n : T_n)$	$x \in T \wedge (T \neq \mathrm{datalist}(T_1, \ldots, T_n))$	$T_1 : \mathrm{Set}, \ldots, T_n : \mathrm{Set}, T : \mathrm{Set}, x : T,$
$x = ((t_{11} : T_{11}, \ldots, t_{1n} : T_{1n})$ $\mid \cdots \mid (t_{m1} : T_{m1}, \ldots, t_{mn} : T_{mn}))$	$x \in T \wedge (T = \mathrm{datalist}(T_{11}, \ldots, T_{1n})$ $\vee \cdots \vee T = \mathrm{datalist}(T_{m1}, \ldots, T_{mn}))$	$T_{11} : \mathrm{Set}, \ldots, T_{1n} : \mathrm{Set}, \ldots,$ $T_{m1} : \mathrm{Set}, \ldots, T_{mn} : \mathrm{Set}, T : \mathrm{Set}, x : T,$ $t_{11} : T_{11}, \ldots, t_{mn} : T_{mn}$
$x \neq ((t_{11} : T_{11}, \ldots, t_{1n} : T_{1n})$ $\mid \cdots \mid (t_{m1} : T_{m1}, \ldots, t_{mn} : T_{mn}))$	$x \in T \wedge (T \neq \mathrm{datalist}(T_{11}, \ldots, T_{1n})$ $\wedge \cdots \wedge T \neq \mathrm{datalist}(T_{m1}, \ldots, T_{mn}))$	$T_{11} : \mathrm{Set}, \ldots, T_{1n} : \mathrm{Set}, \ldots,$ $T_{m1} : \mathrm{Set}, \ldots, T_{mn} : \mathrm{Set}, T : \mathrm{Set}, x : T$
$x = (\ldots, t : \mathrm{int}, \ldots)$	$x = (\ldots, t : \mathbb{Z}, \ldots) \wedge (\mathrm{MinInt} \leqslant t \leqslant \mathrm{MaxInt})$	$\ldots, t : \mathbb{Z}, \ldots, x : \mathrm{datalist}(\ldots, \mathbb{Z}, \ldots)$
$x = (\ldots, t : \mathrm{float}, \ldots)$	$x = (\ldots, t : \mathbb{R}, \ldots) \wedge$ $(\mathrm{MinFloat} \leqslant t \leqslant \mathrm{MaxFloat})$	$\ldots, t : \mathbb{R}, \ldots, x : \mathrm{datalist}(\ldots, \mathbb{R}, \ldots)$

Контекст в этом случает будет определяться так:

$$A_1 : \mathrm{Set}, \ldots, A_m : \mathrm{Set}, a_1 : A_1, \ldots, a_m : A_m, T_2 : \mathrm{Set}, \ldots, T_n : \mathrm{Set}, t_2 : T_2, \ldots, t_n : T_n,$$
$$x : \mathrm{datalist}\,(\mathrm{datalist}\,(A_1, \ldots, A_n), T_2, \ldots, T_n).$$

2.3.2 Аксиомы

В качестве аксиом используются тройки Хоара для всех встроенных функций языка. Ввиду того, что встроенные функции определены для всех типов аргументов, то, в зависимости от типа аргумента, у каждой функции может быть несколько путей выполнения, приводящих к разному результату вычислений. Поэтому каждой встроенной функции соответствует несколько аксиом, их количество равно числу различных путей выполнения функции. При построении аксиом используются семантические правила языка Пифагор [16, 149, 173], которые можно представить в виде ориентированных деревьев с вершинами двух типов.

Проиллюстрируем вышесказанное на примере аксиом для функции `dup`, которая создаёт список из одинаковых элементов. В качества входного значения функция принимает список из двух элементов, первый из которых — любой элемент a, а второй — целочисленная константа n. Результат — список из n копий элемента a. Для любого другого аргумента

функция вернёт ошибку [16].

Семантическое правило для функции `dup` (см. приложение Б), представленное в виде дерева приведено на рисунке 2.3. Каждый путь от корня к листу дерева соответствует одной аксиоме. В листьях дерева находятся результаты выполнения функции, получаемые при равенстве выражений, записанных в вершинах тёмного цвета, значениям, приписанных к дугами, выходящими из этих вершин. Если для некоторого пути приравнять все выражения из темных вершин к соответствующим значениям, приписанным дугам, и объединить все вновь полученные выражения конъюнкцией, то получается предусловие тройки. А постусловие — содержимое листа.

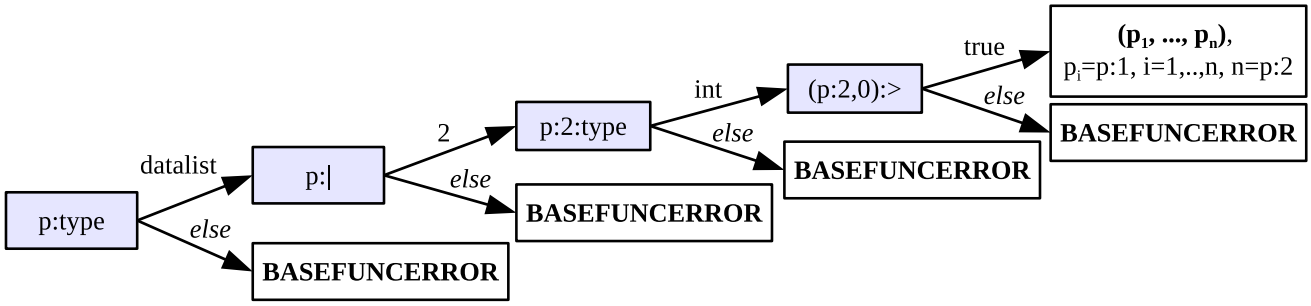


Рисунок 2.3 — Семантическое правило для функции дублирования (`p:dup`) в виде дерева

Тогда, при переводе всех выражений на язык спецификации, получаем следующие аксиомы для функции дублирования:

1. $(p \in \text{datalist}\, L) \wedge (L \in \text{list}\, \text{Set}) \wedge (\text{hlength}(p) = 2) \wedge (p[2] \in \text{int}) \wedge (p[2] > 0)$ `p:dup` $\rightarrow r$ $(r \in \text{datalist}\, L_1) \wedge (L_1 \in \text{list}\, \text{Set}) \wedge (n = \text{hlength}(r)) \wedge (n = p[2]) \wedge \forall i : \mathbb{N}.\, (1 \leqslant i \leqslant n) \Rightarrow (r[i] = p[1])$,
2. $(p \notin \text{datalist}\, L) \wedge (L \in \text{list}\, \text{Set}))$ `p:dup` $\rightarrow r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$,
3. $(p \in \text{datalist}\, L) \wedge (L \in \text{list}\, \text{Set})) \wedge (\text{hlength}(p) \neq 2)$ `p:dup` $\rightarrow r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$,
4. $(p \in \text{datalist}\, L) \wedge (L \in \text{list}\, \text{Set}) \wedge (\text{hlength}(p) = 2) \wedge (p[2] \notin int)$ `p:dup` $\rightarrow r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$,
5. $(p \in \text{datalist}\, L) \wedge (L \in \text{list}\, \text{Set}) \wedge (\text{hlength}(p) = 2) \wedge (p[2] \in int) (p[2] \leqslant 0)$ `p:dup` $\rightarrow r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

Аксиомы остальных встроенных функций приведены в приложении Д.

Замечание. Количество аксиом для каждой встроенной функции определяется структурой семантического правила. Данное деление весьма условно и исходит из удобства исполь-

зования так, чтобы одному типу аргумента соответствовал один тип результата. Язык спецификации допускает более сложные формулировки, например, можно описать, что несколько типов входных аргументов приводят к одному результату. Так для функции `dup` все пути, приводящие к одинаковому результату (ошибки `BASEFUNCERROR`), можно объединить в одну аксиому с помощью дизъюнкции:

$$\boxed{\begin{array}{l}((p \notin \text{datalist}\, L) \wedge (L \in \text{list Set}))\vee \\ \Big((p \in \text{datalist}\, L) \wedge (L \in \text{list Set})\wedge \\ \big((\text{hlength}(p) \neq 2) \vee \big((\text{hlength}(p) = 2)\wedge \\ (p[2] \notin \text{int} \vee ((p[2] \in \text{int}) \wedge (p[2] \leqslant 0)))\big)\big)\Big)\end{array}} \texttt{p:dup} \rightarrow r \boxed{r = \texttt{BASEFUNCERROR}}.$$

Также возможно вообще сократить количество аксиом до одной (и не использовать условие применимости) или двух (когда в одной аксиоме все «правильные» значения аргумента, а во второй — «ошибочные»). Однако это приводит к значительному усложнению формулы и снижению наглядности доказательства, поэтому такой приём не используется в данной работе.

2.3.3 Правила вывода

Для определения истинности произвольных троек Хоара вводятся правила вывода. Они позволяют связать аксиомы для встроенных функций с произвольными программами, описывая схему преобразования композиций операторов и позволяя получать (выводить) теоремы из имеющихся аксиом и уже доказанных теорем. Совокупность аксиом и правил вывода формирует аксиоматическую систему Хоара для языка Пифагор.

Правила вывода определяют порядок преобразования тройки Хоара в формулу на языке спецификации, далее истинность этой формулы проверяется с помощью аппарата математической логики [150, 151]. Потребуем, чтобы процесс преобразования тройки в формулу определялся правилами вывода однозначно, то есть на каждом шаге можно было применить только одно правило. Ввести правила вывода с этим условием можно двумя способами [1]:

1. использовать правила *прямого прослеживания*, когда правила вывода применяются начиная от аргумента функции и заканчивая её результатом (свёрху вниз по информационному графу);

2. использовать правила *обратного прослеживания*, когда правила вывода применяются начиная от результата вычисления функции и заканчивая её аргументом (снизу вверх по информационному графу).

Используем первый вариант. В этом случае правила вывода позволяют преобразовывать произвольную тройку Хоара, начиная от функции, применяемой непосредственно к

входному аргументу, то есть функции, которая вычисляется первой при выполнении программы. В случае языка Пифагор таких функций может быть несколько, тогда из них выбирается произвольная. Это не делает процесс преобразования недетерминированным, так как параллельные ветви графа выполняются независимо друг от друга.

В общем случае правило прямого прослеживания для некоторой функции с кодом «$\mathtt{x}:\mathtt{F}_1:\mathtt{F}$» имеет следующий вид:

$$\left\{ \boxed{P_1(x_1)}\ \mathtt{x}_1:\mathtt{F} \to r\ \boxed{Q(r)}\ ,\ A_1,\ A_2 \right\} \quad \vdash \quad \boxed{P(x)}\ \mathtt{x}:\mathtt{F}_1:\mathtt{F} \to r\ \boxed{Q(r)}\ , \tag{2.1}$$

где:

$$A_1 \equiv \boxed{\varphi(x)}\ \mathtt{x}:\mathtt{F}_1 \to x_1\ \boxed{\psi(x_1)}\ , \quad A_2 \equiv (P(x) \Rightarrow \varphi(x)),$$
$$P_1(x_1) \equiv ((P(x) \Rightarrow \varphi(x)) \Rightarrow \psi(x_1)),$$

$\mathtt{x}$ — входной аргумент, $\mathtt{F}_1$ — функция, применяемая непосредственно к входному аргументу, $\mathtt{F}$ — остальная часть программы, которая может содержать другие функции, применяемые непосредственно к аргументу $\mathtt{x}$, A_1 — одна из аксиом для функции $\mathtt{F}_1$, символ выводимости $\vdash$ обозначает, что тройка Хоара справа истинна, если истинны формулы слева.

Согласно этому правилу, при применении аксиомы A_1 к функции $\mathtt{F}_1$, тройка Хоара (справа) преобразуется в новую тройку (слева) с более «короткой» программой, при этом предусловие $P(x)$ изменяется на $P_1(x_1)$, а исходный аргумент x заменяется на x_1 — результат применения функции $\mathtt{F}_1$ к x. Истинность формулы A_2 — необходимое условие применения аксиомы A_1.

Получаем, что при последовательном применении правила прямого прослеживания происходит «сокращение» или «свёртка» программы, и после анализа всех операторов можно получить тройку Хоара с «пустой» программой: $\boxed{\mathrm{P}}\ \boxed{\mathrm{Q}}$. Следующее правило позволяет преобразовать эту тройку в формулу на языке спецификации:

$$(P \Rightarrow Q) \vdash \boxed{\mathrm{P}}\ \boxed{\mathrm{Q}}\ . \tag{2.2}$$

Таким образом, с помощью преобразований произвольных троек Хоара на основе правил вывода можно получить формулу на языке спецификации, истинность которой доказывается в рамках исчисления конструкций. Если эта формула истинна, то истинна и исходная тройка Хоара, откуда следует корректность программы.

Рассмотрим, как изменяется предусловие программы при применении правила прямого прослеживания на примере следующей тройки Хоара:

$$\boxed{P(x)}\ \mathtt{x}:\mathtt{f}:\mathtt{g} \to r\ \boxed{Q(r)} \tag{2.3}$$

Пусть для функции $\mathtt{f}$ заданы аксиомы:

$$\boxed{P_i(x_a)}\ \mathtt{x}_a\mathtt{:f} \to r_a\ \boxed{Q_i(r_a)}\ ,\ i = 1, 2, \ldots, n,$$

где x, r — входной и выходной аргумент программы, а x_a и r_a — входной и выходной аргумент аксиом функции `f`, n — количество аксиом для функции `f`.

Применение правила прямого прослеживания состоит в следующем.

1. Введём уникальный идентификатор x_1 для обозначения значения выражения `x:f`.

2. Пусть для некоторых i выполнено условие:

$$P(x) \Rightarrow P_i(x). \tag{2.4}$$

Если формула выполнима (не обязательно тождественно истинна), то входной аргумент x удовлетворяет предусловию аксиомы, и, следовательно, она может быть использована в преобразованиях по правилу прямого прослеживания. Отбрасываем все аксиомы, не удовлетворяющие (2.4), в результате остаётся k аксиом (которые перенумеруем от 1 до k). Эти аксиомы соответствуют k возможным путям выполнения программы, каждый из которых рассматривается отдельно. Поскольку функции в языке Пифагор полностью определены, всегда найдётся хотя бы одна аксиома, которую можно использовать.

3. Произведём преобразование тройки Хоара, «сворачивая» программу, заменяя предусловие на выражение $(P(x) \Rightarrow P_i(x)) \Rightarrow Q_i(x_1)$, $i = 1, 2, \ldots, k$, фрагмент кода `x:f` заменяется на идентификатор результата x_1, введенный в пункте 1. В результате получается k новых троек Хоара:

$$\boxed{(P(x) \Rightarrow P_i(x)) \Rightarrow Q_i(x_1)}\ \mathtt{x_1:g} \to r\ \boxed{Q(r)}\ ,\ i = 1, 2, \ldots, k.$$

Исходная тройка Хоара будет истинной, если истинны все k полученных троек.

Замечание 1. При проверке применимости аксиомы и применении правила прямого прослеживания необходимо согласовать идентификаторы входного аргумента и результата у аксиомы с соответствующими идентификаторами у функции. Так, в рассматриваемом примере в аксиому вместо x_a подставляется x, а вместо r_a подставляется x_1.

Замечание 2. Аксиомы каждой функции заданы таким образом, что их предусловия определяют непересекающиеся множества значений аргументов, а объединение множеств значений аргументов предусловий всех аксиом даёт множество всевозможных значений аргумента. Поэтому, если условие (2.4) тождественно истинно для некоторой аксиомы, то эта аксиома полностью определяет единственный путь выполнения программы, а все остальные аксиомы неприменимы. Условие взаимоисключения предусловий аксиом не является обязательным, пользователь может нарушить его, определяя и доказывая теоремы для своих функций. Однако для полного определения функции необходимо, чтобы объединение предусловий всех теорем описывало всё множество значений аргументов. Это утверждение со-

ответствует следующей формуле: $\bigvee\limits_{i=1}^{n} P_i(x) = \text{true}$, где P_i — предусловие i-ой теоремы, n — число теорем для данной функции, x — всевозможные значения аргумента. Если это не так, то добавляется ещё одна теорема с предусловием $\neg \left(\bigvee\limits_{i=1}^{n} P_i \right)$ и постусловием $r \in \text{error}$, где r — результат вычисления функции. Если при доказательстве корректности некоторой программы для этой теоремы выполнено условие (2.4), то программа сразу считается некорректной.

Замечание 3. В любой формуле обязательно должен указываться тип всех свободных переменных (см. раздел 2.3.1). Однако для сокращения формул и увеличения наглядности в данной работе тип переменных результата вычисления функции может не указываться, если он понятен из контекста.

Ввиду того, что при прямом прослеживании предусловие P полагается истинным по определению, выражение $P \Rightarrow Q$ истинно, если истинно более сильное условие $P \wedge Q$, получаем, что выполнение предусловия $(P \Rightarrow P_i) \Rightarrow Q_i$ следует из выполнения $P \wedge P_i \wedge Q_i$. Поэтому в правиле прямого прослеживания (2.1) можно изменить $P_1(x_1)$ следующим образом: $P_1(x_1) \equiv (P(x) \wedge \varphi(x) \wedge \psi(x_1))$.

Например, возьмём одну из троек, полученных при применении правила прямого прослеживания для функции `f` на основе аксиомы i:

$$\boxed{P(x) \wedge P_i(x) \wedge Q_i(x_1)}\ \mathtt{x_1 : g} \to r\ \boxed{Q(r)}\,, \tag{2.5}$$

и применим к ней правило прямого прослеживания для функции `g` на основе одной из допустимых аксиом j. Получаем тройку Хоара с «пустой» программой:

$$\boxed{P(x) \wedge P_i(x) \wedge Q_i(x_1) \wedge \tilde{P}_j(x_1) \wedge \tilde{Q}_j(r)}\ \boxed{Q(r)}\,,$$

где $\tilde{P}_j$ и $\tilde{Q}_j$ — предусловие и постусловие j-ой аксиомы функции g. К данной тройке можно применить правило преобразования в формулу (2.2) и проверить полученную формулу на истинность.

Надёжность системы. Покажем, что система Хоара с описанными правилами надёжна, то есть из истинных троек можно получить только истинные тройки. Рассмотрим правило прямого прослеживания (2.1). Пусть A_1, A_2 и $A \equiv \boxed{P_1(x_1)}\ \mathtt{x_1 : F} \to r\ \boxed{Q(r)}$ являются тождественно истинными. Если $P(x)$ истинно, то по A_2 истинна $\varphi(x)$. Далее из истинности A_1 следует истинность $\psi(F_1(x))$, и следовательно, $P_1(F_1(x))$ тоже истинно. В результате, по A формула $Q(F(F_1(x))) = Q(r)$ истинна. Доказательство для правила (2.2) является тавтологией. Также отметим, что в языке отсутствуют побочные эффекты, а для всех вводимых идентификаторов используется принцип единственного присваивания.

Данную систему можно считать полной в том смысле, что любую тройку всегда можно свести к формуле на языке логики конечным числом преобразований [136]. Действительно,

код программы конечен (рекурсивные вызовы представлены именем функции, поступающим на оператор интерпретации, и, в случае завершения функции, считаются корректными по индуктивному предположению, подробнее см. раздел 3.1.1). Программа состоит из операторов и встроенных функций. Количество аксиом для встроенных функций конечно, и для любой функции, поступающей на оператор интерпретации, всегда найдётся хотя бы одна аксиома, применимая для разметки (это следует из тотальности встроенных функций). Значит, за конечное число шагов любая программа может быть «свёрнута» и преобразована в формулу на языке логической спецификации.

Выводы. Набор аксиом для базовых функций языка позволяет использовать их для анализа корректности произвольной ФФП программы. Если программа состоит из нескольких функций, то каждая функция рассматривается в отдельности. В зависимости от значений входных данных программа может иметь различные пути выполнения. Аксиомы встроенных функций полностью определяют дерево всех путей выполнения программы $\mathfrak{T}_0$. Количество различных вариантов выполнения программы для каждой функции равно количеству аксиом этой функции. Если на входные данные наложены ограничения (предусловие программы), то часть ветвей в дереве становятся недостижимыми. Эти ветви соответствуют тем аксиомам, предусловие которых не следует из предусловия программы. Откидывая недостижимые пути, получаем новое дерево $\mathfrak{T}_1$, которое является поддеревом $\mathfrak{T}_0$. Каждый путь дерева $\mathfrak{T}_1$ можно рассмотреть отдельно и преобразовать в формулу на языке спецификации по правилам вывода. В результате таких преобразований получается k формул, где k соответствует числу листьев в дереве $\mathfrak{T}_1$. Из истинности всех полученных формул следует, что программа корректна. В противном случае программа некорректна, а ошибки присутствуют в тех ветвях, которым соответствуют не тождественно истинные формулы. Таким образом, доказательство корректности программы сводится к доказательству истинности конечного числа формул.

2.4 Преобразования информационного графа с разметкой

Программу на языке Пифагор и все преобразования троек Хоара нагляднее отображать в виде информационного графа [140]. Информационный граф программы, дуги которого помечены формулами на языке спецификации, будем называть *информационным графом с разметкой* (ИГР). Если в информационном графе размечены только входная и выходная дуги, то граф соответствует тройке Хоара, в которой программа представляется графом, разметка входной дуги есть предусловие, выходной — постусловие. Зададим над информационным графом с разметкой следующие преобразования [172, 179]:

1. разметка дуг;

2. свёртка программы;

3. изменение информационного графа программы:

   3.1. эквивалентное преобразование,

   3.2. расщепление;

Если использовать информационный граф, то процесс доказательства корректности ФПП программы можно рассматривать как последовательность преобразований *исходного ИГР*, то есть ИГР, которому соответствует исходная тройка Хоара. В результате преобразований из исходного ИГР получают множество *полностью размеченных ИГР*, к каждой дуге такого графа приписана одна формула, а к выходной дуге — две: формула, полученная в результате разметки, и постусловие. Эти графы с помощью свёртки преобразуются в тройки Хоара с пустой программой, которые можно напрямую преобразовать в формулы на языке спецификации для проверки истинности. Если все полученные формулы истинны, то и исходная тройка истинна, а программа корректна.

2.4.1 Разметка дуг

Каждое преобразование тройки Хоара по правилу прямого прослеживания (2.1) для некоторой функции можно отобразить на информационном графе приписыванием к выходной дуге рассматриваемой функции новой формулы, а все формулы ранее рассмотренных дуг будут сохраняться в неизменном виде. В этом случае преобразования тройки в формулу по правилу (2.2) можно выполнить тогда, когда все дуги графа окажутся размеченными, а к выходной дуге программы будут приписаны две формулы: постусловие и формула преобразований самой внешней функции.

В качестве примера на рисунке 2.4а приведён исходный ИГР для тройки Хоара (2.3). На рисунке 2.4б приведён ИГР после первого применения правила прямого прослеживания для функции `f` (на основе одной из допустимых аксиом i). К дуге x_1 приписывается формула $P_i \wedge Q_i$, полученная из предусловия и постусловия i-ой аксиомы. При этом предусловие P, приписанное к входной дуге, остаётся неизменным. Следующее применение правила прямого прослеживания для функции `g` (на основе одной из допустимых аксиом j) приведёт к появлению формулы $\tilde{P}_j \wedge \tilde{Q}_j$ у дуги r, где $\tilde{P}_j$ и $\tilde{Q}_j$ — предусловие и постусловие j-ой аксиомы функции g. Получается полностью размеченный ИГР (рисунок 2.4в).

Имеем следующую аналогию. В работе [16] динамика функционирования модели вычислений описывается с помощью разметки дуг графа флагами готовности данных. Оператор начинает выполняться, когда размечены все его входные дуги, а появление разметки на его выходной дуге происходит после завершения вычисления и получения результата.

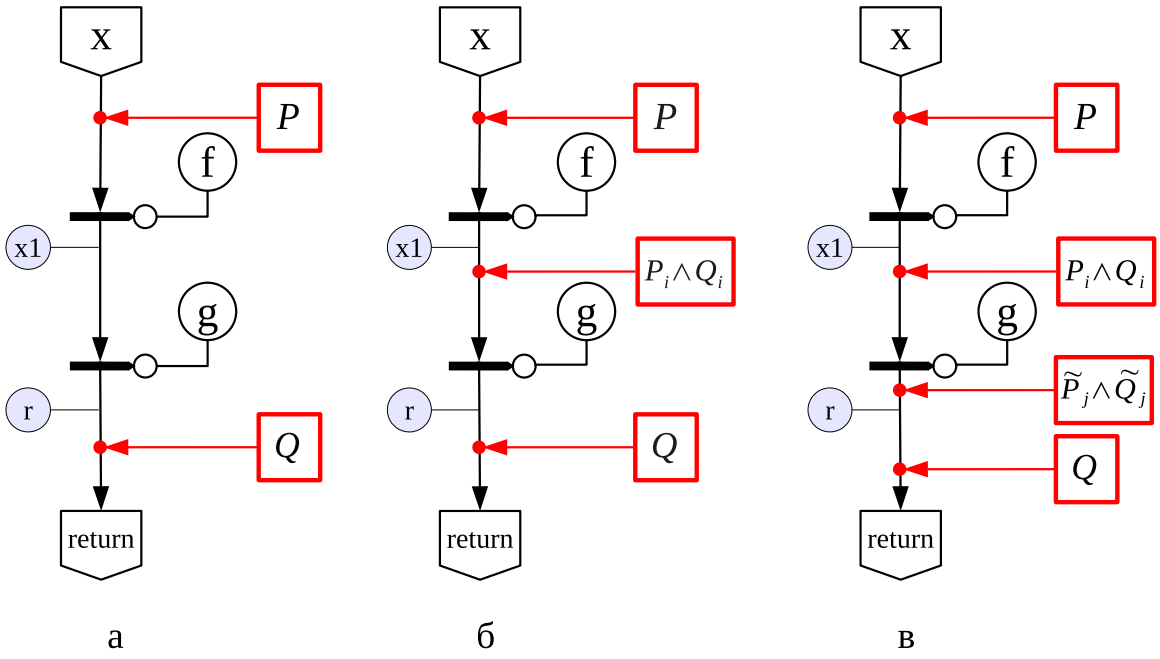


Рисунок 2.4 — Преобразования информационного графа с разметкой для функции с кодом «`x:f:g`»; а — исходный ИГР с введёнными идентификаторами выходных дуг операторов интерпретации; б — ИГР после первого применения правила прямого прослеживания; в — полностью размеченный ИГР

Граф будет полностью размечен при окончании вычисления функции. Аналогично происходит разметка дуг графа формулами при доказательстве корректности функции. Изначально всем дугам информационного графа присваиваются идентификаторы. Все идентификаторы должны быть различными. Считаем, что имя оператора совпадает с именем его входной дуги. Будем говорить, что *оператор готов к разметке*, если размечены все его входные дуги. Выходная дуга каждого готового оператора размечается формулой. Для обозначения результирующего значения в формуле используется идентификатор дуги.

Перечислим правила разметки всех типов операторов.

1. Выходная дуга оператора интерпретации размечается по «правилу прямого прослеживания» (2.1) (раздел 2.3.1).

2. К выходной дуге константного оператора приписывается формула вида $(x \in T) \wedge$ $\wedge (x = a)$, где x — идентификатор дуги, a — значение константы, а T — её тип.

3. Оператор копирования просто копирует данные, поэтому значения на его выходных дугах совпадают со значением входной дуги. Для того, чтобы не вводить новых идентификаторов, оператор копирования считается автоматически размеченным, если размечена его входная дуга. Разметка на выходных дугах не ставится, а используется разметка входной дуги.

4. Список данных размечается приписыванием формулы вида $x = (a_1 : A_1, \ldots, a_n : A_n)$, где a_i — идентификатор i-ой входной дуги с типом A_i.

5. Параллельный список не может быть входным аргументом функции, так как при появлении параллельного списка на операторе интерпретации граф преобразуется по правилам эквивалентных преобразований (см. раздел 2.1.2). Однако он может быть результатом выполнения функции, тогда параллельный список присутствует в формуле приписанной к выходной дуге этой функции.

6. Задержанный список является константой и размечается так же, как константный оператор.

Таким образом, все виды операторов, кроме оператора интерпретации, могут быть размечены автоматически.

Иерархия формул. На множестве формул, помечающих дуги одного ИГР, введём отношение иерархии, назвав формулу, помечающую дугу (a, b), родительской по отношению к формуле, помечающей дугу (b, c) для узлов графа a, b, c.

При разметке дуг информационный граф программы не изменяется, и в случае, если в результате разметки какой-либо дуги получается несколько новых ИГР, то они будут отличаться между собой только формулой у размечаемой дуги. Эта ситуация возникает в случае, если несколько аксиом одной функции удовлетворяют условию (2.4). Для компактного отображения, эти ИГР можно объединить в один, у которого дуга будет размечена несколькими формулами одновременно. Отношение иерархии между формулами позволяет затем разделить компактное представление на изначальные ИГР.

Проиллюстрируем вышесказанное на примере. На рисунке 2.5 приведена часть графа некоторой программы, у которой четыре оператора соединены информационными связями. Узел 3 принимает входные данные с узлов 1 и 2, узел 4 принимает данные от узла 3. Вначале дуги в графе не размечены.

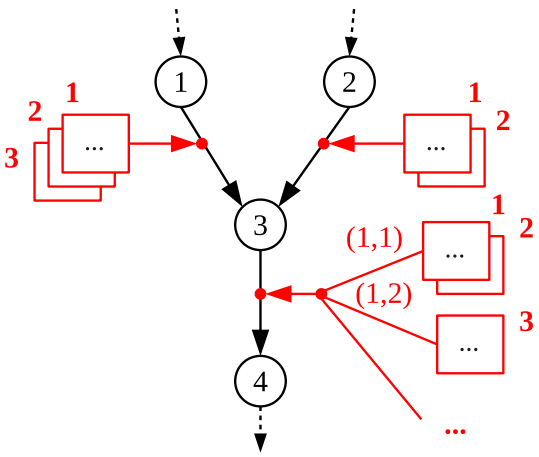


Рисунок 2.5 — Схематичное изображение разметки дуг графа формулами; рядом с формулами указаны их номера, в круглых скобках указаны номера родительских формул

На первом шаге разметим дугу $(1,3)$. Если для оператора 1 применимо три аксиомы, то в результате преобразования получится три новых ИГР, которые компактно будут отображаться как исходный граф, у которого к дуге $(1,3)$ приписаны три формулы. На следующем шаге разметим дугу $(2,3)$ каждого из трёх ИГР. Число применимых аксиом для каждого из трех ИГР первого шага преобразований одинаково, так как операторы 1 и 2 независимы. Если при преобразовании трех ИГР для оператора 2 применимо две аксиомы, то в результате преобразования получится шесть новых графов. В компактном отображении этих шести графов к дуге $(2,3)$ будут приписаны две формулы. Оператор 3 зависит от операторов 1 и 2, поэтому к каждому из шести полученных ИГР может быть применено разное количество аксиом. Пусть для первого ИГР это число соответствует двум (он распадется на два новых ИГР), для всех остальных — единице. В результате разметки получится семь графов, это число соответствует общему числу применимых аксиом и количеству формул, которые будут приписаны к дуге $(3,4)$. На рисунке 2.5 индексы (i,j) возле формул, помечающих дугу $(3,4)$, обозначают номер родительской формулы i, помечающей дугу $(1,3)$, и номер родительской формулы j, помечающей дугу $(2,3)$.

Разделение формул. Рассмотрим некоторый ИГР. Формула f, приписанная к выходной дуге некоторого оператора *op* характеризует множество значений, которые может возвращать этот оператор. Данное множество можно разделить на n подмножеств (не обязательно непрекращающиеся) и описать каждое из подмножеств своей формулой f_n. Тогда рассматриваемый ИГР будет истинен, если истинны n ИГР, у каждого из которых к дуге *op* приписана одна из формул f_i, $i = 1, 2, \ldots, n$. Эти ИГР изображаются в компактной форме приписыванием к дуге *op* n формул f_i, $i = 1, 2, \ldots, n$. Назовём эту операцию «разделением формул».

2.4.2 Свёртка

Второй тип преобразований — свёртка программы. Эта операция позволяет преобразовать частично или полностью размеченный ИГР в тройку Хоара. Изначально только исходный ИГР соответствует тройке Хоара (например, тройка (2.3) и ИГР на рисунке 2.4а). При дальнейшей разметке дуг графа требуется его преобразование для получения тройки (например, тройка (2.5) и ИГР на рисунке 2.4б).

Общая идея свёртки заключается в том, что часть подграфа с размеченными дугами заменяется на идентификатор выходной дуги данного подграфа, а формулы, размечающие дуги подграфа, объединяются с помощью конъюнкции в новое предусловие. Операция свёртки может проводиться для любого узла, выходная дуга которого размечена формулой, если

эта формула не является предусловием или постусловием.

В общем случае свёртка состоит в следующем.

1. Для выбранного узла x с размеченной выходной дугой ищется порождающий его подргаф G_x — подграф, содержащий все узлы, из которых достижим рассматриваемый узел x, включая входной аргумент функции.

2. Поражающий подграф G_x удаляется из графа функции. При этом рассматриваемый узел x заменяется на свой идентификатор, который обозначается как входной аргумент. Если из одного из узлов y подграфа G_x выходит дуга (y, z) в узел $z \notin G_x$, то узел y также заменяется на свой идентификатор и преобразуется во входной аргумент.

3. Поскольку выходная дуга x размечена, то размечены и все дуги графа G_x. Все формулы, которые приписаны к дугам порождающего подграфа G_x и к дуге, выходящей из узла x, объединяются в одну с помощью конъюнкции и в качестве предусловия приписываются к дуге аргумента. Возможен случай, когда в графе появляется несколько входных аргументов, тогда новое предусловие приписывается одновременно ко всем их дугам. Заметим, что исходное предусловие обязательно входит в новое предусловие, так как все пути начинаются от входного аргумента.

Схематично пример свёртки приведён на рисунке 2.6. Слева изображён граф, у которого для проведения свёртки выбран узел x_3. В порождающий его подграф входят четыре узла (отмечены серым). В порождающем подграфе из узлов arg и x_2 выходят дуги в узлы соответственно y_1 и x_4, не лежащие в удаляемом подграфе. Поэтому узлы arg, x_2 и рассматриваемый узел x_3 заменяются на свои идентификаторы, и весь подграф удаляется. В результирующем графе (на рисунке справа) имеется три входных аргумента, и к каждому из них приписана одна формула, являющаяся конъюнкцией всех формул порождающего подграфа узла x_3.

Свёртку можно проводить при частичной или полной разметке дуг графа программы. Если провести свёртку над всеми размеченными дугами ИГР, то полученному в результате ИГР будет соответствовать тройка Хоара. Назовем такую свёртку *полной*. Если в ИГР все дуги размечены, то при проведении полной свёртки от графа остаётся одна переменная, которая является результатом работы программы. Её свойства описаны в предусловии, в котором объединены формулы всех дуг. Такая программа называется *пустой*, и для определения её корректности достаточно проверить следование постусловия из предусловия (использовать правило преобразования тройки в формулу (2.2) и доказать истинность формулы).

Замечание 1. Важно отметить, что перед разметкой выходной дуги оператора интерпретации необходимо выполнить свёртку его входного аргумента, так как при проверке условия применимости аксиомы (2.4) в качестве $P(x)$ берется не исходное предусловие функции,

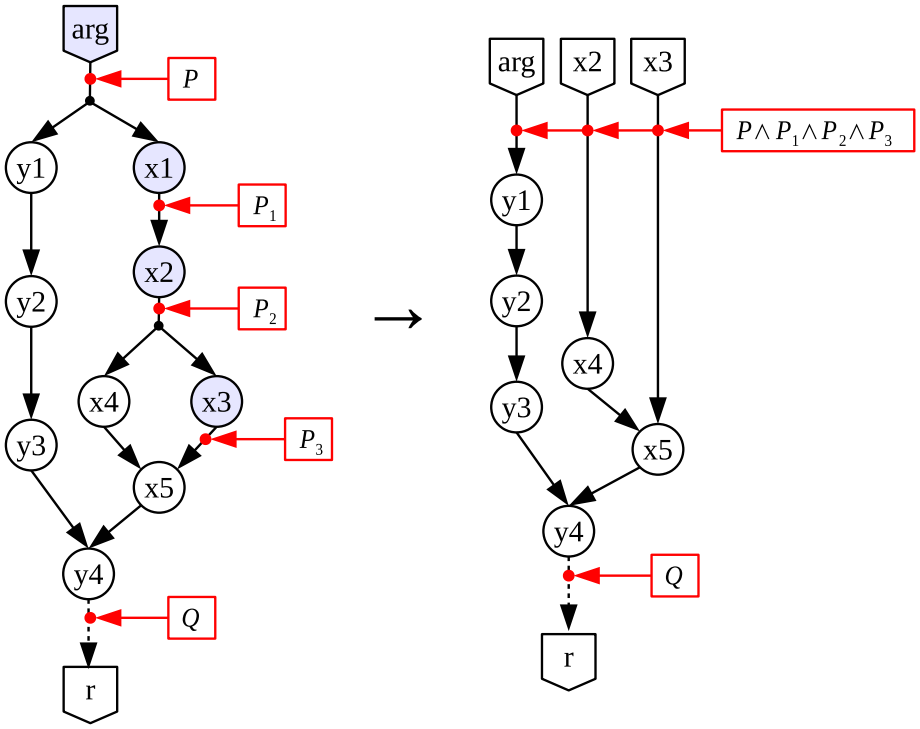


Рисунок 2.6 — Схематичное изображение свёртки для узла x_3, узлы порождающего подграфа отмечены серым

а предусловие тройки, полученной при полной свёртке размечаемого ИГР. Далее в работе считаем, что операция свёртки проводится неявно, при проверке условия применимости аксиомы (2.4).

Замечание 2. При проведении свёртки возможно упрощение формул за счёт удаления части переменных выходных дуг операторов, если они выражаются через переменные входных дуг. Например, если при свёртке, изображённой на рисунке 2.6, $x_3 = f_3(x_2)$, $x_2 = f_2(x_1)$, $x_1 = f_1(x)$, где f_i — некоторые функции, $i = 1, 2, 3$, то $x_3 = f_3(f_2(f_1(x)))$. Тогда переменные x_1, x_2 можно исключить из формул предусловия получаемой при свёртке тройки. Также при свёртке полностью размеченного графа иногда возможно выразить результат r через исходный аргумент x, без использования промежуточных идентификаторов дуг. Далее в работе такие преобразования проводятся неявно там, где это возможно.

Пример. Рассмотрим доказательство корректности программы на примере простой функции, вычисляющей значение выражения $a \cdot b + c$, где a, b и c — целые числа. Имеем программу на языке Пифагор:

```
Func << funcdef x {
    a << x:1;
    b << x:2;
    c << x:3;
   ((a, b):*,c):+  >>  return
}
```

Пользователь задаёт для неё следующую тройку Хоара:

$$\boxed{x = (a : \text{int}, b : \text{int}, c : \text{int})} \ \texttt{((a, b):*,c):+} \ \rightarrow r \ \boxed{(r \in \text{int}) \wedge (r = a \cdot b + c)} \ .$$

Представление данной тройки в виде информационного графа приведено на рисунке 2.7а. Для упрощения изложения присваивание идентификаторов a, b и c результатам применения функции выбора элемента из списка опускается, и аргумент x сразу представляется в виде списка (a, b, c), а дуги a, b, c считаются размеченными формулами $a \in \text{int}$, $b \in \text{int}$, $c \in \text{int}$ соответственно.

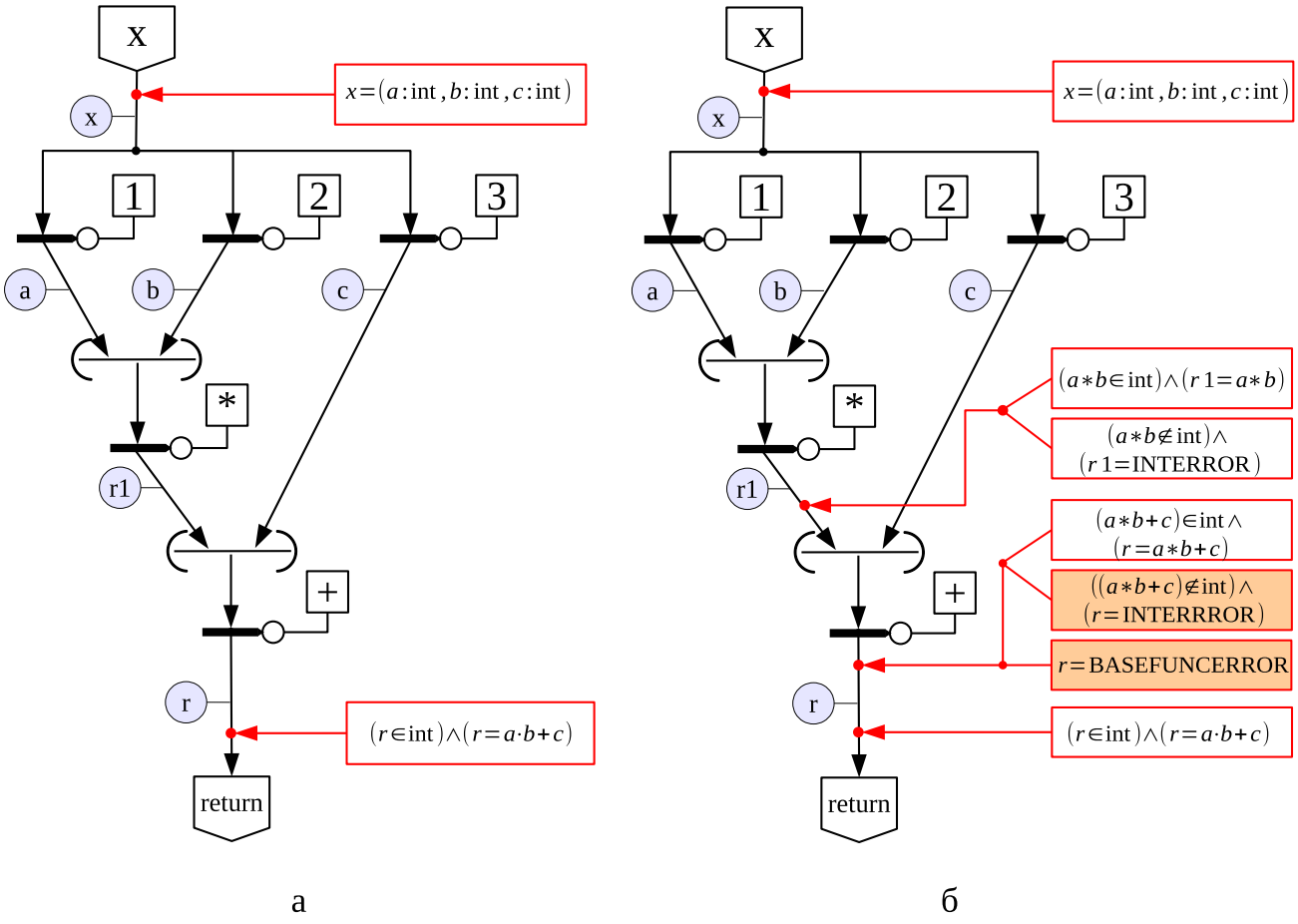


Рисунок 2.7 — Разметка формулами дуг графа функции `Func`

Замечание. При разметке информационных графов формулы указываются только у выходных дуг операторов интерпретации. Подразумевается, что дуги остальных операторов размечаются автоматически, как только разметка появляется у всех их входных дуг.

Если дуги a, b и c, размечены, то может быть размечена дуга оператора группировки в список данных `(a,b)`. Также разметка имеется у встроенной функции «*». Тогда оператор интерпретации r_1 готов к разметке.

Для функции умножения имеется шесть аксиом (аксиомы 7.1–7.6 из приложения Д). Каждая из аксиом определяет свой путь выполнения программы. Из всех аксиом необходимо

выбрать применимые. Только те пути выполнения программы могут быть реализованы, у которых предусловие аксиомы может следовать из предусловия программы (2.4). Поэтому из всех путей выполнения программы нужно исключить заведомо недостижимые, за счёт исключения части аксиом. Для этого формируются формулы (их количество соответствует числу аксиом для функции «*»), описывающие истинность условия недостижимости (то есть отрицание условия (2.4)). Данные формулы проверяются на истинность. Аксиомы, соответствующие истинным формулам, откидываются.

Ниже приведены шесть формул для каждой из аксиом:

1. $\neg\Big((a,b,c \in \text{int}) \Rightarrow (a,b \in \text{bool})\Big)$
2. $\neg\Big((a,b,c \in \text{int}) \Rightarrow (a,b,a \cdot b \in \text{int})\Big)$
3. $\neg\Big((a,b,c \in \text{int}) \Rightarrow (a,b \in \text{int}) \vee (a \cdot b \notin \text{int})\Big)$
4. $\neg\Big((a,b,c \in \text{int}) \Rightarrow \big(((a \in \text{int} \wedge b \in \text{float}) \vee (a \in \text{float} \wedge b \in \text{int}) \vee$
   $\vee (a \in \text{float} \wedge b \in \text{float})) \wedge (a \cdot b \in \text{float})\big)\Big)$
5. $\neg\Big((a,b,c \in \text{int}) \Rightarrow \big(((a \in \text{int} \wedge b \in \text{float}) \vee (a \in \text{float} \wedge b \in \text{int}) \vee$
   $\vee (a \in \text{float} \wedge b \in \text{float})) \wedge (a \cdot b \notin \text{float})\big)\Big)$
6. $\neg\Big((a,b,c \in \text{int}) \Rightarrow (a,b \notin (\text{bool} \mid \text{int} \mid \text{float}))\Big)$

Из указанных формул 1, 4, 5 и 6 всегда истинны, поэтому соответствующие аксиомы могут быть откинуты.

Можно разметить дугу r_1 формулами, применяя правило вывода на основе оставшихся двух аксиом. В результате из исходной тройки Хоара получается две новых тройки:

1. $\boxed{(a,b,c \in \text{int}) \wedge (a \cdot b \in \text{int}) \wedge (r_1 = a \cdot b)}$ `(r1,c):+` $\rightarrow r$ $\boxed{(r = a \cdot b + c)}$;
2. $\boxed{(a,b,c \in \text{int}) \wedge \neg(a \cdot b \in \text{int}) \wedge (r_1 =$ `INTERROR`$)}$ `(r1,c):+` $\rightarrow r$ $\boxed{(r = a \cdot b + c)}$.

Разметка дуги r_1 формулами приведена на рисунке 2.7б. Приписывание нескольких формул к одной дуге соответствует «компактному» изображению нескольких троек с одинаковыми графами.

Далее в программе формируется список данных `(r1,c)`. И следующая функция, применяемая непосредственно к аргументу преобразованной программы `(r1,c):+` — функция сложения «+».

Аксиомы функции «+» аналогичны аксиомам функции умножения «*», если в них заменить знак $\wedge$ на $\vee$, а $\cdot$ на $+$ (аксиомы 5.1–5.6 из приложения Д).

Формулы применимости данных аксиом приведены ниже:

1.1. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = a \cdot b)) \Rightarrow (r_1, c \in \text{bool})\Big)$;

1.2. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = a \cdot b)) \Rightarrow (r_1, c, (r_1 + c) \in \text{int})\Big)$;

1.3. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = a \cdot b)) \Rightarrow (r_1, c \in \text{int}) \vee (r_1 + c \notin \text{int})\Big)$;

1.4. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = a \cdot b)) \Rightarrow (((r_1 \in \text{int} \wedge c \in \text{float}) \vee (r_1 \in \text{float} \wedge c \in \text{int}) \vee$
$\vee (r_1 \in \text{float} \wedge c \in \text{float})) \wedge (r_1 + c \in \text{float}))\Big)$;

1.5. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = a \cdot b)) \Rightarrow (((r_1 \in \text{int} \wedge c \in \text{float}) \vee (r_1 \in \text{float} \wedge c \in \text{int}) \vee$
$\vee (r_1 \in \text{float} \wedge c \in \text{float})) \wedge (r_1 + c \notin \text{float}))\Big)$;

1.6. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = a \cdot b)) \Rightarrow (r_1, c \notin (\text{bool} \mid \text{int} \mid \text{float}))\Big)$;

2.1. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = \texttt{INTERROR})) \Rightarrow (r_1, c \in \text{bool})\Big)$;

2.2. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = \texttt{INTERROR})) \Rightarrow (r_1, c, (r_1 + c) \in \text{int})\Big)$;

2.3. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = \texttt{INTERROR})) \Rightarrow (r_1, c \in \text{int}) \vee (r_1 + c \notin \text{int})\Big)$;

2.4. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = \texttt{INTERROR})) \Rightarrow (((r_1 \in \text{int} \wedge c \in \text{float}) \vee (r_1 \in \text{float} \wedge c \in \text{int}) \vee$
$\vee (r_1 \in \text{float} \wedge c \in \text{float})) \wedge (r_1 + c \in \text{float}))\Big)$;

2.5. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = \texttt{INTERROR})) \Rightarrow (((r_1 \in \text{int} \wedge c \in \text{float}) \vee (r_1 \in \text{float} \wedge c \in \text{int}) \vee$
$\vee (r_1 \in \text{float} \wedge c \in \text{float})) \wedge (r_1 + c \notin \text{float}))\Big)$;

2.6. $\neg\Big(((a,b,c \in \text{int}) \wedge (r_1 = \texttt{INTERROR})) \Rightarrow (r_1, c \notin (\text{bool} \mid \text{int} \mid \text{float}))\Big)$.

Проверяя истинность данных формул, получаем, что для первой тройки остаётся две аксиомы (1.2 и 1.3), для второй — одна (2.6). Применяем правило прямого прослеживания для разметки дуги r (см. рисунок 2.7б). В результате свёртки получаем три тройки с «пустой программой»:

1. $\boxed{(a,b,c,(a \cdot b),(a \cdot b + c) \in \text{int}) \wedge (r = a \cdot b + c)}$ $\boxed{(r = a \cdot b + c)}$;
2. $\boxed{(a,b,c,(a \cdot b) \in \text{int}) \wedge \neg((a \cdot b + c) \in \text{int}) \wedge (r = \texttt{INTERROR})}$ $\boxed{(r = a \cdot b + c)}$;
3. $\boxed{\begin{array}{l}(a,b,c \in \text{int}) \wedge \neg(a \cdot b \in \text{int}) \wedge \\ (r_1 = \texttt{INTERROR}) \wedge (r = \texttt{BASEFUNCERROR})\end{array}}$ $\boxed{(r = a \cdot b + c)}$.

Полученные тройки с «пустой программой», преобразуются в формулы по правилу преобразования тройки в формулу (2.2). Программа будет корректна, если все формулы истины.

Для рассматриваемого примера с программой `((a, b):*,c):+`, истинна только первая формула (получена из тройки с «пустой программой» с соответствующим номером), оставшиеся две формулы ложны (подформулы, приводящие к ложному результату, выделены на рисунке 2.7б) . Значит, программа не является корректной.

Однако, если изменить исходную тройку Хоара, дописав ограничения на результат вычислений, программа станет корректной:

$\boxed{(a, b, c \in \text{int}) \wedge (a \cdot b \in \text{int}) \wedge (a \cdot b + c \in \text{int})}$ `((a, b):*,c):+` $\rightarrow r$ $\boxed{(r = a \cdot b + c)}$.

2.4.3 Изменение информационного графа

Третий тип преобразований связан с изменением информационного графа программы.

Эквивалентные преобразования. Данный тип преобразований осуществляется по правилам эквивалентных преобразований операторов языка Пифагор (описаны в разделе 2.1.2, таблица 2.1). В результате преобразования получается один ИГР с изменённым информационным графом. В качестве примера одного из эквивалентных преобразований, на рисунке 2.8а приведена схема «раскрытия параллельного списка». Параллельный список позволяет явно указывать, что все поступающие в него операторы могут выполняться одновременно. В графе слева параллельный список из двух элементов поступает на функциональный вход оператора интерпретации. В эквивалентном ему графе справа каждый элемент исходного параллельного списка поступает на функциональный вход своего оператора интерпретации, а уже результат их выполнения передаётся в параллельный список.

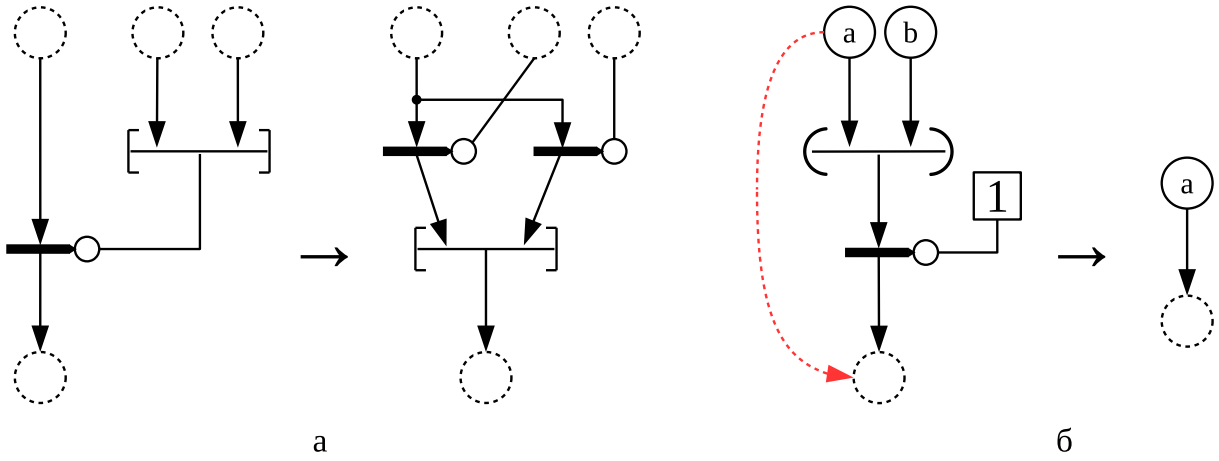


Рисунок 2.8 — Схематичное изображение эквивалентных преобразований информационного графа с разметкой; а — эквивалентное преобразование «раскрытие параллельного списка»; б — перенаправление связей при выборе элемента из списка

Замечание. Параллельный и задержанный список не могут быть аргументами функции, так как по правилам преобразований языка они «раскрываются» на операторе интерпретации. Однако эти списки могут быть возвращаемым значением функции, а значит, могут присутствовать в формуле, приписанной к дуге графа, но не присутствовать в самом графе в явном виде. При поступлении на оператор интерпретации значения, тип которого есть параллельный или задержанный список, необходимо также проводить его эквивалентное преобразование. Задержанный список просто заменяется на идентификатор своей выходной дуги.

Параллельный список можно «извлечь» из формулы, но только если известна его длина, иначе все эквивалентные преобразования проводятся на уровне формул.

Ещё один вариант эквивалентных преобразований связан с применением оператора выбора элементов из списка. Если к списку данных с известным количеством элементов применяется оператор выбора элемента, при этом номер выбираемого элемента задан числовой константой, то действие этого оператора можно рассматривать просто как перераспределение связей в графе. Например, часть кода «`(a,b):1`» эквивалентна коду «`a`» (рисунок 2.8б). Такая замена позволяет значительно упростить код функции.

Ещё один случай, в котором оператор выбора элемента позволяет сократить доказательство корректности программы — это упрощение вида входного аргумента функции. Функции на языке Пифагор принимают один аргумент, который может быть списком данных. И уже в теле функции происходит выбор элементов из этого списка для осуществления вычислений. Если x — входной аргумент функции, а в предусловие этой функции аргумент описан как список данных вида $x = (a_1 : A_1 \ldots, a_n : A_n)$, у которого указано имя каждого элемента (и, значит, известно количество элементов), то в коде функции все операции выбора элементов из этого списка `x:i`, где i — конкретное число, могут быть заменены идентификатором i-го элемента a_i, который просто приписывается к выходной дуге оператора выбора (рисунок 2.9). Этот способ использован в примере нахождения значения выражения $(a \cdot b + c)$ из раздела 2.4.1, где предусловие описывает входной аргумент формулой $x = (a : \text{int}, b : \text{int}, c : \text{int})$, и применение операторов выбора элементов из этого списка сводится к приписыванию соответствующих идентификаторов к их выходным дугам. Данный процесс не требует применения аксиом и производится автоматически.

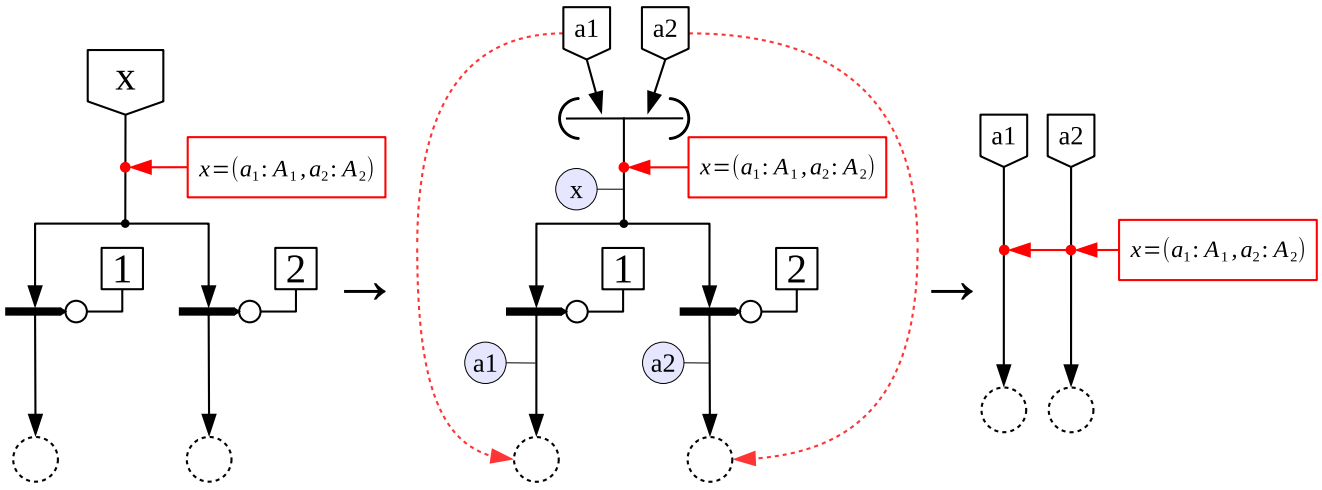


Рисунок 2.9 — Схематичное изображение эквивалентного преобразования аргумента информационного графа с разметкой на основе информации из предусловия; на последнем графе одна и таже формула предусловия приписана к двум дугам

Расщепление. Расщепление информационного графа приводит к получению двух и более ИГР с изменёнными информационными графами. Расщепление применяется в случае, когда на готовый к разметке оператор интерпретации в качестве аргумента поступает оператор группировки в список данных, а в качестве функции — переменная типа int. Данная ситуация соответствует применению функции выбора элемента из списка. При этом все варианты значений, которые может принимать переменная, определяются формулой, приписанной к входной дуге функциональной связи оператора интерпретации. В результате расщепления получается отдельный ИГР для каждого варианта значения селектора.

На рисунке 2.10 приведена схема расщепления ИГР на два графа. Слева приведён

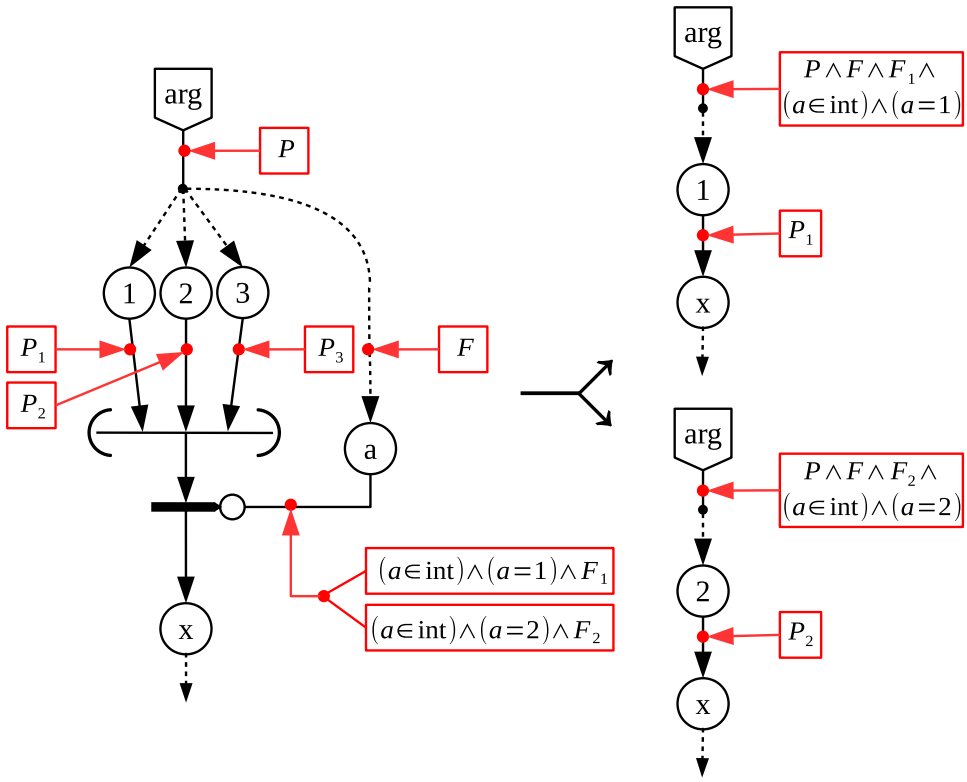


Рисунок 2.10 — Схема расщепления информационного графа с разметкой на два новых графа

ИГР, у которого готов к разметке формулами оператор интерпретации. На функциональный вход этого оператора подаётся некоторый элемент a. Свойства этого элемента описаны в формуле, приписанной к его выходной дуге. Всего к дуге приписано две формулы, значит, данный ИГР является компактной формой записи двух графов с разметкой. По одной из формул элемент a имеет целый тип, и его значение равно единице, по другой — двум. Значит, на рассматриваемый оператор интерпретации поступает функция выбора элемента из списка. Для каждого из ИГР значение селектора своё, поэтому, если применить эквивалентное преобразование с перераспределением связей в графе, то результирующие графы также будут различными. И, значит, их нельзя представить в компактной форме, а можно записать

только раздельно. Получается, что исходный граф разделяется или расщепляется на два.

При перенаправлении связей часть (недостижимых) ветвей в графе удаляется (в результате граф упрощается), однако, при этом необходимо сохранить все формулы, которые характеризуют выбор данного пути выполнения программы, приведшего к соответствующему значению селектора. Для этого предусловие нового графа формируется путём конъюнкции всех формул, приписанных к дугам на пути от элемента a к аргументу arg (то есть проводится свёртка). В данной схеме это формула F и соответствующая формула дуги a.

Пример. Рассмотрим доказательство корректности функции `abs`, находящей абсолютное значение целого числа. Код функции *abs* на языке Пифагор:

```
abs << funcdef arg{
    ({arg:-},arg):[((arg,0):[<,>=]):?]:. >> return
}
```

Для данной функции заданы следующие пред- и постусловие соответственно:

$P \equiv (arg \in \text{int})$,

$Q \equiv (r \in \text{int}) \wedge ((r = arg) \wedge (arg \geqslant 0)) \vee ((r = -arg) \wedge (arg < 0))$,

где r обозначает результат вычислений.

На рисунке 2.11а приведена исходная тройка Хоара для функции `abs`. Для удобства некоторые дуги графа пронумерованы, номер дуги также является номером узла, из которого выходит данная дуга. Узел номер 1 размечен предусловием, узлы 2–8 являются константами и размечаются автоматически. Также автоматически размечаются выходные дуги 9–11 операторов группировки в списки. После этого к разметке готов оператор интерпретации 12. На его функциональный вход поступает параллельный список, значит, граф должен быть преобразован по правилу эквивалентного преобразования 3.2 (а потом 4.1) из таблицы 2.1. В результате получается ИГР, приведённый на рисунке 2.11б. Узлы 11 и 12 готовы к разметке. Рассмотрим разметку дуги 11. Функция «<» имеет шесть аксиом (аксиомы 12.1–12.6 в приложении Д). Из этих аксиом применима только одна (12.1), так как только для неё выполняется условие (2.4) (то есть отрицание условия (2.4) не является тождественной истиной):

$$((arg \in \text{int}) \wedge (a = (arg : \text{int}, 0 : \text{int})) \Rightarrow (a = (arg : T, 0 : T) \wedge T = (\text{int} \mid \text{float})).$$

Тогда по правилу прямого прослеживания (2.1) к дуге 11 приписывается формула:

$$f_1 \equiv \big(a = (arg : T, 0 : T)\big) \wedge \big(T = (\text{int} \mid \text{float})\big) \wedge (a_1 \in \text{bool}) \wedge (a_1 = (arg < 0)).$$

Аналогично размечается дуга 12 формулой

$$f_2 \equiv \big(a = (arg : T, 0 : T)\big) \wedge \big(T = (\text{int} \mid \text{float})\big) \wedge (a_2 \in \text{bool}) \wedge (a_2 = (arg \geqslant 0)).$$

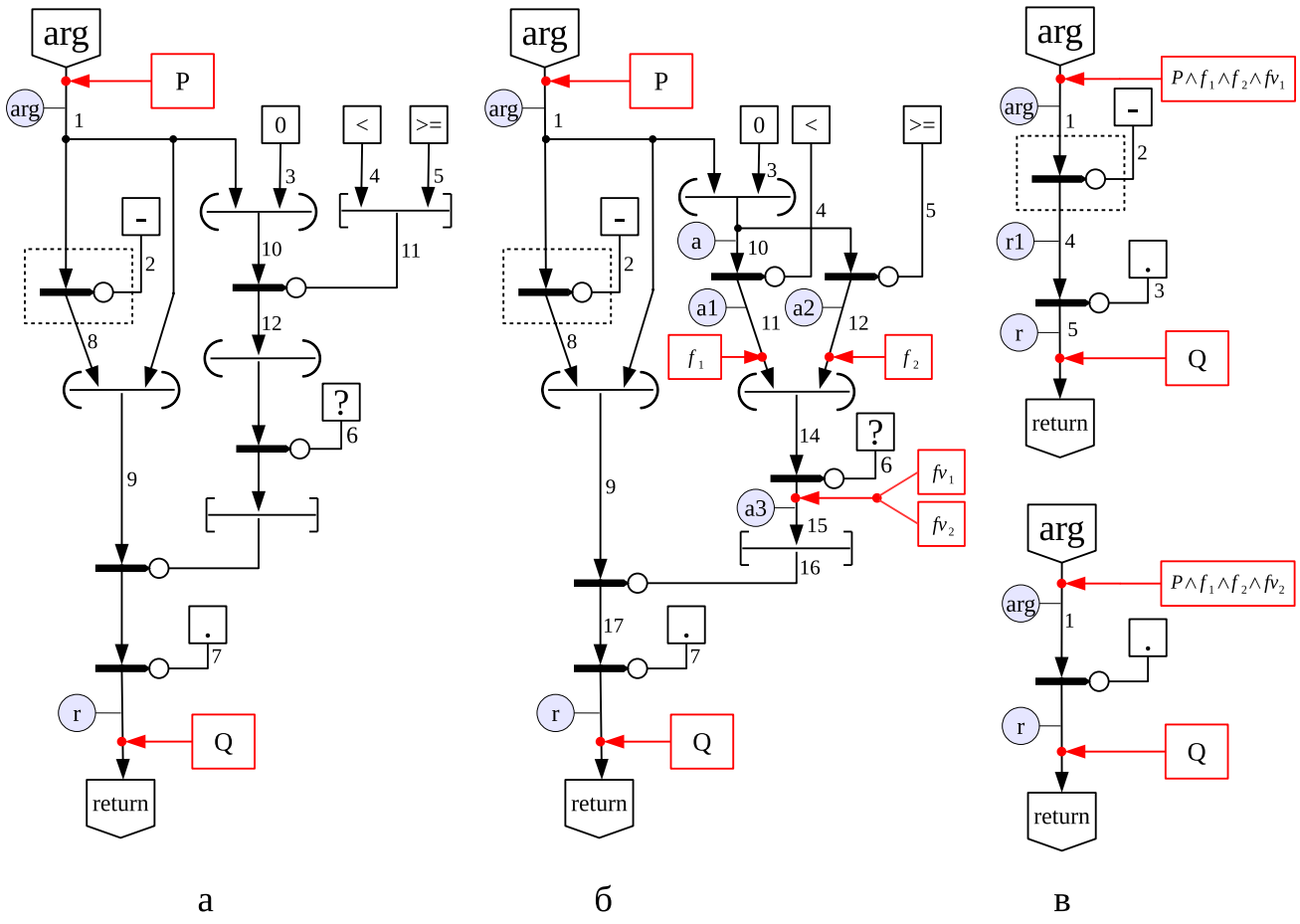


Рисунок 2.11 — Разметка формулами дуг графа функции `abs`

Дуга 14 размечается автоматически, после этого к разметке готов оператор интерпретации 15, принимающий в качестве аргумента список данных $(a_1, a_2) \equiv (arg < 0, arg \geqslant 0)$ и функцию «?». Из трёх аксиом функции «?» применима только первая (см. приложение Д, аксиомы 16.1–16.3). Однако данная аксиома содержит сложную рекурсивную формулу, поэтому, чтобы не усложнять доказательство, целесообразно сразу провести её упрощение. Для этого используем информацию из формул, описывающих свойства списка (a_1, a_2), выделим все варианты его возможных значений и проведём разделение формул (см. раздел 2.4.1). Условия a_1 и a_2 являются взаимоисключающими, поэтому множество возможных значений следующее: $\{(\text{true}, \text{false}), (\text{false}, \text{true})\}$. Определим, какое значение вернёт функция f из аксиомы 16.1 (приложение Д) для каждого из этих значений:

$$f((\text{bool}, \text{bool}), (\text{true}, \text{false}), 1) \equiv \text{phcons}(1, f((\text{bool}), (\text{false}), 2)) \equiv \text{phcons}(1, [\,]) \equiv [1],$$

$$f((\text{bool}, \text{bool}), (\text{false}, \text{true}), 1) \equiv f((\text{bool}), (\text{true}), 2) \equiv [2].$$

Тогда, применяя правило прямого прослеживания для функции «?» (и упрощение), получаем следующие формулы, приписываемые к дуге 15:

$$fv_1 \equiv ((a_1, a_2) = (\text{true}, \text{false})) \wedge (a_3 = [1]),$$

$fv_2 \equiv ((a_1, a_2) = (\text{false}, \text{true})) \wedge (a_3 = [2])$.

Дуга 16 размечается автоматически (при этом используется эквивалентное преобразование 1 из таблицы 2.1.2), и готовым к разметке становится оператор интерпретации 17.

Согласно формулам fv_1 и fv_2, на его функциональный вход поступают целочисленные константы, поэтому проводится расщепление графа на два новых, которые приведены на рисунке 2.11в.

В первом полученном графе к разметке готов оператор интерпретации 5. Ему в качестве аргумента передаётся задержанный список, с которого снимается задержка по правилу 2.1 таблицы 2.1. Также удаляется разметка с дуги 4, так как после снятия задержки это больше не выходная дуга константы. Оператор интерпретации 4 готов к разметке. Из аксиом функции «-» (аксиомы 6.1–6.9 приложения Д) условию применимости (2.4) удовлетворяет только одна аксиома (6.2). Тогда к дуге 4 приписывается формула $(arg \in \text{int}) \wedge$ $\wedge (r_1 \in \text{int}) \wedge (r_1 = (-arg))$. Далее размечается выходная дуга оператора интерпретации 5 формулой $r = r_1$, так как сигнал, используемый в качестве функции, не меняет своего аргумента (аксиома 1 из приложения Д).

Первый граф полностью размечен, и можно применить операцию свёртки и правило преобразования тройки в формулу (2.2). Получаем следующую формулу:

$$P \wedge f_1 \wedge f_2 \wedge fv_1 \wedge ((arg \in \text{int}) \wedge (r_1 \in \text{int}) \wedge (r_1 = (-arg))) \wedge (r = r_1) \Rightarrow Q \equiv$$
$$\equiv (arg, r \in \text{int}) \wedge (arg < 0) \wedge (r = (-arg)) \Rightarrow Q.$$

Формула является тождественно истинной.

Второй граф, полученный в результате расщепления, имеет только одну неразмеченную дугу, которая размечается формулой $arg = r_1$, так как в качестве функции используется сигнал. Полностью размеченный ИГР с помощью свёртки преобразуется в тройку с пустой программой, а далее в формулу:

$$P \wedge f_1 \wedge f_2 \wedge fv_2 \wedge (r = arg) \Rightarrow Q \equiv (arg, r \in \text{int}) \wedge (arg \geqslant 0) \wedge (r = arg) \Rightarrow Q,$$

которая также тождественно истинна. Значит, программа `abs` корректна.

В результате всех рассмотренных преобразований ИГР ФПП программы можно сделать следующий вывод: весь процесс доказательства можно представить в виде дерева, корень которого — исходный ИГР, дочерние узлы получаются из родительских выполнением одного из преобразований графа (разметки дуги, свёртки или изменения графа), а листья — полностью размеченные ИГР, над которыми проводится полная свёртка и преобразование в формулы. Будем называть такое дерево *деревом доказательства* или просто доказательством корректности программы.

Выводы

В главе построена аксиоматическая теория, позволяющая доказывать корректность ФПП программ.

1. Формализована семантика языка программирования Пифагор.

2. Построена аксиоматическая система языка спецификации на базе исчисления конструкций, позволяющая описывать свойства программ на языке Пифагор. В частности, язык спецификации позволяет формулировать утверждения о списках произвольной длины и с элементами произвольного типа.

3. Построена аксиоматическая теория на базе исчисления Хоара, позволяющая доказывать корректность программ на языке Пифагор. Корректность программы выводится из истинности её тройки Хоара, которая преобразуется в формулы на языке спецификации.

4. Показано, что доказательство корректности ФПП программ удобно проводить на информационном графе программы, в этом случае доказательство представимо в виде дерева. Описаны основные преобразования, которые претерпевает ИГР при доказательстве корректности программы.

3 Формальная верификация ФПП программ

В главе 3 разрабатывается метод доказательства корректности рекурсивных ФПП программ с использованием построенной аксиоматической теории для языка Пифагор. Основная особенность ФПП программ состоит в отсутствии операторов цикла, а все повторяющиеся действия реализуются с помощью рекурсии. Наличие рекурсии привносит свои особенности в процесс доказательства корректности ФПП программ с помощью рассмотренных методов. Доказательство корректности рекурсивных функций разделяется на два этапа: доказательство частичной корректности и доказательство завершения программы. Приводится алгоритм преобразования произвольной рекурсивной функции в прямую рекурсию, для сведения задачи к доказательству корректности одной функции.

3.1 Доказательство корректности рекурсивных функций

Для того, чтобы рекурсивная функция была корректной, она должна, во-первых, завершаться, во-вторых, возвращать правильный результат. Проанализируем особенности рекурсивной функции. Если в программе присутствует рекурсия, то один и тот же код вызывается несколько раз, а различия будут только в значениях аргументов. Тогда необходимым условием завершения рекурсии является то, что аргумент пробегает неповторяющуюся последовательность значений. В корректной рекурсивной функции обязательно должна присутствовать «точка ветвления», в которой происходит выбор между возможными путями выполнения программы. Одна часть путей приводит к завершению работы и возвращению результата вычислений, другая — к рекурсивному вызову функции. По какому пути пойдут дальнейшие вычисления, определяется значением некоторой функции от входных аргументов. При этом значения выражений, приводящих к рекурсивному вызову функции или завершению рекурсии, являются взаимоисключающими, то есть

$$\neg(\mathit{условие\_рекурсивного\_вызова} = \text{true}) \Leftrightarrow (\mathit{условие\_завершения} = \text{true}).$$

Используем следующие термины. *Тривиальная (базовая) ветвь* алгоритма — путь выполнения функции, который обеспечивает выход из рекурсии, и *рекурсивная ветвь* алгоритма — путь, на котором происходит рекурсивный вызов функции. Пусть $f(x)$ — рекурсивная функция. Если функция f получает в качестве аргумента x некоторое значение x_0, то это значение назовём аргументом текущего вызова функции f или *текущим аргументом*. Если этот аргумент приводит к выполнению рекурсивной ветви алгоритма, то произойдёт рекурсивный вызов функции f с некоторым аргументом $x_1 = g(x_0)$, где g — некоторая функция. Назовём x_1 аргументом рекурсивного вызова или *рекурсивным аргументом*. Очевидно, что таких аргументов может быть несколько.

При проверке корректности рекурсивной функции доказательство разделяется на две

части: доказательство частичной корректности и доказательство завершения программы. При этом считаем, что проверяемая функция является прямой рекурсией, а корректность всех вызываемых из неё функций уже доказана. Преобразование взаимной рекурсии в прямую рассматривается в разделе 3.2.

3.1.1 Доказательство частичной корректности рекурсивной функции

При доказательстве частичной корректности предполагается, что программа завершается (см. раздел 3.1.2). Доказательство проводится методом индукции. Базу индукции даёт доказательство корректности всех тривиальных ветвей рекурсии. Предположением индукции является корректность доказываемой тройки для всех рекурсивных аргументов.

Доказательство проводится по алгоритму, описанному в разделе 2.3. При этом рекурсивный вызов рассматривается так же, как вызовы других функций, а в качестве теоремы используется доказываемая тройка Хоара, у которой текущий аргумент заменяется на аргумент рекурсивного вызова, а идентификатор результата — на идентификатор выходной дуги рекурсивного вызова.

База индукции доказывается при разметке графа формулами. Имеющаяся «точка ветвления» программы приводит к расщеплению графа, и часть полученных ИГР будет соответствовать варианту выполнения программы без рекурсивных вызовов. Корректность этих ИГР и есть база индукции. Условие завершения функции требуется для того, чтобы показать, что индукция имеет место, и значение ограничивающей функции для каждого рекурсивного аргумента меньше, чем для текущего.

Если у рекурсивной функции предполагается наличие нескольких теорем, по аналогии с несколькими аксиомами для встроенных функций, то корректность каждой из них должна доказываться независимо от других: при разметке выходной дуги рекурсивного вызова нельзя использовать другие тройки, кроме доказываемой. В противном случае доказательство троек надо проводить одновременно, но такая ситуация в работе не рассматривается, поскольку несколько троек для одной функции могут быть объединены в одну. Когда доказываемая тройка одна, необходимо показать, что условие применимости (2.4) теоремы для рекурсивного аргумента является тождественно истинным, тогда как для проверки применимости других вызываемых функций достаточно показать выполнимость условия (2.4).

Пример. Рассмотрим доказательство частичной корректности рекурсии на примере функции `order`, которая находит число десятичных знаков у натурального числа n [1]. В качестве входных аргументов функция принимает список из трёх целых чисел (n, p, k), где n — число, для которого надо найти количество десятичных знаков, p и k — вспомогательные величины, удовлетворяющие соотношению $p = 10^k$. Функция находит минимальную степень

k_0, в которую надо возвести число 10, чтобы число $p = 10^{k_0}$ превысило n. Вычисления всегда можно начать с минимально допустимых значений $p = 1$ и $k = 0$. Вначале n сравнивается с p, и если $n > p$, то возвращается k, иначе происходит рекурсивный вызов функции с аргументами $(n, p \cdot 10, k + 1)$. Ниже приведён исходный код программы на языке Пифагор:

```
order << funcdef x{
    n<<x:1;   p<<x:2;   k<<x:3;  f1<<(n,p):<;
    if<<(f1,f1:-):?;
    act<<( k, {(n,(p,10):*,(k,1):+):order} );
    return << act:if:.;
}
```

Зададим предусловие P и постусловие Q для функции `order`. Ограничение на входные аргументы: n — целое положительное число, p и k — любые положительные числа, связанные соотношением $p = 10^k$, и p обязательно меньше или равно n, либо больше n, но минимально возможное. Результат res такой, что 10^{res-1} ещё меньше или равно n, а 10^{res} уже больше n. Эти ограничения можно выразить следующими формулами:

$$P \equiv (n, p, k \in \text{int}) \wedge (n, p, k > 0) \wedge (p = 10^k) \wedge \left((n \geqslant 10^k) \vee ((n < 10^k) \wedge (n \geqslant 10^{k-1}))\right),$$

$$Q \equiv (res \in \text{int}) \wedge (n \geqslant 10^{res-1}) \wedge (n < 10^{res}).$$

Покажем, что функция частично корректна. Исходный информационный граф программы приведён на рисунке 3.1а. На приведённом графе уже использовано правило эквивалентного преобразования входного списка x с выделением трёх элементов n, p и k. Задержанный список обозначен пунктиром. До снятия задержки он считается константой, поэтому изображён в виде текста, а не подграфа. Идентификатор дуги является идентификатором результата, который возвращает оператор, из которого выходит дуга. Далее в тексте пишем «оператор t» вместо «оператор, из которого выходит дуга t».

Изначально дуги графа не размечены, к входной и выходной дуге приписывается предусловие P и постусловие Q соответственно. Все выходные дуги констант сразу считаются размеченными, а выходные дуги списков данных размечаются автоматически, как только разметку получают все их входные дуги.

На основе аксиом для встроенной функции «<» можно разметить дугу f_1, а после этого аксиомы функции «-» позволяют разметить дугу f_2 (формулы разметки приведены на рисунке 3.1а).

После этого все входные дуги оператора if становятся размеченными. Оператор if применяет функцию «?» к списку (f_1, f_2). Эта функция вычисляет номера позиций истинных булевских констант в булевском списке и используется для организации выборочного продолжения вычислений. Если элементы списка является взаимоисключающим (при истинности одного условия все остальные обязательно ложные), то функция возвращает одну

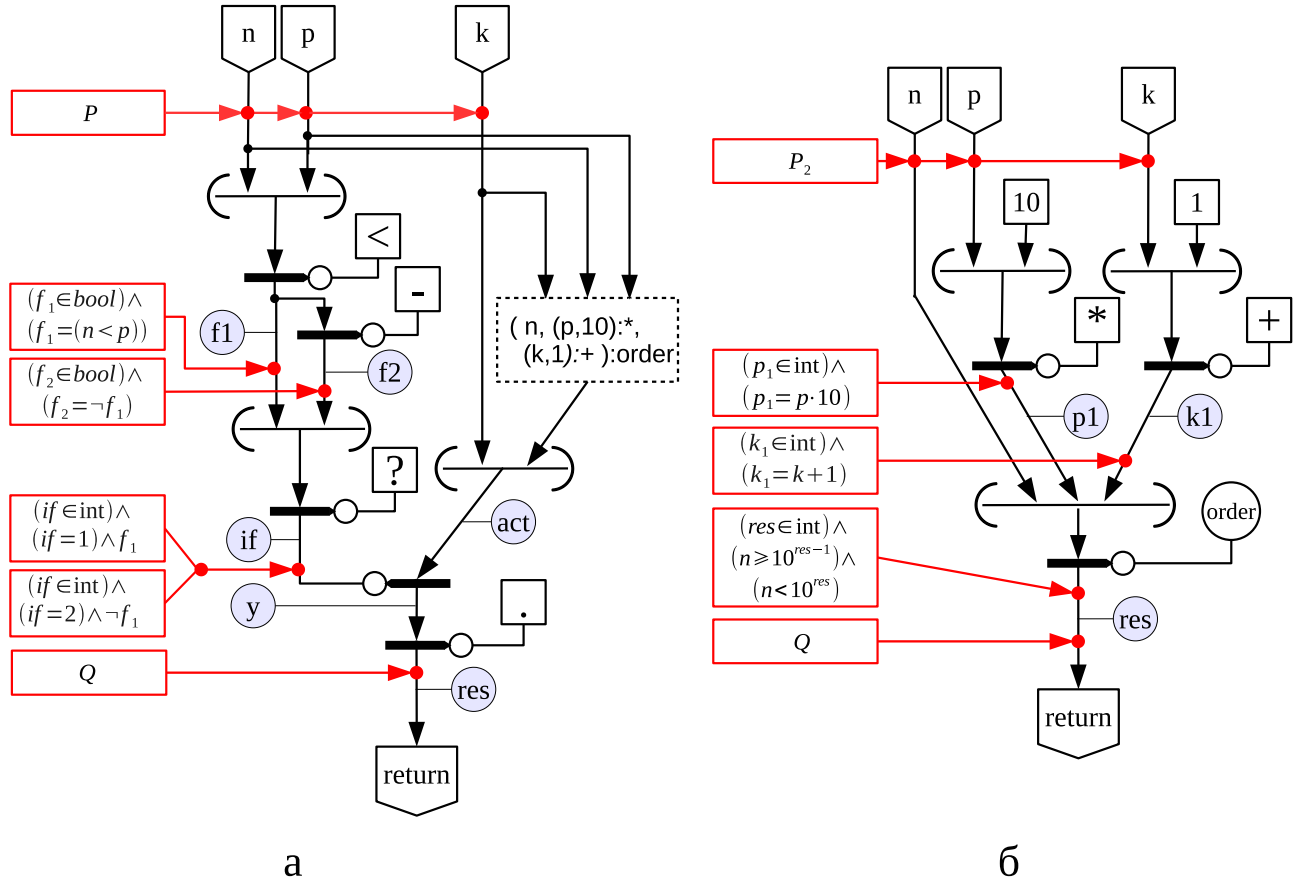


Рисунок 3.1 — Информационный граф с разметкой для функции `order`; а — исходный информационный граф функции с частичной разметкой, задержанный список представлен как константа; б — полностью размеченный информационный граф, содержащий рекурсивный вызов функции

целочисленную константу, соответствующую позиции истинного условия.

В данном случае длина входного списка известна, и условия f_1 и f_2 взаимоисключающие, поэтому, используя разделение формул и аксиомы функции «?», получаем две формулы, которые припишутся к дуге if:

$$(f_1, f_2 \in \text{bool}) \wedge (f_1 = \text{true}) \wedge (f_2 = \text{false}) \wedge (if \in \text{int}) \wedge (if = 1),$$

$$(f_1, f_2 \in \text{bool}) \wedge (f_1 = \text{false}) \wedge (f_2 = \text{true}) \wedge (if \in \text{int}) \wedge (if = 2).$$

После разметки дуги if (см. рисунок 3.1а) становятся размечены все входные дуги оператора y. Согласно разметке дуги if, на функциональный вход оператору y подаётся целочисленная константа, принимающая значение 1 или 2, а в качестве аргумента — список данных arg длины два. Поэтому константа if на функциональном входе будет интерпретироваться как функция выбора элемента из списка. Применение оператора y приведёт к расщеплению ИГР на два графа G_1 и G_2. В G_1 попадёт первый элемент списка arg, а в G_2 — второй. Эти ИГР будут иметь вид:

$$G_1 \equiv \boxed{P_1}\ \texttt{k:.}\rightarrow res\ \boxed{Q}\ ,$$

$$G_2 \equiv \boxed{P_2}\ \texttt{\{(n,(p,10):*,(k,1):+):order\}:.}\rightarrow res\ \boxed{Q}\ ,$$

где $P_1 \equiv P \wedge (n < p)$, а $P_2 \equiv P \wedge \neg(n < p)$.

Граф G_1 может быть полностью размечен и преобразован в формулу $P_1 \wedge (k = res) \Rightarrow$ $\Rightarrow Q$, которая является тождественно истинной. В графе G_2 происходит раскрытие задержанного списка, в результате чего получается граф, приведённый на рисунке 3.1б.

Изначально у графа на рисунке 3.1б размечена только входная и выходная дуга предусловием P_2 и постусловием Q соответственно. На основе аксиом для функций «`*`» и «`+`» размечаются дуги p_1 и k_1. После этого к разметке готов оператор res, который осуществляет рекурсивный вызов функции `order`.

Для доказательства частичной корректности рекурсивной функции, во-первых, требуется показать, что аргумент рекурсивного вызова удовлетворяет предусловию функции `order`, то есть является допустимым. И, во-вторых, предположив, что рекурсивный вызов функции возвращает результат, удовлетворяющий постусловию, показать, что текущий вызов функции вернёт верный результат. Это эквивалентно доказательству истинности информационного графа с разметкой на рисунке 3.1б.

Рекурсивный аргумент является допустимым, так как из ограничений

$$P_r \equiv (P_2 \wedge (p_1, k_1 \in \text{int}) \wedge (p_1 = p \cdot 10) \wedge (k_1 = k + 1))$$

следует предусловие рекурсивного вызова $P_{rec} \equiv P(n, p_1, k_1)$, которое получается из предусловия P заменой переменной p на p_1, а k на k_1:

$$\begin{aligned} P_r \Rightarrow P_{rec} \equiv (P_2 \wedge (p_1, k_1 \in \text{int}) \wedge (p_1 = p \cdot 10) \wedge (k_1 = k + 1)) \Rightarrow \\ \Rightarrow \big((n, p_1, k_1 \in \text{int}) \wedge (n, p_1, k_1 > 0) \wedge (p = 10^{k_1}) \wedge \\ \wedge \big((n \geqslant 10^{k_1}) \vee ((n < 10^{k_1}) \wedge (n \geqslant 10^{k_1 - 1}))\big)\big)\,. \end{aligned}$$

Эта формула является истинной, так как из посылки P_r имеем $p_1 = 10 \cdot p = 10 \cdot 10^k = 10^{k+1} =$ $= 10^{k_1}$ и $n \geqslant 10^k$, значит, $n \geqslant 10^{k_1}$ или $(n < 10^{k_1}) \wedge (n \geqslant 10^{k_1 - 1})$.

На основе предположения о корректности рекурсивного вызова, получаем тройку Хоара рекурсивного вызова функции *order*:

$$\boxed{P_r}\ \texttt{(n,p1,k1):order}\rightarrow res\ \boxed{(res \in \text{int}) \wedge (n \geqslant 10^{res-1}) \wedge (n < 10^{res})}\ ,$$

которая позволяет разметить дугу res графа G_2, как показано на рисунке 3.1б.

Теперь граф G_2 полностью размечен и может быть преобразован в формулу:

$$\begin{aligned} (P_2 \wedge (p_1, k_1 \in \text{int}) \wedge (p_1 = p \cdot 10) \wedge (k_1 = k + 1) \wedge \\ \wedge (res \in \text{int}) \wedge (n \geqslant 10^{res-1}) \wedge (n < 10^{res})\big) \Rightarrow Q, \end{aligned} \quad (3.1)$$

которая является тождественной истинной. А значит, исходная функция `order` является частично корректной для предусловия P и постусловия Q.

3.1.2 Доказательство завершения рекурсии

При доказательстве завершения программы используется трансфинитная индукция (индукция на фундированном множестве). Задаётся фундированное множество S с отношением порядка $\prec$ и ограничивающая функция φ (см. раздел 1.2.5).

В общем случае доказательство завершения функции `Prog`, для которой доказана частичная корректность, то есть истинна тройка $\boxed{P(x)}$ `Prog(x)` $\boxed{Q}$, состоит в следующем [176, 180]: необходимо выбрать множество S, отношение $\prec$ на S и ограничивающую функцию φ, удовлетворяющие следующим свойствам:

1. $S \equiv \{x : T \mid R(x)\}$, где $T : \mathrm{Set}$ — некоторое множество, $R(x) : T \to \mathrm{bool}$ — предикат, определяющий непустое подмножество множества T; если подмножество непустое, то выполнено условие $(\exists x : T.\, R(x)) = \mathrm{true}$;

2. отношение $\prec$ является отношением порядка, то есть удовлетворяет свойствам:

   1. антирефлексивность $\forall a : T.\, \neg(a \prec a)$,

   2. антисимметричность $\forall (a, b : T).\, (a \prec b) \Rightarrow \neg(a \prec b)$,

   3. транзитивность $\forall (a, b, c : T).\, ((a \prec b) \wedge (b \prec c)) \Rightarrow (a \prec c)$;

3. множество S фундированное относительно порядка $\prec$, то есть любое его непустое подмножество $M \subseteq S$ имеет минимальный элемент; это эквивалентно следующему утверждению:

$$\forall f : T \to \mathrm{bool}.\, \exists x : T.\, \big((R(x) \wedge f(x)) = \mathrm{true}\big) \wedge \exists (m \in M).\, \forall (x \in M).\, (m \prec x) \vee (x = m),$$

где $M \equiv \{x : T \mid R(x) \wedge f(x)\}$;

4. ограничивающая функция $\varphi : U \to T$ (где U — множество, которому принадлежит аргумент x рекурсивной функции `Prog`) принимает значения из фундированного множества, если её аргумент удовлетворяет предусловию $P(x)$ функции `Prog`:

$$\forall x : U.\, P(x) \Rightarrow (\varphi(x) \in S);$$

5. любой допустимый текущий и рекурсивный аргумент будет принадлежать области определения ограничивающей функции;

6. значение ограничивающей функции для текущего аргумента будет больше, чем для каждого рекурсивного.

Так как область определения ограничивающей функции содержит аргументы, удовлетворяющие предусловию доказываемой функции, получаем, что текущий аргумент принадлежит области определения ограничивающей функции (по определению), а принадлежность

рекурсивного аргумента следует из частичной корректности функции (соответствует выполнению условия применимости (2.4) для рекурсивного аргумента). Поэтому, если доказана частичная корректность функции, то для доказательства завершения достаточно показать, что значение ограничивающей функции для текущего аргумента будет больше каждого рекурсивного. Это требование можно выразить следующей формулой:

$$\bigwedge_i \left(\varphi(x) < \varphi(\xi_i(x))\right),$$

где $\varphi(x)$ — ограничивающая функция, $\bigwedge$ — конъюнкция по всем рекурсивным вызовам (которые пронумерованы), присутствующих в теле рассматриваемой функции `Prog`, $\xi_i(x)$ — функция, выражающая i-ый рекурсивный аргумент через текущий.

Аргументы ограничивающей функции совпадают с аргументами рекурсивной функции, поэтому условия завершения функции могут быть добавлены в предусловие и постусловие рекурсивной функции, и тогда доказательство истинности тройки Хоара для рекурсивной функции станет доказательством тотальной корректности: программа завершается для всех допускаемых предусловием входных аргументов и возвращает верный результат, удовлетворяющий постусловию. Если фундированное множество S с отношением порядка $\prec$ и ограничивающая функция φ введены корректно, то для рекурсивной функции `Prog` с условием частичной корректности $\boxed{P(x)}$ `Prog(x)` $\boxed{Q}$, условие тотальной корректности будет следующим:

$$\boxed{P(x)} \texttt{ Prog(x) } \boxed{Q \bigwedge_i \left(\varphi(x) < \varphi(x_i)\right)}, \tag{3.2}$$

где x_i — идентификатор i-го рекурсивного аргумента, также являющийся идентификатором входной дуги оператора интерпретации, выполняющего рекурсивный вызов функции.

Пример. Рассмотрим доказательство завершения рекурсии на примере функции `order`, частичная корректность которой доказана в разделе 3.1.1. Докажем, что программа `order` завершается для всех значений аргументов, удовлетворяющих предусловию.

1. В качестве фундированного множества возьмём множество $S(n)$, где n — целое положительное число (первый элемент списка, принимаемого функцией `order`):

$$S(n) \equiv \{x : \mathbb{R} \mid \exists k : \mathbb{Z}.\, (k > 0) \wedge (x = \frac{n}{10^k}) \wedge (n > 10^{k-1})\}.$$

2. Данное множество является подмножеством действительных чисел, поэтому отношение «меньше» $<$ на его элементах будет отношением порядка, удовлетворяющим всем требованиям.

В зависимости от значения n множество $S(n)$ будет состоять из следующих элементов: $\left\{\frac{n}{10^1}, \frac{n}{10^2}, \ldots, \frac{n}{10^{k_0}}\right\}$, при условии $10^{k_0-1} < n < 10^{k_0}$. Например, при $n = 4503$

$$S(4503) \equiv \left\{ \frac{4503}{10}, \frac{4503}{10^2}, \frac{4503}{10^3}, \frac{4503}{10^4} \right\} \equiv \{450.3, 45.03, 4.503, 0.4503\},\ (10^3 < 4503 < 10^4).$$

3. $S(n)$ — конечное множество, отношение $<$ является отношением линейного порядка, поэтому у любого подмножества $M(n) \subset S(n)$ будет минимальный элемент.

4. Введём ограничивающую функцию $\varphi(n, p, k) = \dfrac{n}{10^k}$. Если входной аргумент (n, p, k) удовлетворяет предусловию P функции `order`, то соотношение $n > 10^{k-1}$ будет выполнено, отсюда значения ограничивающей функции будут принадлежать множеству $S(n)$. Это утверждение эквивалентно доказательству истинности формулы:

$$P \Rightarrow ((n, k \in \text{int}) \wedge (n, k > 0) \wedge (n > 10^{k-1}).$$

Очевидно, что данная формула истинна так как $n \geqslant 10^k > 10^{k-1}$.

5. В силу доказанной ранее частичной корректности, все возможные аргументы принадлежат области определения функции φ.

Осталось показать, что значение функции φ от текущего аргумента больше значения функции от любого рекурсивного аргумента. В функции `order` присутствует один рекурсивный вызов функции с рекурсивным аргументом (n, p_1, k_1), где p_1 и k_1 — идентификаторы дуг ИГР (см. рисунок 3.1б). После разметки графа к дугам приписаны формулы, характеризующие свойства p_1 и k_1, откуда следует, что $p_1 = p \cdot 10$, а $k_1 = k + 1$. Получаем:

$$\varphi(n, p, k) = \frac{n}{10^k} > \varphi(n, p \cdot 10, k + 1) = \frac{n}{10^{k+1}}.$$

Таким образом, завершение рекурсивной функции `order` доказано.

Значит, функция является тотально корректной. Это утверждение можно выразить с помощью следующей тройки:

$$\boxed{P(x)}\ \texttt{(n,p,k):order} \rightarrow r\ \boxed{Q \wedge \big(\varphi(n, p, k) < \varphi(n, p \cdot 10, k + 1)\big)}.$$

В приложении Е приведено два дополнительных примера доказательства корректности рекурсивной функции: первая функция вычисляет частное и остаток от целочисленного деления, вторая находит факториал неотрицательного целого числа [176, 178, 181].

В приложении Ж рассматривается доказательство завершения программ с явно выраженным рекуррентным соотношением [152]. Это частный случай рекурсии, который позволяет упростить доказательство корректности программы, так как не требует нахождения ограничивающей функции. Доказательство состоит в установлении эквивалентности некоторой функции на языке Пифагор рекуррентному соотношению [182].

3.2 Верификация программ со взаимной рекурсией

Рассмотренный метод доказательства корректности напрямую не работает в случае взаимной рекурсии нескольких функций (например, функция f вызывает функцию g, а функция g вызывает функцию f). Существует два основных способа решения данной проблемы [154]. Первый способ — одновременное доказательство корректности всех функций во взаимной рекурсии, которое обычно требует проведения доказательства с помощью одновременной индукции (simultaneous induction) [155]. Второй способ — удаление взаимной рекурсии посредством преобразования программы, в результате которого получается одна рекурсивная функция [153].

В сущности, это два одинаковых метода, а мы выбираем только внешнее представление доказательства для пользователя. Недостаток первого способа — необходимость доказывать несколько утверждений одновременно, а недостаток второго способа — результирующая функция может получиться достаточно сложной.

В данной работе рассматривается второй способ решения проблемы взаимной рекурсии для программ на языке Пифагор [183]. Метод преобразования программы, устраняющий взаимную рекурсию, выбран потому, что он более удобный в применении к доказательству корректности программ на языке Пифагор, при котором происходит разметка дуг графа функции формулами. Доказательство более наглядное, пользователю не требуется удерживать в памяти несколько одновременно доказываемых утверждений и «переключаться» между несколькими графами доказываемых функций. Кроме того, если присутствует взаимная рекурсия, то при доказательстве завершения программы, свой порядок (своё фундированное множество и ограничивающая функция) необходим для аргументов каждой из функций. Если провести преобразование и устранить взаимную рекурсию, то достаточно построить один порядок на множестве аргументов результирующей функции. Ещё одно преимущество состоит в том, что процесс преобразования кода при устранении взаимной рекурсии может проводиться полностью автоматически.

Виды рекурсий. Рекурсия может быть прямой или косвенной. *Прямая рекурсия* — это рекурсивная функция, которая может вызывать себя только непосредственно из своего тела. *Косвенная рекурсия* — это рекурсивная функция, в теле которой нет вызовов самой себя, но эта функция вызывается вновь через цепочку вызовов других функций [152]. Введём понятие *смешанной рекурсии*, как функции в которой присутствует прямой и косвенный рекурсивный вызов.

Также используем термины прямая и косвенная связь для разных функций. Функция A *прямо связанна* с функцией B, если B вызывается непосредственно в теле A. Функция

A *косвенно связана* с B, если вызов B опосредован вызовами других функций. A и B — *связанные* функции, если они связаны прямо или косвенно. Если функция A не связана с функцией B ни прямо, ни косвенно, то назовём A *независимой* от B функцией.

Пусть A — рассматриваемая рекурсивная функция. Обозначим через $\Omega(A)$ *множество всех связанных функций* — функций, которые вызываются из A, как непосредственно в её теле, так и через цепочку вызовов других функций. Саму функцию A также отнесём к $\Omega(A)$. Для каждой функции в $\Omega(A)$ укажем её имя, а после него в круглых скобках укажем список вызываемых непосредственно в её теле функций. Например, запись вида $\Omega(A) = \{A(B,D), B(A,B,C), C(A), D()\}$ означает, что рассматриваемая функция A является косвенной рекурсией, в её теле присутствует вызов функции B и D; B — смешанная рекурсия, вызывающая себя и функции A, C; C — косвенная рекурсия, вызывающая функцию A; D — нерекурсивная функция, независимая от других функций. Множество связанных функций $\Omega(A)$ можно изобразить в виде *дерева возможных вызовов*. Это дерево, корень которого помечен именем рассматриваемой функции A; с каждым узлом, помеченным именем функции X, связаны дочерние узлы, соответствующие вызываемым в теле X функциям. Пример такого дерева для множества $\Omega(A) = \{A(B,D), B(A,B,C), C(A), D()\}$ приведён на рисунке 3.2а.

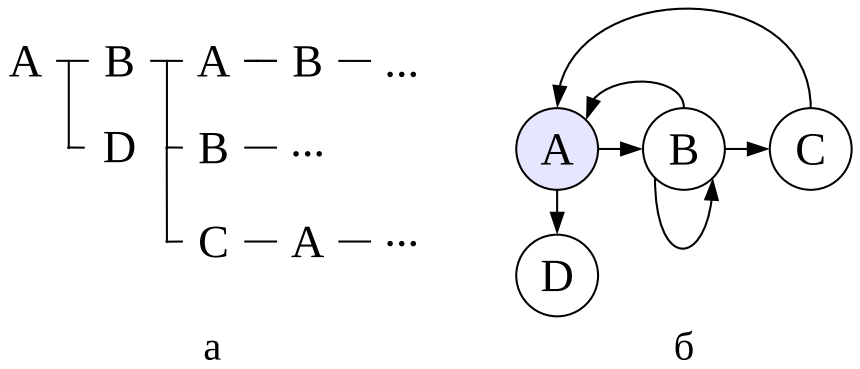


Рисунок 3.2 — Различные варианты представления множества связанных функций $\Omega(A) = \{A(B,D), B(A,B,C), C(A), D()\}$; а — дерево возможных вызовов функций; б — граф всех связанных функций, серым отмечена вершина с именем рассматриваемой функции

Бесконечное дерево возможных вызовов функций можно «свернуть» в *граф всех связанных функций*. Это ориентированный граф, вершины которого помечены именами функций $\Omega(A)$, а дуги направлены из вызывающей функции в вызываемую. Пример такого графа приведён на рисунке 3.2б.

Функции A и B *взаимно рекурсивные*, если функция A связана с B, и функция B связана с A. Обозначим отношение взаимной рекурсии $\leftrightarrow$. Если считать, что отношение

взаимной рекурсии рефлексивно ($A \leftrightarrow A$), то отношение $\leftrightarrow$ будет отношением эквивалентности, которое разделяет множество всех связанных функций на непересекающиеся классы. На графе всех связанных функций эти классы будут соответствовать компонентами сильной связности [153].

3.2.1 Универсальная рекурсивная функция

Одним из способов сведения произвольной рекурсии к прямой является построение универсальной рекурсивной функции [53]. Пусть $\mathfrak{F} = \{f_1, f_2, \ldots, f_k\}$ — множество функций $f_i : M \to M$, $i = 1, 2, \ldots, k$, M — произвольное множество. Каждой функции f_i приписано число $i \in \mathbb{N}$, называемое номером функции. Функция $F : \mathbb{N} \times M \to M$ называется *универсальной рекурсивной функцией* (УРФ) для множества $\mathfrak{F}$, если

$$F(n, x) = \begin{cases} f_1(x), \ n = 1; \\ \ldots \\ f_k(x), \ n = k. \end{cases}$$

Например, рассмотрим две связанные функции A и B. В общем случае, функция A является смешанной рекурсией, вызывает функцию B, которая тоже является смешанной рекурсией. Это случай взаимной рекурсии двух функций. Множество связанных рекурсивных функций $\Omega(A)$ имеет вид: $\{A(A, B), B(A, B)\}$. Дерево возможных вызовов функций приведено в левой части рисунка 3.3. Устраним взаимную рекурсию с помощью универсальной рекурсивной функции. Зададим универсальную рекурсивную функцию $F(n, x)$, у которой n — номер рекурсивной функции, объединяемой в УРФ, а x — аргумент для функции с номером n; n будет принимать значение «1», когда надо выполнить код функции A и «2», если требуется выполнить код функции B. Дерево возможных вызовов универсальной рекурсивной функции F приведено в правой части рисунка 3.3. Построение УРФ для большего числа функций аналогично.

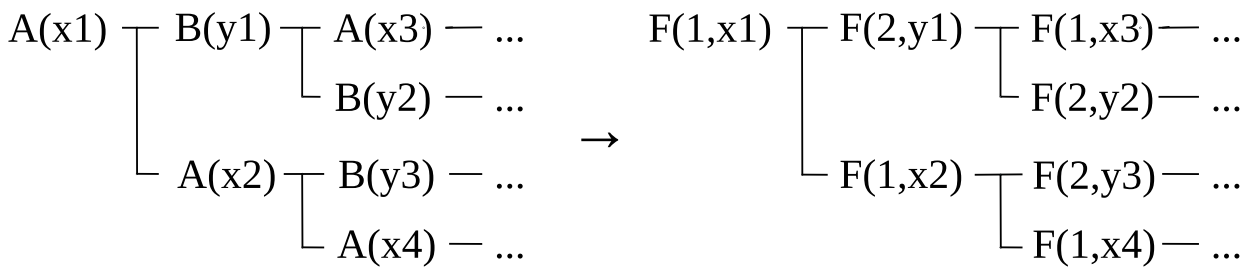


Рисунок 3.3 — Схема удаления взаимной рекурсии двух смешано рекурсивных функций A и B с помощью универсальной рекурсивной функции F; после имени каждой функции в скобках указаны её аргументы

3.2.2 Объединение функций

В простых случаях косвенную рекурсию можно свести к прямой рекурсии с помощью *объединения* тел связанных функций в одну функцию. Подобный метод используется в [153] для логических программ и называется развёрткой (unfolding). Поясним данный способ преобразования на примере двух функций. Пусть дана функция A, которая, в общем случае, является смешанной рекурсией. В ней присутствует вызов функции B, которая является косвенной рекурсией и прямо связанна с функцией A. То есть в функции B нет вызова самой себя, но присутствует вызов функции A. В этом случае косвенную рекурсию можно свести к прямой рекурсии объединением тел функций A и B в одну функцию AB. Таким образом, код функции B «встраивается» в код функции A в месте вызова B. Изменение графа двух связанных функций для описанного случая приведено на рисунке 3.4а. Исходное множество связанных функций $\Omega(A) = \{A(A,B), B(A)\}$ после объединения кода функций A и B в функцию AB преобразуется в $\Omega(AB) = \{AB(AB)\}$.

Если рассматриваемое множество связанных функций состоит из более чем двух функций, то объединение кода функций не приведёт к дублированию кода в том случае, если только одна функция связана с рассматриваемой косвенной рекурсией B. Это эквивалентно тому, что на графе всех связанных функций вершина B имеет только одну входную дугу. В остальных случаях косвенную рекурсию лучше устранять с помощью универсальной рекурсивной функции, описанной ранее.

Пример объединения функций для случая трёх связанных функций приведён на рисунке 3.4б, граф связанных функций для исходного $\Omega(A) = \{A(B), B(A,B,C), C(A)\}$. Функция C является косвенной рекурсией и имеет одну входную дугу на графе связанных функций. Поэтому её код может быть «встроен» в функцию B, узел которой является смежным с узлом C по его единственной входной дуге. Полученной в результате объединения функции присваивается имя BC, изменённое $\Omega(A) = \{A(BC), BC(A,BC)\}$. Для дальнейшего преобразования полученной рекурсии в прямую необходимо построить УРФ.

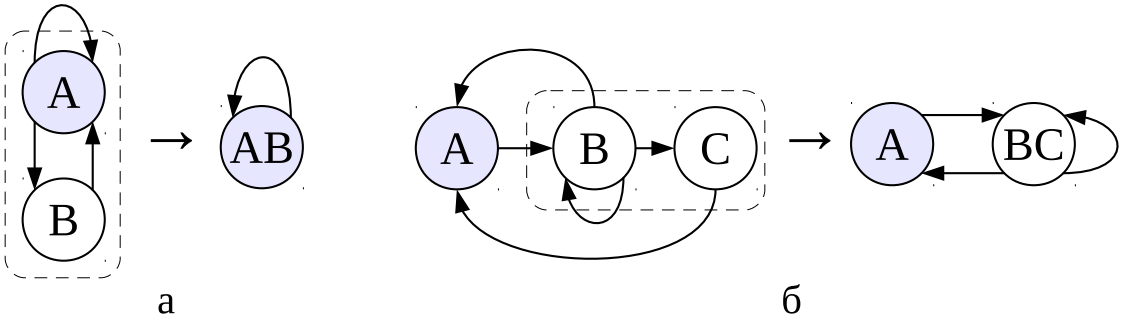


Рисунок 3.4 — Удаление косвенной рекурсии с помощью объединения кода функций

3.2.3 Алгоритм преобразования произвольной рекурсии в прямую

Резюмируя вышесказанное, можно предложить следующий алгоритм преобразования произвольной рекурсии в прямую.

Дана произвольная рекурсивная функция A со множеством связанных функций $\Omega(A)$. Требуется преобразовать её в прямую рекурсию.

1. На основе $\Omega(A)$ строим граф всех связанных функций $G(A)$.

2. В $\Omega(A)$ находим все нерекурсивные функции. В $G(A)$ таким функциям будут соответствовать вершины, не входящие ни в один цикл. То есть для вершины X, помеченной нерекурсивной функцией, не существует пути, который привёл бы опять в вершину X. Все найденные нерекурсивные функции являются независимыми от A и удаляются из множества $\Omega(A)$. В результате чего получаем новое множество связанных функций $\Omega_1(A)$ с графом связанных функций $G_1(A)$.

3. Удаляем из $\Omega_1(A)$ все рекурсивные функции, не связанные с A. Для этого в графе $G_1(A)$ находим все узлы, для которых любой цикл, проходящий через этот узел, не проходит через вершину A. Все независимые от A рекурсивные функции удаляются из множества $\Omega_1(A)$. В результате получаем множество связанных функций $\Omega_2(A)$ с графом связанных функций $G_2(A)$.

4. В $G_2(A)$ находим все вершины, имеющие только одну входную дугу. Код соответствующих косвенно рекурсивных функций может быть объединён с кодом функции из узла, смежного по входной дуге рассматриваемого узла. После объединения получаем $\Omega_3(A)$.

5. Задаём универсальную рекурсивную функцию F для всех функций из множества $\Omega_3(A)$.

Полученная в результате функция F будет прямой рекурсией.

Замечание 1. При доказательстве корректности функции A вначале необходимо доказать корректность всех независимых от A нерекурсивных и рекурсивных функций, которые удаляются из множества всех связанных функций в пунктах 2 и 3.

Замечание 2. Если узел нерекурсивной функции, найденной в пункте 2, имеет одну входную дугу, то код этой функции можно объединить с кодом функции, смежной по входной дуге.

Замечание 3. Если пункт 4 пропустить (после пункта 3 сразу переходить к пункту 5), то все функции из $\Omega_2(A)$ войдут в универсальную рекурсивную функцию.

Пример. В качестве иллюстрации работы алгоритма на рисунке 3.5а изображён граф всех связанных функций для функции A со множеством $\Omega(A)$:

$$\{A(A,B,C,D,E), B(A,C,H,I), C(B), D(J), E(F,G), F(E,F,G), G(), H(C), I(A), J()\}.$$

Три узла в графе (D, J и G, выделенные пунктиром) не лежат ни в каком цикле. Следовательно функции, которыми помечены эти узлы, являются нерекурсивными и удаляются из множества $\Omega(A)$. Полученное $\Omega_1(A)$ имеет вид:

$$\{A(A,B,C,E), B(A,C,H,I), C(B), E(F), F(E,F), H(C), I(A)\}.$$

На следующем этапе удаляются все рекурсивные функции, независимые от рассматриваемой функции A. Из графа связанных функций $\Omega_1(A)$ (рисунок 3.5б) видно, что не существует цикла, проходящего через узлы E или F и узел A. Значит, рекурсивные функции E и F не зависят от A и удаляются из $\Omega_1(A)$, получается (рисунок 3.5в):

$$\Omega_2(A) = \{A(A,B,C), B(A,C,H,I), C(B), H(C), I(A)\}.$$

Два узла I и H в графе для $\Omega_2(A)$ имеют по одной входной дуге, поэтому их код может быть объединён со смежным по их входной дуге узлу. Для обоих вершин это узел B. Назовём функцию, полученную в результате объединения, B_1, тогда $\Omega_3(A) = \{A(A,B_1,C), B_1(A,C), C(B_1)\}$ (рисунок 3.5г). Далее для трёх оставшихся функций формируется УРФ.

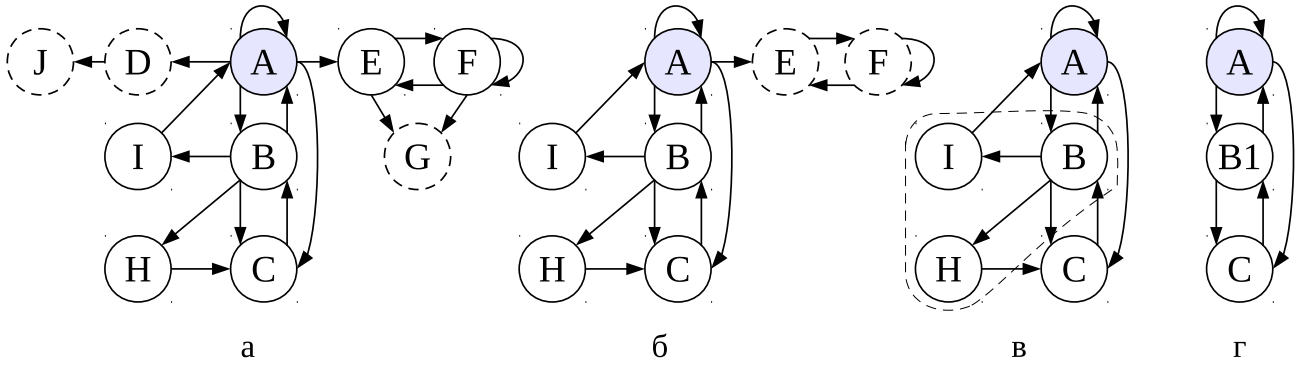


Рисунок 3.5 — Пример изменения графа связанных функций при преобразовании рекурсии A в прямую

3.2.4 Преобразование рекурсивных функций на языке Пифагор

Язык программирования Пифагор хорошо подходит для проведения преобразования программы с удалением взаимной рекурсии. В языке присутствует только один оператор, применяющий функцию к аргументу — оператор интерпретации. У оператора два входа, на которые поступают функция и аргумент функции. Аргумент всегда один, но он может быть списком данных с достаточно сложной структурой, которая может интерпретироваться как передача нескольких аргументов. Таким образом, у каждой функции во взаимной рекурсии один аргумент, и после преобразования в результирующей УРФ также будет один аргумент.

Рассмотрим удаление различных видов рекурсий из программ на языке Пифагор.

Объединение кода функций. Вначале рассмотрим случай косвенной рекурсии. Дано две функции A и B на языке Пифагор, с исходным кодом:

Листинг функции A:

```
A << funcdef x
{
    (
        {x:g0},
        {x:g1:A},
        {x:g2:B}
    ):
    [(c0, c1, c2):?]:.:p
    >> return;
}
```

Листинг функции B:

```
B << funcdef y
{
    (
        {y:e0},
        {y:e1:A}
    ):
    [(d0, d1):?]:.:q
    >> return;
}
```

В данном коде g_i, c_i, $i = 0, 1, 2$ и e_j, d_j, $j = 0, 1$, p, q — некоторые функции, независимые от функций A и B. Множество $\Omega(A) = \{A(A, B), B(A)\}$, соответствующее дерево всех связанных функций приведено на рисунке 3.4а.

Для удаления косвенной рекурсии объединим код функций A и B в функцию AB, как показано ниже:

```
AB << funcdef x
{
    (
        {x:g0},
        {x:g1:AB},
        {
            (
                {x:g2:e0},
                {x:g2:e1:AB}
            ):
            [(d0, d1):?]:.:q
        }
    ):
    [(c0, c1, c2):?]:.:p >> return;

}
```

Покажем, что объединение кода не изменяет корректность исходной программы. Пусть условия корректности рассматриваемой функции A задано в виде тройки:

$$\boxed{P_A}\ \text{A}\ \boxed{Q_A}.$$

При объединении кода получаем функцию AB с условием корректности

$$\boxed{P_{AB}}\ \text{AB}\ \boxed{Q_{AB}},$$

где $P_{AB} = P_A$, $Q_{AB} = Q_A$. Предположим, что корректность AB доказана, требуется показать, что A корректна. Очевидно, что из истинности $(P_{AB} \Rightarrow Q_{AB})$ следует истинность формулы

$(P_A \Rightarrow Q_A)$, значит, функция A корректна.

Построение универсальной рекурсивной функции. Рассмотрим удаление взаимной рекурсии. Возьмём схему рекурсии, представленную на рисунке 3.3. В общем случае исходный код функций A и B на языке Пифагор будет следующим:

Листинг функции A:

```
A << funcdef x
{
    (
        {x:g0},
        {x:g1:A},
        {x:g2:B}
    ):
    [(c0, c1, c2):?]:.:p
    >> return;
}
```

Листинг функции B:

```
B << funcdef y
{
    (
        {y:e0},
        {y:e1:A},
        {y:e2:B}
    ):
    [(d0, d1, d2):?]:.:q
    >> return;
}
```

где g_i, c_i, e_i, d_i, $i = 0, 1, 2$, p, q — некоторые функции, независимые от функций A и B.

Для удаления взаимной рекурсии введём универсальную рекурсивную функцию F следующим образом:

```
F << funcdef nx
{   n<<nx:1;
    x<<nx:2;
    (
        {
            (
                {x:g0},
                {(1,x:g1):F},
                {(2,x:g2):F}
            ):
            [(c0, c1, c2):?]:.:p
        },
        {
            (
                {x:e0},
                {(1,x:e1):F},
                {(2,x:e2):F}
            ):
            [(d0, d1, d2):?]:.:q >> return;
        }
    ):
    [((n,1):=,(n,2):=):?]:. >> return;
}
```

Полученная функция F является прямой рекурсией.

Покажем, что объединение функций в УРФ не изменяет корректность исходной программы. Пусть условие корректности функции A задано в виде тройки $\boxed{P_A}$ A $\boxed{Q_A}$, а функ-

ции B — $\boxed{P_B}$ В $\boxed{Q_B}$. Зададим для функции $F(n, x)$ следующее предусловие и постусловие:

$$P_F \equiv ((n = 1) \wedge P_A) \vee ((n = 2) \wedge P_B),$$

$$Q_F \equiv ((n = 1) \wedge Q_A) \vee ((n = 2) \wedge Q_B).$$

Тогда, при верном указании номера функции $n = 1$, получаем, что из корректности функции F будет следовать корректность исходной функций A:

$$(P_F \wedge (n = 1)) \Rightarrow P_A,$$

$$(Q_F \wedge (n = 1)) \Rightarrow Q_A.$$

Пример. Доказательство корректности рекурсивной программы рассмотрим на примере решения задачи распознавания простого арифметического выражения, порождаемого в соответствии со следующим правилом в нотации Бэкуса-Наура:

```
<выражение> ::= x | +<выражение><выражение> | *<выражение><выражение> .
```
(3.3)

Здесь x — терминальный операнд. Для упрощения примера введём следующие ограничения: функция должна принимать на вход строку с правильно построенным выражением и возвращать пустую строку, никакой обработки ошибок не требуется.

Распознавание данного выражения определяется следующими функциями на языке Пифагор:

```
parse << funcdef str  // Основная функция parse, принимающая строку str
                      // с арифметическим выражением
{   s1   << str:1;    // Сохранение первого символа строки str в s1
    case << ( (s1,'+'):=, // Сравнение s1 с тремя допустимыми
              (s1,'*'):=, // вариантами значений
              (s1,'x'):=
            ):?;          // Выбор номера пути вычисления и
                          // сохранение номера в case
    act  << ( {str:fp},   // Формирование списка act трёх различных путей
              {str:fm},   // вычислений, зависящих от значения s1
              {str:fn}
            );
    act:case:.   // Выбор нужного пути вычисления и активация вычисления
    >> return;   // Возврат результата вычислений
}

fp << funcdef str1 // Функция fp для разбора сложения, принимающая str1
{    str1:-1:parse:parse // Из строки str1 удаляется первый символ '+'
     >> return;          // и с помощью двух последовательных вызовов
                         // функции parse удаляется 1-ое и 2-ое слагаемое
}
```

```
fm << funcdef str2 // Функция fm для разбора умножения, принимающая str2
{    str2:-1:parse:parse >> return;   } // Аналогична fp

fn << funcdef str3 // Функция fn для разбора формального символа x
{    str3:-1 >> return;   } // Из входной строки str2 удаляется один символ
```

Основная функция, которая получает строку с арифметическим выражением, имеет имя `parse`. Также имеется три вспомогательные функции, которые отвечают за разбор одного из видов выражений: `fp` разбирает применение функции сложения, `fm` — функции умножения, а `fn` — формального символа «x». Функция `parse` определяет тип выражения по первому символу входной строки и передаёт эту строку соответствующей вспомогательной функции. И в качестве ответа возвращает результат работы вспомогательной функции. Функции `fp`, `fm` разбирают получаемую строку, удаляя из неё первый символ операции, а затем дважды применяя к ней функцию `parse`. Первый вызов функции `parse` удаляет из строки первое слагаемое (множитель), второй вызов — второе слагаемое (множитель), оставшийся «хвост» строки возвращается в качестве ответа. Функция `fn` разбирает получаемую строку удалением из неё первого символа и возвращает оставшийся «хвост» строки. Таким образом, если входное выражение для функции `parse` не содержит ошибок, то функция вернёт пустую строку, иначе результат непредсказуем.

На естественном языке спецификация к функции `parse` будет следующей:

> Входным аргументом для функции является строка символов str, в которой можно выделить две подстроки str_1 и $tail$ таких, что $str = str_1 \circ tail$, где «$\circ$» обозначает объединение (конкатенацию) двух строк. Строка str_1 удовлетворяет правилу (3.3), а строка $tail$ содержит произвольные символы, в том числе, она может быть пустой. В результате работы функция `parse` возвращает строку $tail$.

Запишем спецификацию функции `parse` на языке логики. Опишем множество функций S_F, аргументом которых является целое положительное число n_1, а результатом — строка длиной $(2n_1 - 1)$, удовлетворяющая условию (3.3):

$$S_F = \{f : (\Pi\, n_1 : \mathbb{N}.\, \mathrm{datalist}\, L_{(2n_1-1)}) \mid F(f)\},$$

где $L_{(2n_1-1)}$ — список вида $(\mathrm{char}, \mathrm{char}, \ldots, \mathrm{char})$, содержащий $(2n_1 - 1)$ элементов, то есть $(\mathrm{datalist}\, L_{(2n_1-1)}) \equiv (\mathrm{datalist}\, L)$ и выполнена формула

$$(L \in \mathrm{list\,Set}) \wedge (\mathrm{length}(L) = (2n_1 - 1)) \wedge (\forall i : \mathbb{N}.\, (0 < i \leqslant 2n_1 - 1) \Rightarrow (L[i] = \mathrm{char})).$$

F имеет тип $\big((\Pi\, n : \mathbb{N}.\, \text{datalist}\, L_{(2n-1)}) \to \text{bool}\big)$ и определяется следующим выражением:

$$F(f) \equiv \forall n_1 : \mathbb{N}.\, \Big(f(1) = \text{“x”}\Big) \wedge \Big(f(2) = \text{“+xx”} \vee f(2) = \text{“*xx”}\Big) \wedge$$
$$\wedge \left(\bigvee_{i=1}^{n_1-1} \Big(f(n_1) = \text{“+”} \circ f(i) \circ f(n_1 - i)\Big) \vee \Big(f(n_1) = \text{“*”} \circ f(i) \circ f(n_1 - i)\Big)\right), \quad (3.4)$$

Тогда предусловие $P(str)$ для функции `parse` можно записать на языке логической спецификации следующим образом:

$$\exists (n, m : \mathbb{N})(f : (\Pi\, n : \mathbb{N}.\, \text{datalist}\, L_{(2n-1)})).\, (f \in S_F) \wedge (str \in \text{datalist}\, L_{(2n-1+m)}) \wedge$$
$$\wedge \big(\exists str_1 : \text{datalist}\, L_{(2n-1)}.\, (str_1 = f(n)) \wedge (str_1 = \text{sublist}(str, 1, 2n-1))\big) \wedge$$
$$\wedge \big(\exists tail : \text{datalist}\, L_{(m)}.\, (tail = \text{sublist}(str, 2n, 2n-1+m))\big),$$

где $\text{sublist}(s, p, q)$ — функция, возвращающая подстроку строки s, начиная с позиции p до q, функция sublist возвращает пустую строку, если $p = q$.

Предусловие P описывает требования к входному аргументу str: любой входной аргумент str является строкой длины $2n - 1 + m$, в которой можно выделить подстроку str_1, длиной $2n - 1$, удовлетворяющую условию (3.3) (совпадает с $f(n)$ — результатом, возвращаемым одной из функций f множества S_F), оставшаяся часть строки $tail$ имеет длину m.

Постусловие $Q(return)$ функции `parse` зададим следующей формулой:

$$return = tail,$$

где $return$ обозначает результат вычислений программы.

Для доказательства корректности косвенно рекурсивной функции `parse` преобразуем её в прямую рекурсию. Множество всех связанных функций для `parse` имеет вид:

$$\Omega(\text{parse}) = \{\text{parse}(\text{fp}, \text{fm}, \text{fn}), \text{fp}(\text{parse}), \text{fm}(\text{parse}), \text{fn}()\},$$

граф всех связанных функций приведён на рисунке 3.6.

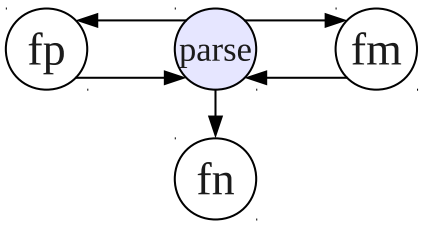


Рисунок 3.6 — Граф всех связанных функций для `parse`

Преобразование рекурсии `parse` в прямую при объединении кода. Связи функций из $\Omega(\text{parse})$ допускают объединение кода функций для удаления косвенной рекурсии. В $\Omega(\text{parse})$ функция `fn` является нерекурсивной, воспользуемся замечанием 2 алгоритма

преобразования рекурсий из раздела 3.2.3 и объединим код функции `fn` с кодом `parse`. Полученную функцию обозначим `parsefn`,

$$\Omega(\text{parsefn}) = \{\text{parsefn}(\text{fp}, \text{fm}), \text{fp}(\text{parsefn}), \text{fm}(\text{parsefn})\}.$$

В Ω(parsefn) отсутствуют независимые рекурсивные функции. Функции `fp` и `fm` имеют одну входную дугу, поэтому, согласно пункту 4 алгоритма преобразования рекурсий, объединим их код с кодом функции `parsefn`. Полученной функции дадим имя `parseUn`. Эта функция является прямой рекурсией и не требует задания УРФ. Ниже приведён исходный код `parseUn`:

```
parseUn << funcdef str // Функция parseUn, принимающая строку str с
                       // арифметическим выражением
{   s1   << str:1;     // Сохранение первого символа строки str в s1
    case << ( (s1,'+'):=, // Сравнение s1 с тремя допустимыми
              (s1,'*'):=, //  вариантами значений
              (s1,'x'):=
            ):?;          // Выбор номера пути вычисления и
                          //сохранение номера в case
    act << (  // Формирование списка act трёх различных путей вычислений
            {str:-1:parseUn:parseUn }, //Результат объединения кода с fp
            {str:-1:parseUn:parseUn }, //Результат объединения кода с fm
            {str:-1}                   //Результат объединения кода с fn
          );
    act:case:.           // Выбор нужного пути и запуск вычисления
    >> return;           // Возврат результата вычислений
}
```

Докажем частичную корректность `parseUn`. Предусловие и постусловие `parseUn` совпадают с предусловием и постусловием `parse` (см. раздел 3.2.3). Информационный граф `parseUn` приведён на рисунке 3.7а. Для удобства все дуги графа пронумерованы. Считаем, что номер выходной дуги задаёт и номер оператора (узла).

Дуги графа `parseUn` размечаются в порядке готовности данных: если все входные дуги узла размечены, можно размечать выходной узел. Узел 1 размечен предусловием, дуги 2–10 являются выходными дугами констант и получают автоматическую разметку, дуги 11–13 — выходные дуги задержанных списков, которые до снятия задержки являются константами. Дуга 14 с идентификатором act — выходная дуга списка данных также размечается автоматически. Дуга 15 с идентификатором s_1 может быть размечена на основе аксиом функции выбора элемента из списка. В результате к дуге s_1 приписывается формула $(s_1 \in \text{char} \wedge s_1 = str[1])$, где квадратные скобки обозначают выбор элемента из списка. В данном случае нельзя воспользоваться эквивалентным преобразованием с перенаправлением связей, так как длина строки неизвестна. Дуги 16–18 являются выходными дугами списков данных и размечаются автоматически после разметки дуги s_1. На основе аксиом для функции «=» можно разметить дуги 19–21. В результате к дуге x_1 припишутся две формулы:

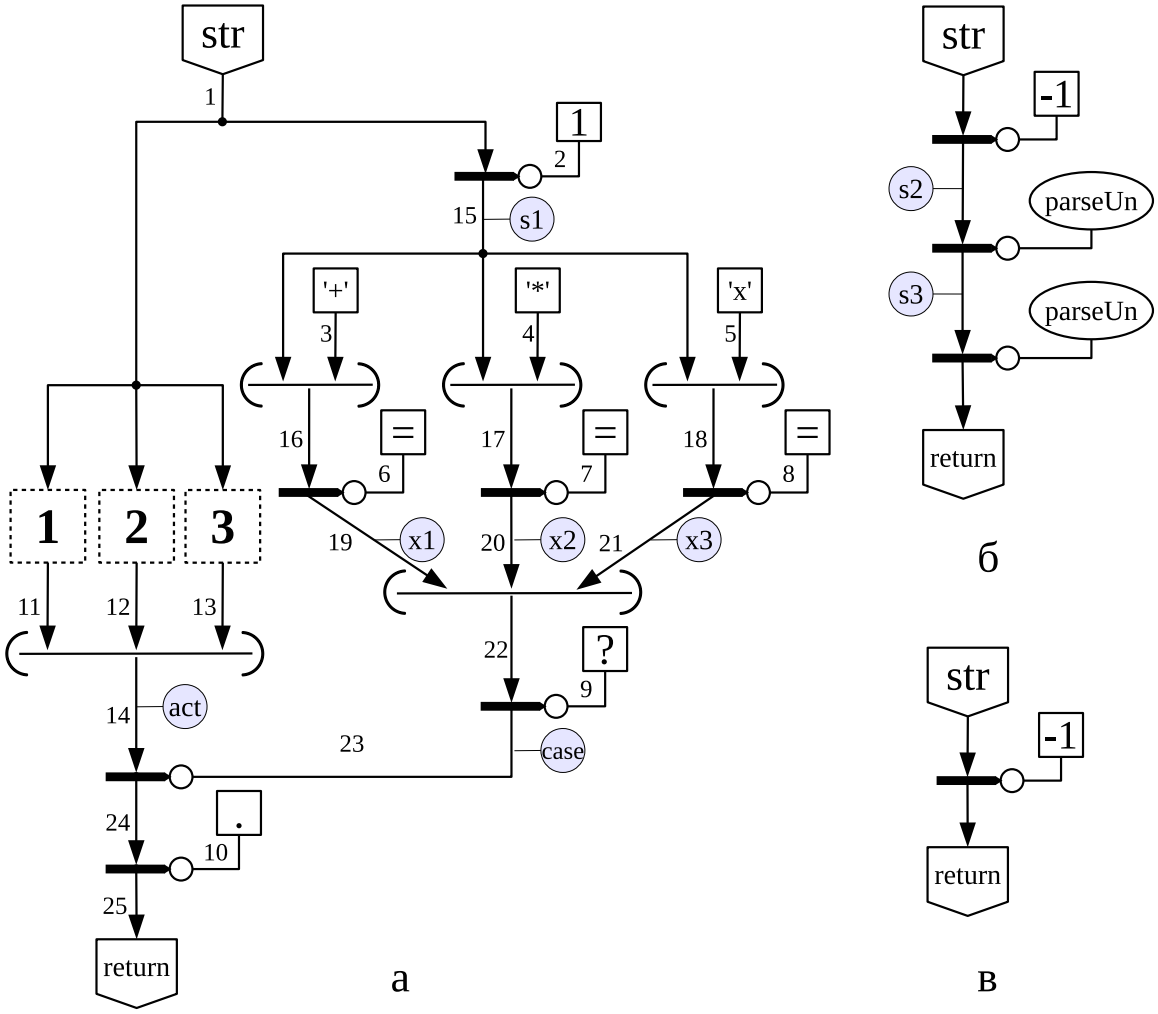


Рисунок 3.7 — Информационные графы функции `parseUn`, к некоторым дугам приписаны идентификаторы, обозначенные серыми кружками; а — исходный информационный граф функции `parseUn`, все дуги графа пронумерованы, задержанные списки обозначены пунктиром и пронумерованы; б, в — информационные графы функции `parseUn` после расщепления и раскрытия задержанных списков

$$(x_1 \in \text{bool}) \wedge (x_1 = \text{true}) \wedge (s_1 = \text{`} + \text{'}),$$

$$(x_1 \in \text{false}) \wedge (x_1 = \text{false}) \wedge \neg(s_1 = \text{`} + \text{'}),$$

где одинарные кавычки «'» используются для записи символьной константы типа char. К дугам x_2 и x_3 будут приписаны аналогичные формулы, в которых заменён идентификатор дуги и символьная константа на '$*$' и 'x' соответственно.

Дуга 22 размечается автоматически. К 23 дуге с идентификатором *case*, при разметке на основе аксиомы для функции «?», приписываются три формулы:

$$(case \in \text{int}) \wedge (case = 1) \wedge (x_1 = \text{true}) \wedge (x_2, x_3 = \text{false}),$$

$$(case \in \text{int}) \wedge (case = 2) \wedge (x_2 = \text{true}) \wedge (x_1, x_3 = \text{false}),$$

$$(case \in \text{int}) \wedge (case = 3) \wedge (x_3 = \text{true}) \wedge (x_1, x_2 = \text{false}).$$

Применение *case* как функции к аргументу *arg* при выполнении оператора интерпретации 24 приведёт к расщеплению исходного графа на три, в каждом из которых произойдёт раскрытие задержанного списка при применении функции «.». Пусть номера полученных графов совпадают с номерами их задержанных списков. Первый и второй полученные графы одинаковые и приведены на рисунке 3.7б, а третий — на рисунке 3.7в.

Рассмотрим первый граф. Его предусловие P_1 будет следующим: $P \wedge (s_1 \in \text{char}) \wedge$ $\wedge (s_1 = str[1]) \wedge (s_1 = \text{`}+\text{'})$, а постусловие $Q_1 = Q$. То есть предусловие P_1 отличается от исходного предусловия P, тем, что в нём присутствуют формулы, которые описывают причины выбора данного пути вычисления: первый символ строки *str* равен символу '+'. Для разметки дуги s_2 используются аксиомы функции выбора элемента из списка, которая при отрицательном входном аргументе удаляет элемент из списка. В результате к дуге s_2 приписывается формула:

$$(s_2 \in \text{datalist}\, L_{(2n-2+m)}) \wedge (s_2 = \text{sublist}(str, 2, 2n-1+m)).$$

Далее к результату s_2 применяется функция `parseUn`, то есть производится рекурсивный вызов. Для доказательства частичной корректности программы достаточно предположить, что все рекурсивные вызовы функции возвращают верный результат, если их аргумент удовлетворяет предусловию. Покажем, что s_2 удовлетворяет предусловию `parseUn` P. Так как строка *str* удовлетворяет предусловию и первый её символ $s_1 = \text{`}+\text{'}$ (следует из P_1), значит, $f(n)$ (из P) по правилу (3.4) будет имеет вид $f(n) = \text{“}+\text{”} \circ f_1(k) \circ f_2(n-k)$, для некоторого k и $f_1, f_2 \in S_F$. На языке логики это утверждение можно выразить следующей формулой:

$$\exists (k : \mathbb{N})(f_1, f_2 \in S_F).\, (0 < k < n) \wedge \big(f(n) = \text{“}+\text{”} \circ f_1(k) \circ f_2(n-k)\big).$$

Следовательно, при удалении первого символа из $str = f(n) \circ tail$, получим $s_2 = f_1(k) \circ$ $\circ f_2(n-k) \circ tail$. Таким образом, в начале строки s_2 стоит подстрока $f_1(k)$, удовлетворяющая (3.3), а «хвост» $tail_1 = f_2(n-k) \circ tail$ имеет длину $m_1 = 2(n-k) - 1 + m$. Тогда по индуктивному предположению s_3 — результат применения функции `parseUn` к s_2, будет удовлетворять постусловию: $s_3 = tail_1$. И к дуге s_3 будет приписана формула:

$$\exists (k, m_1 : \mathbb{N})(f_1 \in S_F).\, (0 < k < n) \wedge (\text{sublist}(s_2, 1, 2k-1) = f_1(k)) \wedge (s_3 = \text{sublist}(s_2, 2k, 2k-1+m_1)).$$

Поскольку $s_3 = f_2(n-k) \circ tail$, то s_3 также удовлетворяет предусловию функции `parseUn`. Тогда по индуктивному предположению при применении `parseUn` к s_3 получаем $return = tail$. Постусловие Q всегда следует из этой формулы, следовательно первый информационный граф с разметкой эквивалентен тождественно истинной формуле.

Второй граф (рисунок 3.7б) полностью совпадает с первым. Отличие затрагивает только его предусловие P_2, в котором символ '+', заменён на '$*$'. Остальные формулы разметки полностью совпадают с соответствующими формулами первого графа, поэтому второй ИГР также эквивалентен тождественно истинной формуле.

Рассмотрим третий граф (рисунок 3.7в). Его предусловие P_3 имеет вид: $(P \wedge (s_1 \in \text{char}) \wedge (s_1 = str[1]) \wedge (s_1 = \text{'}x\text{'}))$, а постусловие $Q_3 = Q$. К входной строке str применяется функция удаления первого элемента, в результате к дуге $return$ приписывается формула $return = sublist(str, 2, 2n - 1 + m)$. Исходя из (3.4) и того, что $s_1 = \text{'}x\text{'}$, $str = \text{'}x\text{'} \circ tail$, при удалении первого символа str получаем, что $return = tail$. Следовательно третий ИГР эквивалентен тождественно истинной формуле. Из истинности всех трёх ИГР следует частичная корректность программы `parseUn`, а значит, функция `parse` тоже частично корректна.

Преобразование рекурсии `parse` в прямую с помощью универсальной рекурсивной функции. Метод объединения кода функций применим в достаточно простых случаях, в сложных ситуациях необходимо использовать более сложный, но универсальный метод построения универсальной рекурсивной функции. Данный метод можно применить и для удаления взаимной рекурсии в функции `parse`. Для этого применим алгоритм преобразования произвольной рекурсии в прямую из раздела 3.2.3 к Ω(parse) ещё раз. В Ω(parse) функция `fn` является нерекурсивной, удалим её из Ω(parse) (её корректность докажем отдельно), получим

$$\Omega_1(\text{parse}) = \{\text{parse}(\text{fp}, \text{fm}), \text{fp}(\text{parse}), \text{fm}(\text{parse})\}.$$

В Ω_1(parse) отсутствуют независимые от `parse` рекурсивные функции. Пропустим шаг 4 алгоритма преобразования рекурсий и сразу зададим УРФ, которую назовём `pURF`. Ниже приведён исходный код `pURF`:

```
pURF << funcdef sn{ // Функция pURF принимающая список (N, str),
                    // N - номер функции, объединённой в УРФ, str - строка
    N << sn:1;      // Сохранение первого аргумента в N
    s << sn:2;      // Сохранение второго аргумента в s
    csU << (
        (N,1):=,
        (N,2):=,
        (N,3):=   // Сравнение N с тремя допустимыми вариантами значений
    ):?;          // Выбор номера пути вычисления и его сохранение в csU
    actU << (
         {  block{ // Тело функции parse с заменой всех вызовов функций
                   // на вызов УРФ
                s1 << s:1;
                case << ( (s1,'+'):=, (s1,'*'):=, (s1,'x'):= ):?;
                act << ( {(2,s):pURF}, {(3,s):pURF}, {s:fn} );
                act:case:.>>break;
```

```
            }
         },
         { // Тело функции fp с заменой всех вызовов функций на вызов УРФ
            ( 1, (1, s:-1):pURF ):pURF
         },
         { // Тело функции fm с заменой всех вызовов функций на вызов УРФ
            ( 1, (1, s:-1):pURF ):pURF
         }
   ); // Формирование списка actU 3-х различных путей вычислений
      // в зависимости от N
   actU:csU:. // Выбор нужного пути вычисления и активация вычисления
   >> return; // Возврат результата вычислений
}
```

Информационный граф функции `pURF` приведён на рисунке 3.8. Согласно разделу 3.2.4 предусловие и постусловие функции `pURF` формируется из предусловий и постусловий функций, входящих в УРФ. Зададим предусловие и постусловие для `fp` (для `fm` всё аналогично). Функция `fp` должна принимать строку str_1, которая начинается с символа ‘+’, далее идут

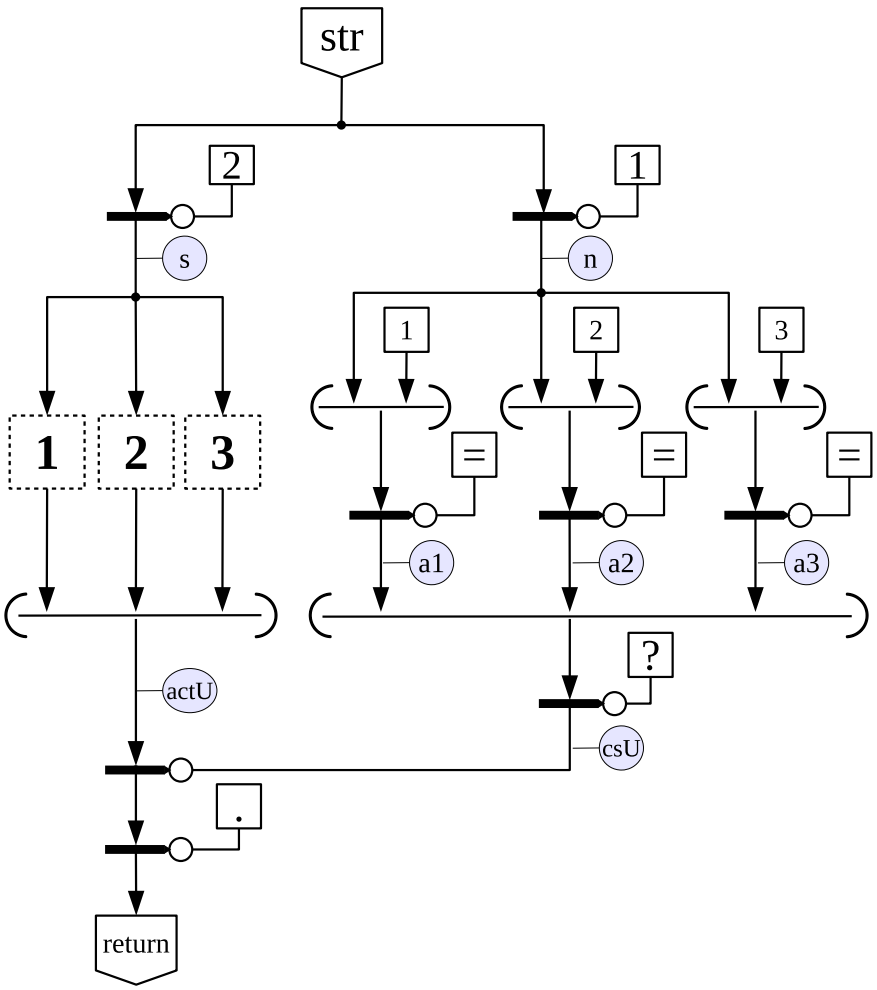


Рисунок 3.8 — Информационный граф функции `pURF`; задержанные списки обозначены пунктиром и пронумерованы

две подстроки, удовлетворяющие (3.3), а оставшаяся часть строки $tail$ может содержать любые символы. В результате работы функция должна вернуть $tail$. На формальном языке предусловие P_{fp} и постусловие Q_{fp} выражаются, соответственно, следующими формулами:

$$P(str_1) \wedge \big(\exists (k \in \mathbb{N})(f_1, f_2 \in S_F).\, (0 < k < n) \wedge (str_1 = \text{“}+\text{”} \circ f_1(k) \circ f_2(n-k) \circ tail)\big),$$

$$return = tail,$$

где $P(str_1)$ — предусловие функции `parse`, в котором имя переменной str заменено на str_1. Тогда предусловие функции `pURF` со входным аргументом $sn = (N, s)$ будет следующим:

$$((N = 1) \wedge P(s)) \vee ((N = 2) \wedge P_{fp}(s)) \vee ((N = 3) \wedge P_{fm}(s)).$$

После упрощения получаем предусловие P_{URF}:

$$P(s) \wedge ((N = 1) \vee (N = 2 \wedge s[1] = \text{`}+\text{'}) \vee (N = 3 \wedge s[1] = \text{`}*\text{'}))).$$

Оно означает, что строка s удовлетворяет $P(s)$, то есть представима в виде $f(n) \circ tail$, $f \in S_F$; при $N = 2$ первый символ s равен ‘+’, а при $N = 3$ первый символ — ‘∗’.

Постусловие функции `pURF` имеет вид $((N = 1) \wedge Q) \vee ((N = 2) \wedge Q_{fp}) \vee ((N = 3) \wedge Q_{fm})$, и после упрощения приводится к формуле $return = tail$, то есть $Q_{URF} = Q$.

После автоматической разметки выходных дуг константных операторов и списков данных в графе `pURF` (рисунок 3.8), размечаются дуги a_i, $i = 1, 2, 3$. К каждой дуге a_i приписывается пара формул:

$$(a_i \in \text{bool}) \wedge (a_i = \text{true}) \wedge (N = i),$$

$$(a_i \in \text{bool}) \wedge (a_i = \text{false}) \wedge \neg(N = i).$$

Далее, к дуге csU приписывается три формулы:

$$(csU \in \text{int}) \wedge (csU = i) \wedge (a_i = \text{true}) \bigwedge_{j \neq i} (a_j = \text{false}),\ i = 1, 2, 3.$$

Применение csU как функции к аргументу $argU$ приводит к расщеплению исходного графа на три, в каждый из которых попадёт один задержанный список. При применении функции «.» произойдёт раскрытие задержанных списков. Через G_1, G_2, G_3 обозначим графы, полученные в результате снятия задержки. Нижний индекс 1, 2, 3 соответствует номеру задержанного списка, попавшего в граф. Граф G_1 совпадёт с исходным графом функции `parseUn`, приведённом на рисунке 3.7а, если в нём заменить идентификатор входного аргумента str на s (в этих графах отличаются константы задержанных списков, но на рисунке это не отражено). Графы G_2 и G_3 одинаковые и приведены на рисунке 3.9а.

Рассмотрим граф G_1. Его предусловие P_{URF1} имеет вид: $P_{URF} \wedge (N = 1)$. А после упрощения может быть записано как $P(s) \wedge (N = 1)$. Граф G_1 размечается так же, как

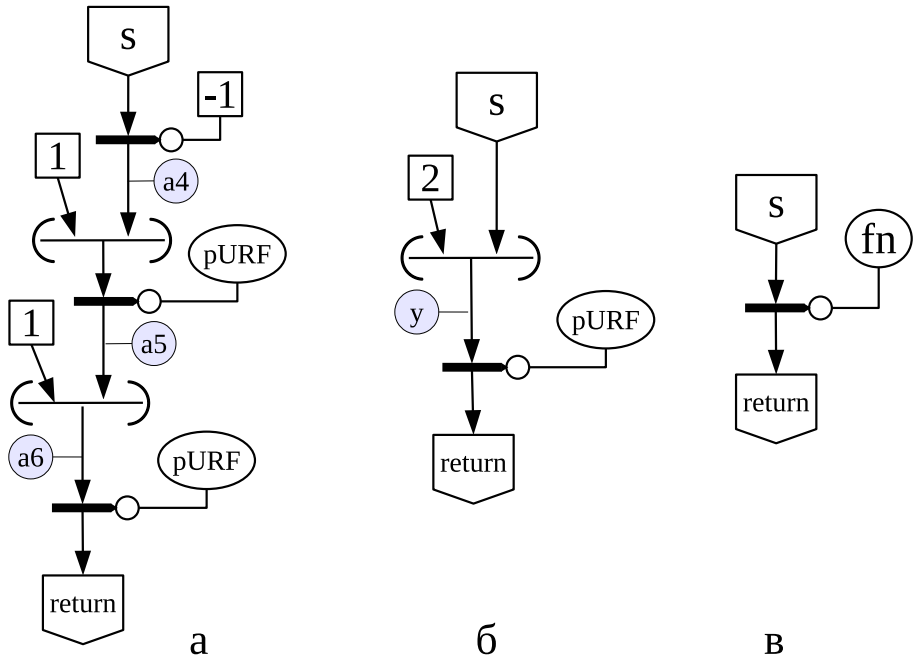


Рисунок 3.9 — Модифицированные информационные графы функции `pURF`

исходный граф `parseUn`. Применение *case* к *act* приводит к расщеплению графа на три, и после снятия задержки получаются графы, которые обозначим G_{11}, G_{12} и G_{13}, где вторая цифра в индексе соответствует номеру попавшего в граф задержанного списка. Граф G_{11} приведён на рисунке 3.9б. Граф G_{12} такой же, как G_{11}, только константа «2» заменяется на «3». Граф G_{13} приведён на рисунке 3.9в.

Разметим граф G_{11}. Предусловие G_{11} имеет вид:

$$P_{URF} \wedge (N = 2) \wedge (s[1] = \text{`}+\text{'}).$$

Значит, для аргумента (N, s), имеем $N = 2$, а строка s представима как $f(n) \circ tail$, $(f \in S_F)$, и первый её символ равен '+'. Это эквивалентно тому, что существует натуральное число k, такое что s представимо как $\text{“}+\text{”} \circ f_1(k) \circ f_2(n-k) \circ tail$, $(f_1, f_2 \in S_F)$.

В графе G_{11} константа «2» и аргумент s формируют список данных, которому присваивается идентификатор y. Далее к y применяется рекурсивный вызов функции `pURF`. Покажем, что y удовлетворяет предусловию функции `pURF`. Действительно, из предусловия графа G_{11} следует $P_{URF}(2, s)$ — предусловие функции `pURF`, для которого входной аргумент $(N, s) = (2, s)$. Тогда, по индуктивному предположению, рекурсивный вызов $pURF(y)$ вернёт верный ответ, и к дуге *return* припишется формула: $return = tail$. Следовательно, постусловие выполнено, и полностью размеченный граф G_{11} эквивалентен тождественно истинной формуле.

Для G_{12} всё аналогично, поэтому после разметки он также будет соответствовать

тождественно истинной формуле.

В графе G_{13} (рисунок 3.9в) предусловие G_{13} имеет вид:

$$P_{URF} \wedge (N = 3) \wedge (s[1] = \text{`}x\text{'}).$$

К аргументу s применяется функция `fn`, а результат её работы присваивается $return$.

Докажем корректность функции `fn`. Зададим предусловие P_{fn} и постусловие Q_{fn} для функции `fn`:

$$P(str_3) \wedge (str_3 = \text{“}x\text{”} \circ tail)\,,$$

$$return = tail.$$

В теле функции `fn` единственная функция, которая применяется к входному аргументу — это функция удаления первого элемента из списка. В результате из строки str_3 удаляется первый символ ‘x’, поэтому результат $return = \text{sublist}(s, 2, 1 + m) = tail$, что и требуется в постусловии. Следовательно, функция `fn` корректна.

На основе доказанной теоремы (корректности `fn`) можно разметить выходную дугу графа G_{13}: $return = tail$. Из последней формулы следует постусловие Q_{URF}. Значит, полностью размеченный граф G_{13} сворачивается в тождественно истинную формулу.

Остаётся разметить графы G_2 и G_3 (рисунок 3.9а). Рассмотрим граф G_2, для G_3 всё аналогично. Предусловие P_{URF2} имеет вид: $P_{URF} \wedge (N = 2)$. После упрощения P_{URF2} можно записать $P(s) \wedge (N = 2) \wedge (s[1] = \text{`}+\text{'})$, откуда следует, что для некоторого натурального k строку s можно представить в виде “ $+$ ” $\circ\, f_1(k) \circ f_2(n-k) \circ tail$, $(f_1, f_2 \in S_F)$.

Вначале к аргументу s применяется функция удаления первого элемента из списка, и результату $f_1(k) \circ f_2(n-k) \circ tail$ присваивается идентификатор a_4. Список данных $(1, a_4)$ — допустимый аргумент для рекурсивного вызова функции `pURF`, поэтому по индуктивному предположению результат a_5 удовлетворяет постусловию `pURF` и равен $f_2(n-k) \circ tail$. Новый формируемый список данных $(1, a_5)$ также удовлетворяет предусловию `pURF`, поэтому, по индуктивному предположению, $return$ будет равен $tail$. Откуда следует постусловие Q_{URF}. Значит, полностью размеченный граф G_2 (и аналогичный ему G_3) эквивалентен тождественно истинной формуле.

Таким образом, корректность `pURF` доказана, из этого следует корректность функции `parse`.

Выводы

В главе разрабатывается метод формальной верификации рекурсивных ФПП программ с использованием аксиоматической теории для языка Пифагор. Корректность рекур-

сивных функций доказывается в два этапа: доказывается частичная корректность (в предположении, что программа завершается) и завершение программы.

Если программа состоит из нескольких функций, то корректность каждой доказывается отдельно. Если функции связаны косвенной рекурсией, то перед доказательством необходимо преобразовать их в прямую рекурсию. Рассмотренный алгоритм позволяет преобразовать произвольную рекурсию в прямую при помощи универсальной рекурсивной функции или, в более простых случаях, путём объединения кода функций.

Описанный метод доказательства формирует достаточное условие корректности программы. В случае, если не удаётся доказать частичную корректность, то это может свидетельствовать о том, что присутствует ошибка в предусловии, постусловии или программе. Если программа в результате работы возвращает результат типа error, то по дереву доказательства можно проследить место возникновения первой ошибки. Также отметим, что доказательство истинности тройки не означает отсутствие ошибок одновременно в предусловии или в постусловии и в самой программе, так, например, следующие тройки всегда истинны [136]:

$$\boxed{\text{false}}\ \texttt{Prog}\to r\ \boxed{\psi(r)}\,,\qquad \boxed{\varphi(x)}\ \texttt{Prog(x)}\to r\ \boxed{\text{true}}\,,\qquad \boxed{\varphi(x)}\ \texttt{Rec(x)}\to r\ \boxed{\psi(r)}\,,$$

где φ, ψ — произвольные формулы, `Rec(x)` — бесконечная рекурсия.

В случае, если не удаётся доказать завершение программы, это может свидетельствовать о том, что программа содержит бесконечную рекурсию или неправильно выбрана ограничивающая функция. В последнем случае придется искать другую ограничивающую функцию.

Предложенный метод верификации ФПП программ опробован на функциях из библиотек языка Пифагор [156, 157]: математических функциях и функциях обработки строк. Спецификация формировалась на основе описания функций на естественном языке. В результате анализа было выявлено несколько функций, не соответствующих спецификации. Для устранения несоответствий у части функций достаточным оказалось изменение спецификации путём добавления дополнительных ограничений на входной аргумент. Другой тип проблем заключался в использовании неучтённых ранее особенностей выполнения встроенных функций, которые после формализации семантики стали приводить к ошибкам. В этом случае для устранения несоответствия спецификации необходимые изменения вносились в код программы. Применение метода также показало, что формализация спецификации и доказательство корректности программы — достаточно трудоемкий процесс, зачастую занимающий больше времени, чем разработка алгоритма и написание кода программы. Поэтому необходима разработка инструментально средства для поддержки процесса доказательства.

4 Инструментальная поддержка формальной верификации ФПП программ

В главе 4 описывается инструментальное средство поддержки формальной верификации ФПП программ. Рассматривается архитектура системы, разделение реализации системы на модули и интерфейс пользователя.

Процесс доказательства корректности программ на языке Пифагор, описанный в предыдущем разделе, достаточно трудоемок, так как доказательство истинности исходной тройки сводится к доказательству истинности нескольких преобразованных троек. При этом на каждом этапе доказательства необходимо проверять возможность проведения эквивалентных преобразований, выбирать применимые для разметки теоремы и рассматривать несколько вариантов применимых формул. Всё это значительно усложняет процесс доказательства. В то же время, ряд этапов допускает автоматизацию, например, проведение эквивалентных преобразований, разметка операторов, отличных от оператора интерпретации. Поэтому для упрощения процесса доказательства необходимо инструментальное средство, обеспечивающее поддержку формальной верификации ФПП программ [179, 184, 185, 186].

На рисунке 4.1 приведена общая схема системы для поддержки формальной верификации ФПП программ.

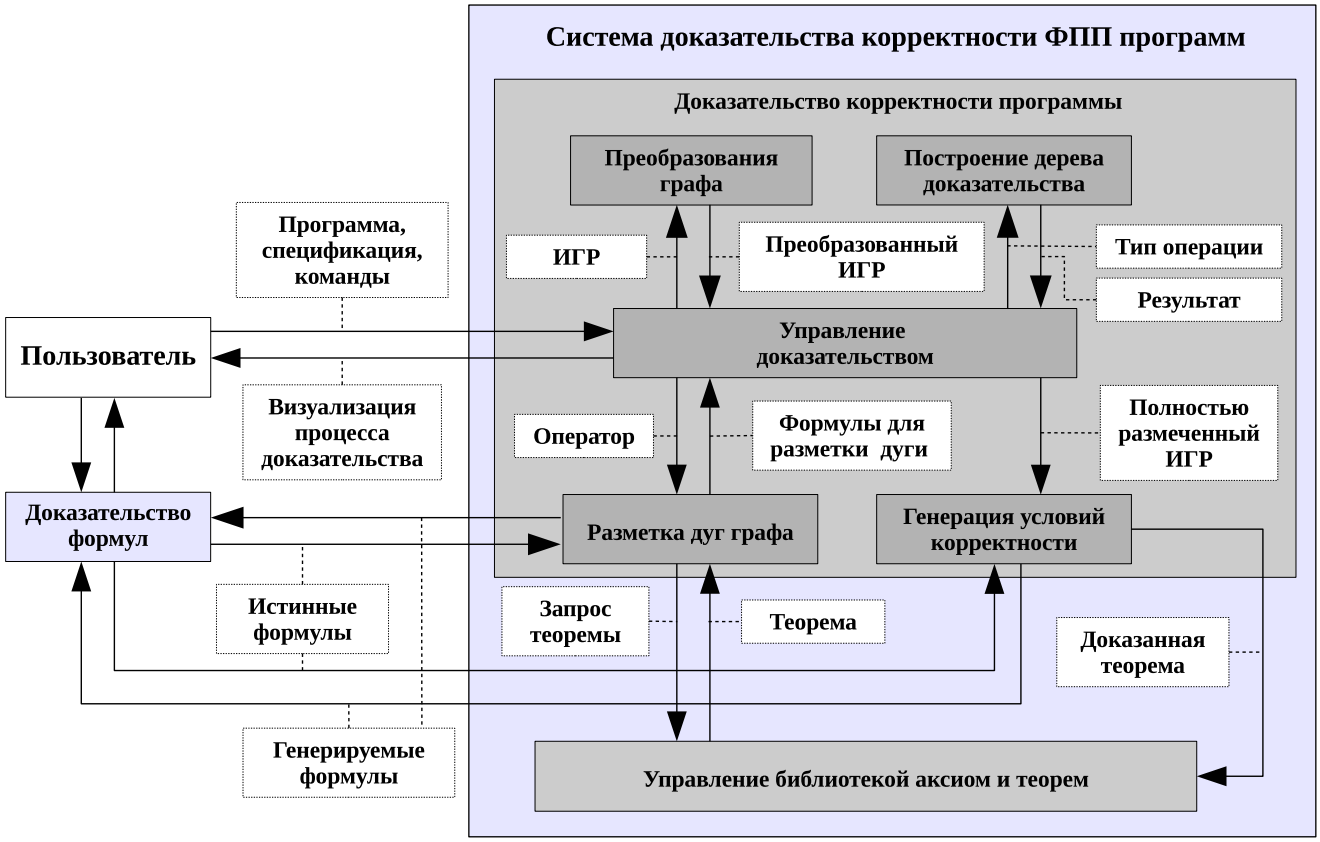


Рисунок 4.1 — Обобщённая структура системы поддержки формальной верификации; модули обозначены закрашенными прямоугольниками, стрелки — информационные связи между модулями

В системе можно выделить несколько элементов: модуль доказательства корректности программы, модуль управления библиотекой аксиом и теорем, модуль доказательства формул (верификатор формул). Модуль доказательства формул обособлен от остальной части системы, так как является сторонним средством и может не использоваться при доказательстве, в этом случае все его функции выполняются пользователем. В модуле доказательства корректности программы выделяется несколько компонентов. Основным является блок управления доказательством. Вспомогательные блоки отвечают за преобразования графа, построение дерева доказательства, разметку дуг и генерацию условий корректности.

Принцип работы системы состоит в следующем. Пользователь передаёт системе программу на языке Пифагор и спецификацию программы на языке спецификации. Модуль доказательства корректности программы формирует исходный ИГР (которому соответствует исходная тройка Хоара) и начинает процесс доказательства, заключающийся в последовательности преобразований ИГР и формировании дерева доказательства. Пользователь управляет процессом, выбирая тип следующего преобразования. Эквивалентные преобразования и разметка готовых операторов, отличных от оператора интерпретации, проводятся полностью автоматически. Для разметки выходных дуг операторов интерпретации используется информация об аксиомах и уже доказанных теоремах из библиотеки аксиом и теорем. Модуль доказательства корректности программы посылает запросы к модулю управления библиотекой аксиом и теорем. В случае, если запрашиваемая функция отсутствует в библиотеке, выдаётся ошибка о невозможности разметки. Если аксиомы (теоремы) для рассматриваемой функции присутствуют в библиотеке, то модуль доказательства корректности программы проводит их отбор. Для каждой аксиомы (теоремы) из набора аксиом (теорем) для рассматриваемой функции формируется условие применимости (2.4) данной аксиомы (теоремы) на языке спецификации. Это условие передаётся стороннему модулю доказательства формул, и, если оно выполнимо, то аксиома (теорема) используется для разметки, иначе она отбрасывается. После того как все дуги в информационном графе программы размечены, проводится полная свёртка. Тройки Хоара, соответствующие полученным ИГР, преобразуются в условия корректности, которые передаются верификатору для проверки истинности. Если все формулы истинны, то программа корректна, а её исходная тройка Хоара — теорема. Модуль доказательства корректности программы имеет возможность передать полученную теорему модулю управления библиотекой аксиом и теорем для сохранения в библиотеке. Блок-схема работы системы приведена в приложение З.

Предполагается, что рассмотренная система может работать в нескольких режимах: полностью ручной, частично автоматизированный и автоматизированный. В первом случае пользователь использует только блок управления доказательством и блок генерации условий

корректности. Он самостоятельно размечает дуги графа формулами, без обращения к библиотеке, и доказывает истинность условий корректности. В частично автоматизированном режиме не задействован только модуль доказательства формул, и пользователь сам указывает, истинна, ложна или выполнима сгенерированная формула. В автоматизированном режиме задействованы все модули.

4.1 Реализация системы

В соответствии с рассмотренной обобщённой структурой разработана программа, обеспечивающая поддержку формальной верификации ФПП программ, которая позволяет строить дерево доказательства программы на языке Пифагор. Разработанная система является единым приложением с графическим интерфейсом, состоящим из взаимосвязанных компонент. На рисунке 4.2 приведена общая схема реализации системы, отражены основные модули и их взаимосвязи между собой. Используются обозначения из [158].

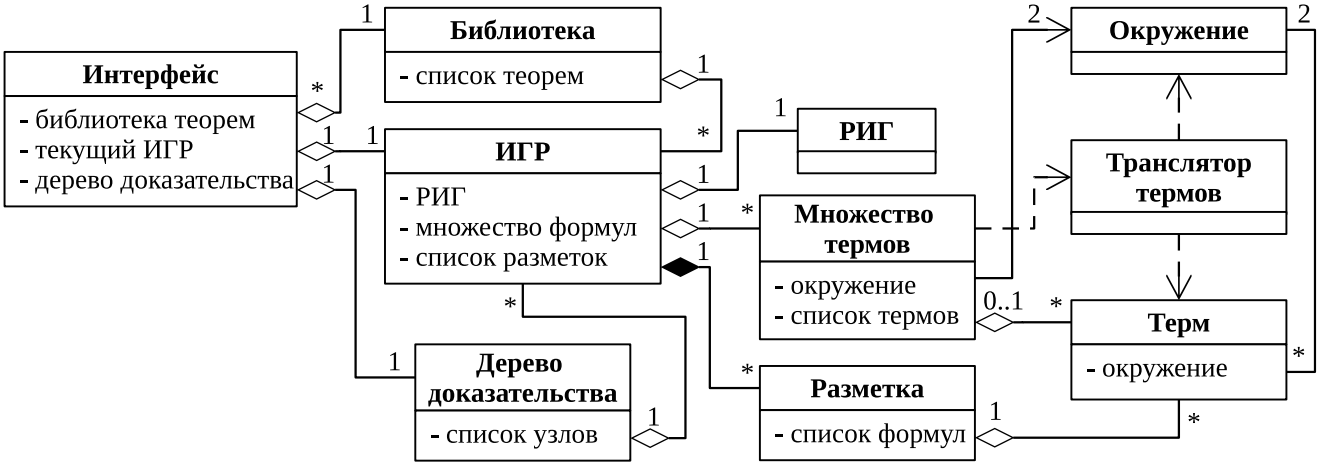


Рисунок 4.2 — Общая схема реализации системы поддержки формальной верификации ФПП программ

Базовым в системе является класс **ИГР**, который совмещает в себе основную функциональность для работы с ИГР. **Интерфейс** использует методы данного класса для проведения преобразований, выполняет запросы к библиотеке и формирует дерево доказательства при изменении ИГР. Система использует общие внутренние форматы данных. Основными объектами в модели данных являются: РИГ, формула (терм), множество, окружение, ИГР, дерево доказательства, библиотека аксиом и теорем.

4.2 Модуль поддержки РИГ

Модуль поддержки реверсивного информационного графа (РИГ) отвечает за внутреннее представление РИГ в памяти компьютера, считывание и запись РИГ в файлы, трансляцию из исходных кодов программы и перевод РИГ в графический формат dot.

4.2.1 Текстовое и внутренне представление информационного графа программы

Каждая функция на языке Пифагор имеет свой информационный граф. При трансляции функции порождается *реверсивный информационный граф* (РИГ), который описывает существующие в программе зависимости по данным и отличается от информационного графа тем, что дуги ориентированны в противоположном направлении [159, 160].

РИГ имеет текстовое и внутреннее представление [166, 187, 188]. Выбор текстового представления для описания РИГ обусловлен тем, что формирование на его основе внутреннего представления в памяти компьютерной системы может быть легко выполнено с помощью простых транслирующих программ. Помимо этого, разработчик может легко читать и анализировать оттранслированные функции, рассматривая данную форму как аналог языка ассемблера. Пример текстового представления РИГ приведён в приложении И.

Внутренне представление РИГ основывается на его текстовом представлении. Схема внутреннего представления приведена на рисунке 4.3.

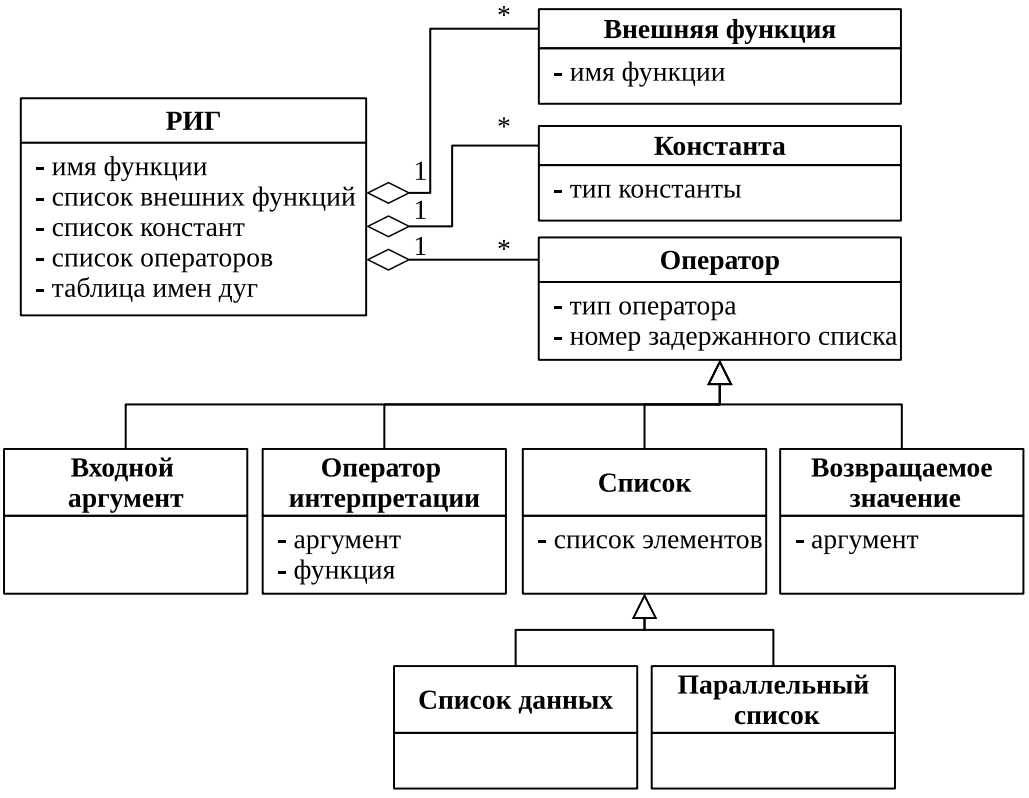


Рисунок 4.3 — Внутренне представление РИГ

Основным является класс **РИГ** (**Rig**), который содержит имя функции, списки внешних функций, констант, операторов и таблицу имён выходных дуг узлов.

У каждой внешней функции имеется номер внешней ссылки (соответствует позиции

в списке) и её имя. На нулевой позиции всегда располагается ссылка на саму функцию. Это позволяет использовать обращение к самой себе в случае рекурсивных вызовов.

Локальные константы — константы, используемые в ходе выполнения функции. Каждой константе соответствует свой номер (соответствует позиции в списке), который при выполнении программы обеспечивает доступ к соответствующим данным. Наряду с числовыми и символьными данными в списке хранятся константы, определяющие параметры задержанных списков. Каждая из таких констант хранит номер задержанного списка, а также номер узла РИГ, возвращающего вычисленное значение из данного задержанного списка.

Каждый оператор из списка операторов имеет номер, который соответствует его положению в списке. Каждый узел имеет свой тип, определяющий тип оператора, а также содержит информацию о номере задержанного списка, в котором он расположен. Если оператор РИГ не находится в задержанном списке, то номер равен нулю. Каждый узел содержит список связей (ссылок) на источники данных. Источниками данных могут быть внешние ссылки, локальные константы, предопределённые символы и узлы РИГ.

Выходная дуга любого узла РИГ может иметь имя (идентификатор). Имена всех дуг хранятся в таблице имён РИГ.

Внутреннее представление РИГ допускает произвольный порядок расположения операторов в списке. Однако в рамках системы доказательства на внутренне представление накладывается следующее ограничение: каждый оператор в списке операторов должен стоять после всех своих ссылок. РИГ, удовлетворяющий данному ограничению, назовём *допустимым*. Данное ограничение требуется для упрощения преобразований РИГ.

4.2.2 Основные функции модуля поддержки РИГ

В модуль РИГ входят следующие компоненты.

1. Транслятор исходных кодов программ на языке Пифагор [161, 162], который принимает на вход файл с исходным кодом, осуществляет его лексический и синтаксический анализ и генерирует внутреннее представление РИГ для каждой функции (классы **pfg_scanner**, **pfg_parser**). Также он позволяет получить файл с текстовым представлением РИГ и файл с идентификаторами дуг графа (пример выходного файла с идентификаторами дуг приведён в приложении И).

2. Парсер РИГ (класс **RigParser**) осуществляет перевод текстового представления РИГ во внутреннее. На вход принимает файл с текстовым представлением РИГ и, при наличии, файл с идентификаторами дуг графа.

3. Транслятор в dot-формат (класс **Rig2Dotter**) работает со внутренним представле-

нием РИГ, переводя его в dot-формат при записи в выходной файл. Данный файл предназначен для программы Graphviz [163], осуществляющей автоматическую визуализацию графов.

Исходный код модуля РИГ разработан А.И. Легаловым и И.В. Матовским [164, 165]. Автором внесены незначительные изменения, связанные с обработкой входных данных, позволяющие интегрировать данный модуль в систему доказательства. Расширена функциональность по работе с именами дуг графа, добавлена возможность считывания имён из файла. Также внесённые изменения позволяют создавать граф с дополнительными аргументами (см. раздел 2.4.2).

4.3 Модуль поддержки термов

В разделе 2.2 описан язык спецификации, который используется для описания свойств программ на языке Пифагор. Модуль поддержки термов обеспечивает работу с термами на языке спецификации.

4.3.1 Текстовое представление термов

Для того, чтобы пользователь мог записывать и передавать системе формулы (частный случай термов), используя обычный текстовый редактор, разработан специальный текстовый формат записи термов. Для всех математических символов введены специальные текстовые обозначения. Соответствие данных обозначений представлено в таблице 4.1.

В приложении К приведено описание синтаксиса текстового представления языка спецификации с использованием формы Бэкуса-Наура. Текстовая запись термов совпадает с математической записью, за исключением обозначений части символов и отсутствия нескольких видов сокращений: после кванторов можно писать только одну переменную, нельзя перечислять переменные через запятую в левой части операции $\in$, нельзя опускать скобки при применении функции к аргументам. При необходимости текстовая запись может быть расширена данными сокращениями.

Приведём несколько примеров:

$\neg \Big(((a, b, c \in int) \wedge (r_1 = a \cdot b)) \Rightarrow (r_1, c \in bool) \Big)$,

```
~( (a in int)/\(b in int)/\(c in int)/\(r1=a*b) => (r1 in bool)/\(c in bool) );
```

$\forall (A : \mathrm{Type})(h : A)(t : \mathrm{list}\ A).\ \neg(\mathrm{nil} = \mathrm{cons}\ h\ t)$,

```
forall A:Type. forall h:A. forall t:List(A). ~(nil = cons(h,t));
```

$(a : \mathbb{N}, b : \mathbb{Z}) \in \mathrm{datalist}\ (\mathbb{N}, \mathbb{Z})$,

```
(_ a:N, b:Z _) in datalist( (_ N, Z _) ).
```

Таблица 4.1 — Текстовое обозначение математических символов для языка спецификации

Тип символа	**Математическое обозначение**	**Текстовое обозначение**
Истина	true	`true`
Ложь	false	`false`
Отрицание	$\neg$	`~`
Конъюнкция	$\wedge$	`/\`
Дизъюнкция	$\vee$	`\/`
Импликация	$\Rightarrow$	`=>`
Эквиваленция	$\Leftrightarrow$	`<=>`
Квантор всеобщности	$\forall$	`forall`
Квантор существования	$\exists$	`exists`
Квантор существования и единственности	$\exists!$	`exists!`
Функция выбора	ε	`choice`
Умножение	$\cdot$	`*`
Равенство	$=$	`=`
Отрицание равенства	$\neq$	`!=`
Отношения	$\leqslant$, $\geqslant$	`<=`, `>=`
Отображение из одного множества в другое	$\rightarrow$	`->`
Лямбда-функция	λ	`fun`
Тип элемента	$x : T$	`x:T`
Применение функции к аргументам	$\mathrm{F}(x_1, \ldots, x_n)$	`F(x1,...,xn)`
Список	$(a_1, a_2, \ldots, a_n)$	`(_ a1, a2,..., an _)`
Принадлежность множеству	$\in$	`in`
Отрицание принадлежности ко множеству	$\notin$	`notin`
Зависимое произведение типов	Π	`dtp`
Множества чисел	$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$	`N`, `Z`, `R`

4.3.2 Внутреннее представление термов

Общая схема внутреннего представления термов приведена на рисунке 4.4. Внутреннее представление термов основано на текстовом. Во внутреннем представлении термы хранятся в виде дерева. Типы узлов дерева приведены в таблице 4.2. Каждому типу узла соответствует свой тип выражения, описываемый нетерминальным символом текстового представления. Семантически узлы (**TermNode**) можно разделить на два типа. К первому типу относятся узлы 1–5 таблицы 4.2, они соответствуют одному из базовых типов термов, описанных в разделе 2.2.2. Ко второму типу относятся узлы 6–11, называемые синтаксическими элементами. Они являются удобными синтаксическими сокращениями составных термов, построенных из базовых. Каждый из узлов второй группы может быть (автоматически) преобразован к поддереву из узлов первого типа. Синтаксические элементы введены для упрощения преоб-

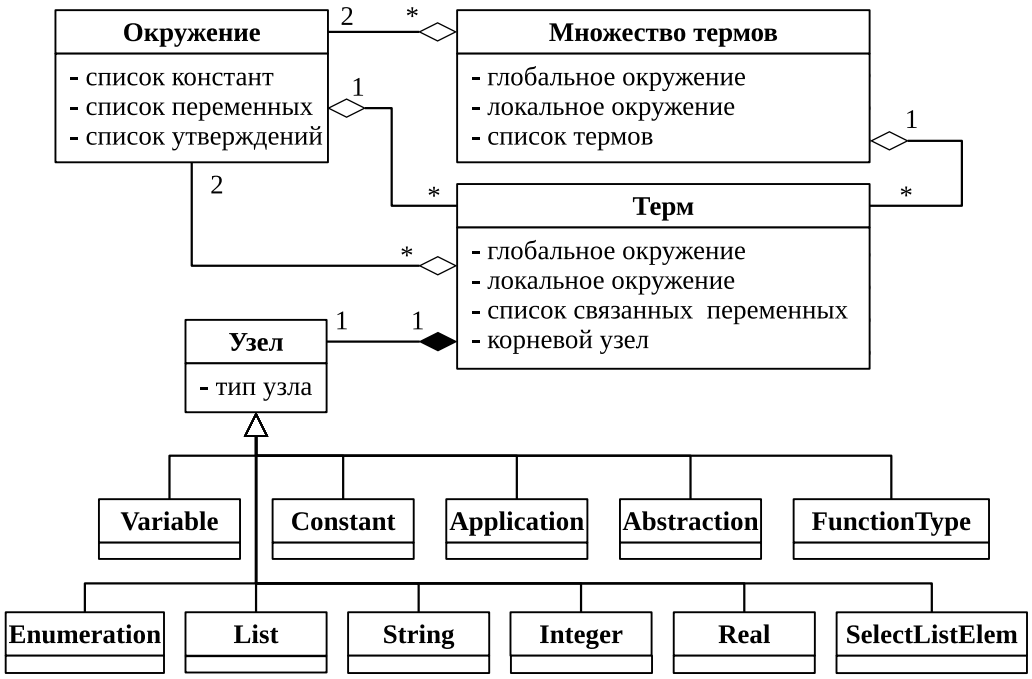


Рисунок 4.4 — Внутреннее представление терма

разований термов, связанных с заменой переменных, и не требуют дополнительного анализа поддеревьев при переводе внутреннего представления дерева в текстовое. Ещё есть два исключения — это узлы, соответствующие нетерминалам `<характеристическая функция>` и `<перечисление>`. Они не являются полноценными термами, это составные части выражения `<принадлежность множеству>`.

Приведём краткое описание структуры и свойств узлов.

1–2. Узлы **Constant** и **Variable** являются листьями дерева. Каждый узел содержит указатель на константу из таблицы локальных или глобальных констант или переменную из таблицы локальных или глобальных переменных соответственно и не имеет дочерних узлов.

3. Узел **Application** имеет несколько дочерних узлов. Первый дочерний узел является термом-функцией, остальные узлы объединены в список и являются термами-аргументами.

4. Узел **Abstraction** используется для представления трех типов термов. Он имеет тип, характеризующий тип хранимого терма, который принимает одно из следующих значений: fun, forall, exists, exists!, dtp, choice, set. Также узел содержит указатель на переменную из таблицы связанных переменных и имеет один дочерний узел — это терм, являющийся телом подкванторного выражения.

5. Узел **FunctionType** характеризует тип функции и имеет два дочерних узла. Первый соответствует терму, описывающему область определения функции, второй — терму области значения функции.

Таблица 4.2 — Типы узлов внутреннего представления дерева разбора терма

№	Имя внутреннего представления	Нетерминал текстового представления	Описание
1	**Variable**	`<переменная>`	Переменная
2	**Constant**	`<константа>`	Константа
3	**Application**	`<применение функции>`	Применение первого терма-функции к одному или нескольким термам-аргументам
4	**Abstraction**	`<функция>`, `<зависимый тип>`, `<характеристическая функция>`	Абстракция; зависимое произведение типов; множество, описываемое характеристической функцией (не является термом)
5	**FunctionType**	`<независимый тип>`	Незавсимое произведение типов
6	**Enumeration**	`<перечисление>`	Перечисление нескольких вариантов типов, которым может принадлежать элемент (не является термом)
7	**List**	`<список>`	Список
8	**String**	`<строка>`	Строка; задержанный список (как строковая константа)
9	**Integer**	`<целое число>`, `<натуральное число>`	Целое число; натуральное число
10	**Real**	`<действительное число>`	Действительное число
11	**SelectListElem**	`<выбор элемента из списка>`	Применение функции выбора элемента из списка

6. Узел **Enumeration** имеет список дочерних термов, описывающих альтернативные варианты термов-типов.

7. Узел **List** имеет тип, указывающий тип списка и принимающий три варианта значений: list, datalist, parlist. Также он содержит список пар дочерних узлов. Первый узел пары является термом, второй — типом первого терма пары.

8. Узел **String** является листом и содержит тип и строку символов. Тип может принимать два значения: 0 (строка) или 1 (задержанный список).

9–10. Узлы **Integer** и **Real** являются листьями и содержат значения числовых констант. Узел **Integer** также содержит тип, принимающий значения N или Z.

11. Узел **SelectListElem** имеет два дочерних узла. Первый описывает список, второй — номер выбираемого элемента.

Дерево разбора терма, состоящее из **Узлов**, не может самостоятельно описать терм языка спецификации (см. рисунок 4.4). Поэтому его вершина хранится в классе **Терм** (**Term**), где также содержится глобальное, локальное окружение и таблица связанных переменных.

Так как терм может содержать свободные переменные, то он может быть полностью определён только в контексте (см. раздел 2.2.2). **Окружение** (**Environment**) введено для описания глобального и локального окружения термов. Окружение включает сигнатуру — список предопределённых констант, контекст — список свободных переменных с типом, а также список утверждений, которые могут быть аксиомами, определениями и доказанными теоремами.

Глобальное окружение описывает теории языка спецификации. Оно содержит сигнатуру и аксиомы теорий, описанных в разделе 2.2.3. Глобальное окружение является общим для всех компонентов системы доказательства и загружается один раз при запуске программы. Файл с загружаемой стандартной сигнатурой приведён в приложении Л. При необходимости пользователь может изменить глобальное окружение, загружая другой файл или изменяя стандартный.

Другим глобальным объектом, связанным с глобальным окружением и используемым всеми компонентами системы, является таблица инфиксных операций. Таблица содержит имена инфиксных операций и их приоритет. Таблица необходима для отделения инфиксных операций от остальных бинарных функций. Выражения с данными бинарными операциями записываются в инфиксной форме, а наличие приоритета позволяет опускать часть скобок при текстовой записи термов. Входные данные для этой таблицы также хранятся во внешнем файле. Файл со стандартными инфиксными операциями приведён в приложении Л. При расширении глобальной сигнатуры дополнительными инфиксными операциями необходимо регистрировать их в системе, дополняя файл стандартных операций.

Локальное окружение своё у каждого терма или каждой группы термов, объединённых во множество. Локальное окружение содержит контекст, определяющий тип свободных переменных термов, также может содержать локальные константы и утверждения, являющиеся определениями и доказанными теоремами пользователя.

Таблица глобальных и локальных констант, таблица глобальных и локальных переменных из окружения, а также таблица связанных переменных терма имеют одинаковый формат: каждый элемент такой таблицы состоит из имени (идентификатора) переменной или константы и типа. При этом тип является термом, поэтому окружение имеет связь с термом. Это отражено двумя связями между **Окружением** и **Термом** на рисунке 4.4. Отсутствие циклических зависимостей гарантируется правилом построения контекста, который является упорядоченной последовательностью: любая переменная в контексте встречается только после определения своего типа. Первоначальные типы загружаются с глобальным окружением. Базовый тип Type считается предопределённым, далее загружается тип Set, потом типы множеств, имеющие тип Set, далее константы и функции над множествами и

т.д. (см. приложение Л). Особенностью термов сигнатуры глобального окружения является отсутствие свободных переменных и, значит, локального контекста (он фиктивный). Только после загрузки глобального окружения становится возможным создавать термы пользователя со свободными переменными и локальным контекстом.

Множество термов (**TermSet**) необходимо для объединения термов с одинаковыми свободными переменными. Множество имеет глобальное и локальное окружение. Локальное окружение является общим для всех термов, входящих во множество. Таким образом, термы из одного множества позволяют формулировать различные утверждения об одних и тех же объектах.

4.3.3 Трансляция термов из текстового представления во внутреннее

Трансляция текстового представления термов во внутреннее осуществляется в два этапа. На первом этапе производится лексический анализ, на втором — синтаксический.

За лексический анализ отвечает сканер (**TermScanner**). Он получает на вход последовательность символов текстового представления терма, а на выходе формирует последовательность лексем. Диаграмма разбора лексического анализатора приведена в приложении М.

Синтаксический анализ выполняет парсер (**TermParser**). На вход он принимает список лексем и «пустой» терм. В терме должно быть задано глобальное и локальное окружение, которое нужно парсеру для поиска имеющихся констант и переменных и добавления новых, в том случае, если очередной лексемой является идентификатор или символ операции.

Парсер работает на основе предпросмотра. Он получает последовательность лексем и в нужном месте осуществляет предпросмотр, чтобы выделить подтерм. Далее подтерм разбирается при рекурсивном вызове функции разбора, которой передаётся позиция начала и конца подтерма в последовательности лексем. Диаграмма Вирта синтаксического анализа приведена в приложении Н.

В трансляторе также осуществляется семантический анализ. Во-первых, он используется в нескольких случаях при распознавании типа выражения (см. приложение Н). Во-вторых, он необходим для определения контекста переменных. Тип связанных переменных указывается непосредственно в выражении, поэтому переменные заносятся в таблицу вместе со своим типом во время синтаксического анализа. Контекст для свободных переменных может быть восстановлен непосредственно по самому терму, как это описано в разделе 2.3.1. Поэтому от пользователя не требуется отдельно указывать контекст свободных переменных, достаточно указать принадлежность к нужному типу для каждой свободной переменной, используя один из вариантов описания из таблицы 2.3. Контекст свободных переменных ча-

стично формируется при синтаксическом анализе. В этом случае контекст восстанавливается только у свободных переменных, являющихся элементами списков данных или параллельных списков, так как в текстовой записи каждый элемент данных списков записывается вместе со своим типом (так же, как и связанные переменные). Поэтому подтерм, определяющий тип элемента списка, легко выделяется из выражения, распознаётся и сразу заносится в таблицу переменных.

За дальнейшее полное восстановление контекста и проверку наличия типов у каждой свободной переменной в таблице локальных переменных отвечает класс **Множество термов**. Восстановление проводится по формулам из таблицы 2.3.

4.3.4 Основные функции модуля поддержки термов

Модуль поддержки термов отвечает за разбор и хранение термов. Вся функциональность сосредоточена в классе **Множество термов**. Фактически, он является интерфейсом модуля поддержки термов. Класс позволяет:

1. осуществлять общие операции (копировать множество, взаимодействовать со сканером и парсером при получении терма в текстовой форме);

2. осуществлять операции с термами (добавлять новые термы, находить терм во множестве, удалять терм, копировать терм, импортировать (копировать или переносить) терм из другого множества);

3. осуществлять операции с переменными из локального контекста, общего для всех термов множества (проверять наличие типов у всех свободных переменных термов из множества, восстанавливать контекст, проверять наличие переменных в локальном контексте множества, генерировать имена переменных, не присутствующих во множестве);

4. формировать сообщения о последней возникшей ошибке.

4.4 Модуль поддержки ИГР

Модуль поддержки ИГР является основным в работе с информационным графом с разметкой. В нём сосредоточена основная функциональность системы, он позволяет проводить доказательства корректности программ на языке Пифагор в ручном режиме, без использования графического интерфейса.

4.4.1 Внутренне представление информационного графа с разметкой

Внутреннее представление ИГР приведено на рисунке 4.2. Класс **ИГР** (**Igr**) включает РИГ, множество формул, содержащее все формулы, размечающие дуги, и список разметок. Размер списка разметок соответствует количеству операторов в списке операторов РИГ.

Разметка (**Mark**) — класс, хранящий разметку одной дуги графа. Класс допускает компактное отображение нескольких ИГР, при котором к дуге приписано несколько формул (см. раздел 2.4.1). Каждой формуле соответствует, во-первых, её *номер* среди формул, приписанных к рассматриваемой выходной дуге оператора, и, во-вторых, вектор номеров родительских формул (*индекс*), длина которого равна количеству входных дуг оператора, без повторов. Для удобства формулы, приписанные к одной дуге, находятся не в плоском массиве, а двумерном, разбивающим формулы на группы по равенству индексов. Первая размерность массива формул называется *позицией индекса*, вторая — *позиция формулы*.

Например, на рисунке 2.5 приведена часть графа некоторой программы. Рассмотрим, какой вид примет массив формул **Разметки** узла 3. У узла 3 две входные дуги, к первой приписано три формулы (пронумерованы от 1 до 3), ко второй — две (пронумерованы от 1 до 2). Так как входных дуг две, то вектор номеров родительских формул узла 3 имеет размерность два. На первой позиции могут стоять числа от 1 до 3, а на второй — от 1 до 2, в соответствие с номерами формул входных дуг. В этом случае количество всевозможных комбинаций двумерных векторов равно $3 \cdot 2 = 6$. Поэтому количество строк в массиве формул будет равно шести. Первая строка массива соответствует индексу $(1, 1)$, вторая — $(1, 2)$, третья — $(2, 1)$ и т.д. Количество формул в каждой строке массива зависит от количества применимых аксиом и определяется особенностью программы. В примере на рисунке 2.5 в первой строке массива содержится две формулы, во второй — одна. Номера формулам присваиваются по порядку следования формул в строках. Так, формула в первой строке и в первом столбце имеет номер 1; в первой строке и во втором столбце — номер 2; во второй строке и в первом столбце — номер 3 и т.д.

4.4.2 Основные функции модуля поддержки ИГР

Все функции, поддерживаемые модулем ИГР, можно разделить на несколько групп:

1. базовые функции для работы с ИГР,
2. работа с РИГ,
3. работа с формулами разметки.

Базовые функции ИГР включают функции инициализации ИГР: создание «пустого» ИГР, загрузку и сохранение ИГР в файл, загрузку РИГ из файла, импортирование программы на языке Пифагор, задание пред- и постусловий, копирование ИГР, удаление данных ИГР. Также к этой группе относятся функции вывода информации ИГР в текстовом виде и формирования сообщений о возникших ошибках. В текстовой форме ИГР хранится в псевдо-xml формате. Пример текстового представления ИГР приведён в приложении О.

Функции работы с РИГ позволяют добавлять, редактировать и удалять глобаль-

ные, локальные константы и операторы. Имеется возможность создавать и удалять задержанные списки. При редактировании графа происходит удаление всей разметки у редактируемого узла и всех связанных с ним (по данным) узлов. Можно задавать, изменять и удалять имена дуг, при этом если узел — оператор, имеющий разметку, то имя идентификатора узла заменяется и в формуле.

К этой группе относятся функции проведения эквивалентных преобразований РИГ. Эквивалентные преобразования соответствуют правилам, указанным в таблице 2.1. Их можно применять по-отдельности или использовать функцию автоматического преобразования. Эта функция обходит все операторы РИГ и проверяет возможность применения одного из преобразований в порядке их следования в таблице 2.1. Узлы пропускаются, если они размечены или находятся в задержанном списке. Если к одному из узлов удалось применить одно из преобразований, то обход начинается с начала. Если при очередном обходе операторов РИГ ни одно преобразование не удалось применить, то функция завершается.

Функции работы с формулами разметки позволяют добавлять, изменять и удалять формулы, принимаемые модулем в текстовом виде. Разбор текстового представления формул выполняет **Множество термов**, вызывая сканер и парсер формул. Далее в списке разметок выбирается **Разметка**, номер которой совпадает с номером размечаемого оператора в РИГ, и формула передаётся в эту **Разметку**. Класс **Разметки** также обеспечивает правильное преобразование индексов при изменении количества родительских формул. Модуль ИГР позволяет проверять наличие разметки, готовность или неготовность оператора к разметке. Узел ИГР готов к разметке, если есть хотя бы по одной формуле у всех индексов его входных дуг.

Модуль содержит функцию автоматической разметки узлов, позволяющую полностью автоматически разметить все готовые операторы ИГР, не являющиеся операторами интерпретации. В самом начале работы функция проверяет наличие идентификаторов всех локальных констант в локальном окружении множества. Далее проводится автоматическая разметка всех (готовых к разметке) списков. Параллельный список размечается константой $true$, а список данных — формулой

$$idOut = (idIn_1 : typeIn_1,\ idIn_2 : typeIn_2,\ \ldots,\ idIn_n : typeIn_n),$$

где $idOut$ — идентификатор выходной дуги списка, $idIn_i$ — идентификатор i-ой входной дуги, $typeIn_i$ — тип i-ой входной дуги, $i = 1, 2, \ldots n$, n — длина списка. Далее проводится операция восстановления контекста, и в таблицу локальных переменных заносится идентификатор выходной дуги с типом datalist($typeIn_1,\ typeIn_2,\ \ldots,\ typeIn_n$).

Для поддержки автоматизированной разметки операторов интерпретации модуль име-

ет функцию свёртки, функцию согласования имён переменных теоремы с доказываемым ИГР и функцию разметки оператора интерпретации готовыми формулами.

Операция свёртки описана в разделе 2.4.2. Она необходима, в первую очередь, при разметке оператора интерпретации, так как используется при генерации условия (2.4) применимости аксиом. В реализации свёртка разделена на две части: функция свёртки формул, которая осуществляет свёртку формул указанного узла, и функция свёртки узлов, которая удаляет все независимые узлы поддерева РИГ, рассматриваемого узла.

При свёртке формул все формулы порождающего подграфа рассматриваемого узла (в случае РИГ — это поддерево с вершиной в рассматриваемом узле) объединяются конъюнкцией в новое «расширенное» предусловие, которое используется для проверки применимости аксиом. Для всех локальных констант, попавших в подграф, генерируются формулы вида $(idConst \in typeConst) \wedge (idConst = valConst)$, где $idConst$ — идентификатор константы, $typeConst$ — тип константы, $valConst$ — значение константы. Эти формулы также добавляются в «расширенное» предусловие. Если среди локальных констант присутствует задержанный список, то специальная функция находит его текстовое представление, которое используется в качестве значения $valConst$.

При свёртке узлов производится изменение графа программы: удаляются все независимые узлы (как операторы, так и локальные константы) порождающего подграфа рассматриваемого узла, а все зависимые становятся *дополнительными* входными аргументами (см. раздел 2.4.2). В результате информационный граф переходит в «нестандартное» состояние и не может быть напрямую переведён в код программы на языке Пифагор.

Таким образом, для проведения свёртки, описанной в разделе 2.4.2, необходимо вначале провести свёртку формул и получить новое предусловие, а потом выполнить свёртку узлов и изменить предусловие ИГР на новое. При этом считается, что все дополнительные аргументы также размечены предусловием (разметка производится автоматически) и не требуют дополнительной разметки. Так как ИГР находится в компактной форме, при свёртке формул возможно получение нескольких разных «расширенных» предусловий для каждого ИГР, входящего в компактную форму. Потому в результате свёртки формул и последующей свёртки узлов возможно получение нескольких ИГР, различающихся предусловием.

После получения «расширенного» предусловия, для разметки оператора интерпретации необходимо выбрать допустимые аксиомы из списка всех аксиом функции, поступающей на функциональный вход рассматриваемого оператора. Для этого используется специальная функция, которая осуществляет сравнение графа функции с шаблоном теоремы (подробнее см. раздел 4.6) и осуществляет согласование имён дуг теоремы и доказываемой функции. Например, имеем программу со следующим фрагментом:

```
...
x << (a,b);
y << x:+;
...
```

где список данных x, содержащий два элемента с идентификаторами a и b, поступает на вход оператору интерпретации вместе со встроенной функцией «+», результат заносится в y. Аксиома функции «+» имеет вид $\boxed{P(p)}$ `p:+` $\rightarrow r$ $\boxed{Q(p,r)}$. Перед использованием аксиомы производится согласование идентификаторов соответствующих дуг, заключающееся в замене переменной p на x, а r на y, одновременно с этим заменяются переменные в предусловии и постусловии, которые принимают вид $P(x)$ и $Q(x,y)$ соответственно. После проведения согласования переменных аксиома или теорема может быть использована для формирования условия применимости. Если она применима, то она используется для разметки дуги.

Функция модуля ИГР осуществляет разметку выходной дуги оператора интерпретации формулой $P \wedge Q$, принимая на вход согласованные предусловие P и постусловие Q допустимой аксиомы.

После того как ИГР полностью размечен, функция генерации условий корректности автоматически создаёт список формул на языке спецификации. Данная функция выполняет полную свёртку (см. раздел 2.4.2), и выдаёт формулы во внутреннем представлении, которое при необходимости можно преобразовать в текстовый вид. Если доказать истинность данных формул, то из этого будет следовать корректность программы.

4.5 Модуль поддержки дерева доказательства

Модуль ИГР позволяет проводить все виды преобразований исходной тройки, рассмотренные в разделе 2.4. Однако данные преобразования могут привести не только к изменению исходного графа, но и разделению (расщеплению) его на несколько новых ИГР. Для того, чтобы иметь возможность хранить всю последовательность преобразований и одновременно работать с несколькими ИГР, весь процесс доказательства представляется в виде дерева. Изначально дерево состоит из корня, которым является исходная тройка Хоара. Разметка дуг проводится на исходном узле и не требует добавления новых. В процессе разметки ИГР может перейти в компактное представление.

Дочерние узлы добавляются при проведении преобразования, которое изменяет граф программы. К таким преобразованиям относят: эквивалентные преобразования (добавляют один дочерний узел), расщепление (количество дочерних узлов зависит от количества значений, которые может принять переменная, вызвавшая расщепление), свёртка (количество дочерних узлов зависит от количества формул, приписанных к дугам порождающего подграфа, что соответствует количеству различных ИГР компактного представления).

В реализации системы доказательство представляется в виде двунаправленного дерева. На рисунке 4.5 приведена схема внутреннего представления дерева доказательства. **Дерево доказательства** (**ProofTree**) содержит вершину и список листьев. Каждая вершина имеет ИГР, родительский узел и список дочерних узлов.

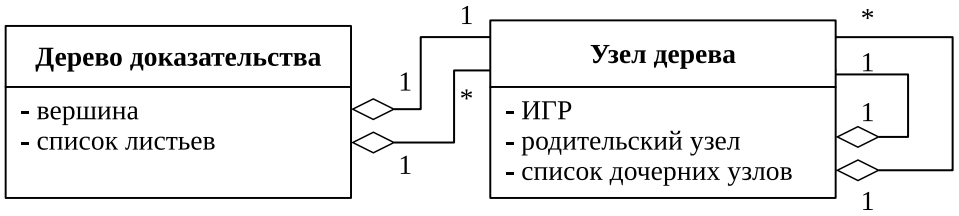


Рисунок 4.5 — Внутренне представление дерева доказательства

К основным функциям модуля относятся: загрузка дерева доказательства из файла и сохранение в файл (для текстового представление используется псевдо-xml формат, подобный текстовому представлению ИГР), удаление дерева, добавление и удаление узлов (удаление узла подразумевает удаление всего поддерева с корнем в этом узле), перенос поддерева от одного узла к другому.

4.6 Модуль поддержки библиотеки аксиом и теорем

Библиотека теорем и аксиом используется для хранения аксиом встроенных функций и теорем для функций с доказанной корректностью. Каждая функция библиотеки имеет уникальное имя, и ей может соответствовать несколько аксиом или теорем (см. раздел 2.3.2 и приложение Д).

4.6.1 Внутренне представление аксиом и теорем

В силу специфики языка Пифагор входной аргумент у функции всегда один, также этот аргумент записывается и в аксиомах языка (см. приложение Д). Однако в большинстве случаев аргумент является списком данных из нескольких элементов, которые предварительно извлекаются из него для проведения операции. Например, у одной из аксиом функции сложения целых чисел «+», приведённой на рисунке 4.6а (аксиома 5.2, приложение Д), аргумент имеет вид (a, b), а результат — сумма этих элементов. При этом предусловие аксиомы «нагружено» дополнительной информацией о структуре входного аргумента $p = (a : \mathrm{int}, b : \mathrm{int})$, которая больше не потребуется, если показать, что в доказываемой программе аргумент, к которому применяется функция, имеет нужный вид. В этом случае из предусловия аксиомы можно удалить информацию о структуре аргумента.

Если записать аксиому в «развёрнутом» виде (рисунок 4.6б), отделив информацию о

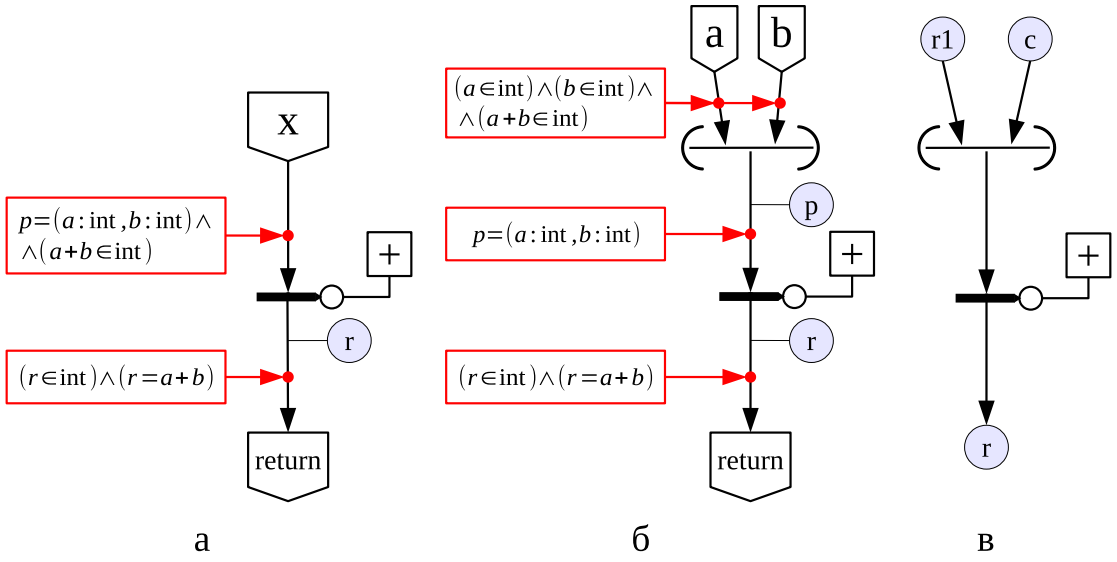


Рисунок 4.6 — Различные представления аксиомы функции «+»; а — исходная аксиома; б — «развёрнутое» представление аксиомы в виде шаблона; в — фрагмент графа функции `Func` с оператором интерпретации, принимающим список и функцию сложения

структуре входного аргумента, то её граф можно сравнивать с графом доказываемой функции. На рисунке 4.6в приведена часть графа функции `Func` (см. раздел 2.4.2), на вход оператору интерпретации в качестве аргумента поступает список из двух элементов, который совпадает по размеру со списком в аксиоме функции «+» (рисунок 4.6б). Поэтому дополнительная информация о структуре аргумента не нужна, и в качестве предусловия достаточно взять формулу $(a \in \text{int}) \wedge (b \in \text{int}) \wedge (a + b \in \text{int})$.

Таким образом, все аксиомы в библиотеке представлены в виде *шаблона* — информационного графа, несущего информацию о структуре аргумента, получаемого рассматриваемой функцией. Информационный граф шаблона имеет следующие особенности:

1. может содержать несколько аргументов;
2. имеет один оператор интерпретации, на функциональный вход которого поступает функция, которую характеризует аксиома;
3. может содержать константы (внешние, локальные и специальные значения), если в графе присутствуют локальные константы, то важен их тип, а не значение;
4. содержит списки данных, которые могут быть вложенными;
5. все дуги должны иметь имена и быть размечены, кроме выходной дуги оператора интерпретации (именем результата считается имя выходной дуги).

За сравнение аргумента оператора интерпретации доказываемой функции с шаблоном аксиомы отвечает модуль ИГР. Если обнаружено полное совпадение, то из аксиомы берётся только предусловие; если совпадения нет, то в качестве предусловия берется свёртка формул

для узла, поступающего в качестве аргумента на оператор интерпретации шаблона. При сравнении с шаблоном сразу производится согласование идентификаторов дуг.

4.6.2 Основные функции модуля поддержки библиотеки аксиом и теорем

К основным функциям модуля относятся загрузка аксиом и теорем из файла и сохранение в файл. Для текстового представление используется псевдо-xml формат, подобный текстовому представлению ИГР. Если пользователь не указывает файл, то загрузка аксиом и теорем производится из стандартного файла «axiom_theorem_base.lib». Возможно дополнить библиотеку новыми функциями, загружаемыми из другого файла.

Поиск аксиом и теорем производится по уникальному имени функции, результат запроса — список всех теорем запрашиваемой функции. Модуль позволяет добавлять в библиотеку новые пользовательские функции, при этом он принимает на вход имя функции и список ИГР, являющихся шаблонами теорем. Также есть возможность удалять функции из библиотеки по имени.

4.7 Интерфейс пользователя

Для работы с системой пользователю предоставляется графический интерфейс. Интерфейс объединяет все модули системы (см. рисунок 4.2) и позволяет строить дерево доказательства корректности программы на языке Пифагор.

Главное окно программы приведено на рисунке 4.7. Оно представляет собой редактор дерева доказательства. В левой его части располагается редактор узлов дерева, а в правой — редактор ИГР, в котором отображается текущий узел дерева. Редактор ИГР включает поля для ввода предусловия и постусловия, редактор узлов графа и область просмотра графического представления РИГ. В каждой области есть своя панель инструментов. Также все основные функции системы собраны в главном меню. Их можно разделить на следующие группы: операции считывания и записи данных в файл, операции настройки отображения содержимого окна, операции с деревом доказательства, операции с узлами и разметкой ИГР, операции с библиотекой аксиом и теорем.

Система может работать в двух режимах. Первый — «Режим ИГР», когда имеется только один ИГР, не связанный с деревом доказательства, и все преобразования проводятся на нём. Второй режим — «Режим дерева» при котором имеется дерево доказательства, и каждая операция, при которой изменяется граф программы, или производится разметка с появлением нескольких вариантов типов у одной переменной, приводит к формированию новых листьев в дереве. Изначально выбор режима работы зависит от того, создаётся (или загружается) тройка или дерево. Также применение некоторых операций в режиме ИГР, на-

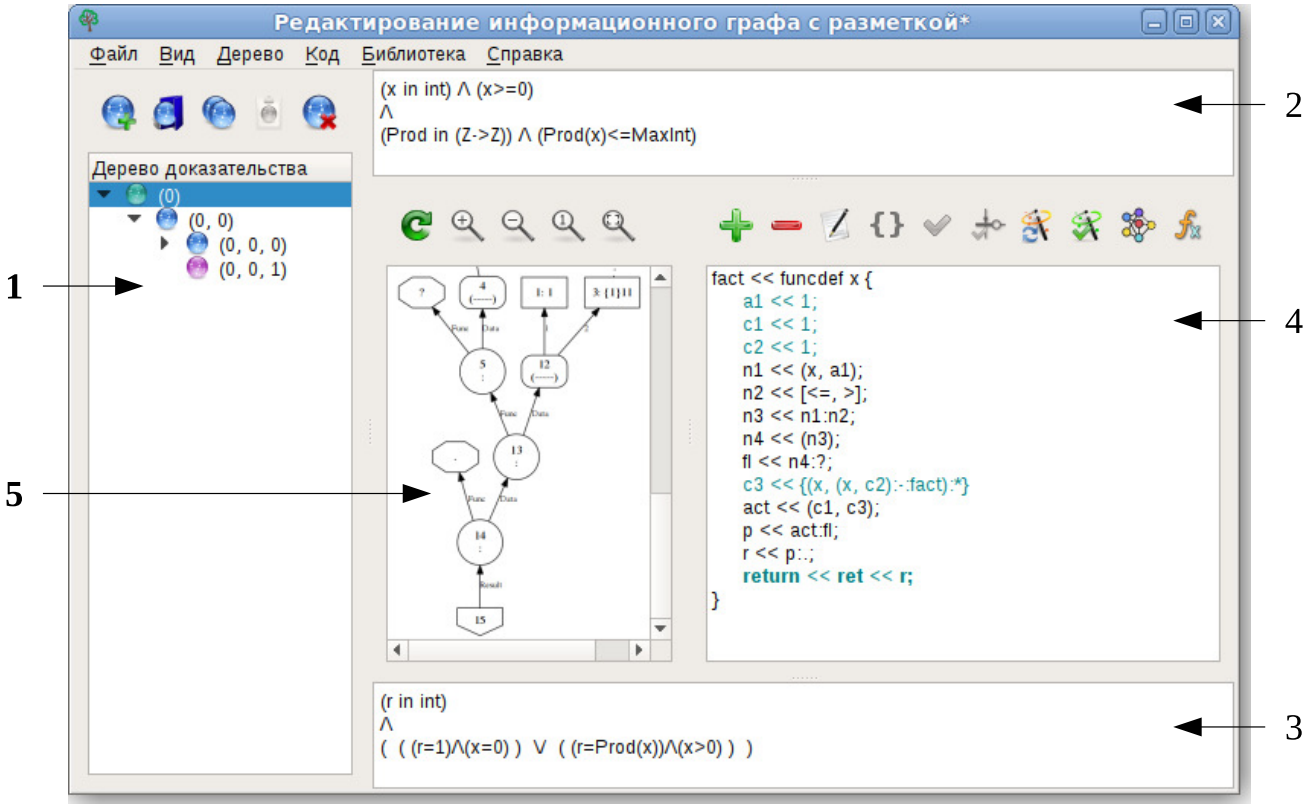


Рисунок 4.7 — Главное окно средства инструментальной поддержки формальной верификации функционально-потоковых параллельны программ; 1 — редактор узлов дерева доказательства, 2 — поле ввода предусловия, 3 — поле ввода постусловия, 4 — редактор узлов графа, 5 — графическое представление реверсивного информационного графа программы

пример, таких как создание дерева или свёртка, автоматически переводят систему во второй режим работы. При этом имеющийся ИГР встраивается в дерево доказательства как корень.

4.7.1 Редактор ИГР

Редактор ИГР в своей верхней и нижней части содержит поля для ввода соответственно предусловия и постусловия программы. В данные области необходимо записать или загрузить из файла формулы пред- и постусловия на языке спецификации в текстовой форме. Если в результате выполняемой при этом проверки в формуле обнаружена ошибка, то формула не заносится в ИГР, а пользователю выдаётся соответствующее сообщение о возможной причине ошибки.

В левой части редактора ИГР находится область просмотра графа в графическом виде. Для получения графического представления РИГ используется сторонняя программа GraphViz [163], которой передаётся файл формата dot, генерируемый модулем РИГ. Область просмотра имеет свою панель инструментов и позволяет обновить изображение, приблизить, удалить, сделать полноразмерным или уместить в области просмотра.

В правой части редактора ИГР находится редактор узлов графа, состоящий из области отображения кода программы и панели инструментов для редактирования ИГР. В области отображения кода записан код доказываемой программы, разделённый на строки. Каждая строка соответствует одному узлу РИГ, поэтому редактор кода, фактически, является редактором графа, представленного в особом текстовом виде (рисунок 4.8). Если РИГ является допустимым, то на основе его текстового представления всегда можно восстановить код программы и представить его в построчном виде.

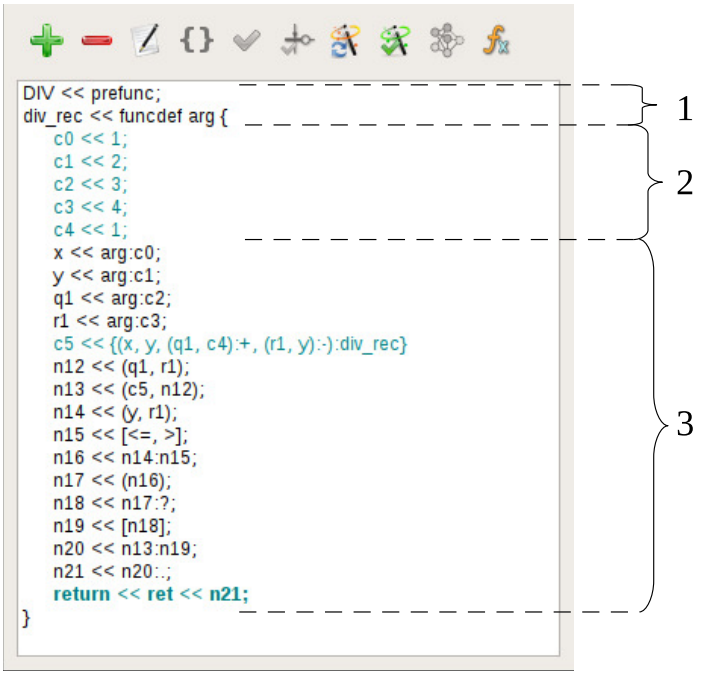


Рисунок 4.8 — Граф программы `div_rec` в виде кода, разделённого на строки; 1 — область внешних функций, 2 — область локальных констант 3 — область операторов и констант задержанных списков

Структура кода напрямую связана с текстовым представлением РИГ. В начале идёт область с внешними функциями. Заканчивается область заголовком функции, который включает имя рассматриваемой функции и входной аргумент. В начале каждой строки с внешней функцией записано её имя, потом ключевое слово `prefunc`, которое отделяется от имени символом «<<». Далее идёт область локальных констант, в которой построчно перечисляются все локальные константы, за исключением констант задержанных списков. Для каждой константы в начале записано имя (идентификатор выходной дуги), а потом значение. После следует область операторов, также расположенных построчно. На первой позиции каждой строки записан идентификатор выходной дуги оператора, далее следует сам оператор с аргументами. В качестве аргументов указываются идентификаторы входных дуг рассматриваемого оператора. Также в данной области располагаются константы задержанных списков. Константа находится на той позиции, на которой должен стоят узел, возвращающий значение из

задержанного списка. Расположение константы задержанного списка на этой позиции обусловлено тем, что узлы, попавшие в задержанный список, могут иметь незадержанные входные дуги, которые должны быть определены раньше, чем ссылки на них. В начале строки с задержанным списком также стоит идентификатор, потом задержанный список в текстовой форме (например, константа `c5` на рисунке 4.8). Последним в области операторов записано возвращаемое значение. На первой позиции стоит ключевое слово `return`, потом идентификатор оператора, возвращающего значение, потом идентификатор входной дуги (аргумента) оператора, возвращающего значение.

Для улучшения читаемости кода используется подсветка текста. Размеченные узлы подсвечены зелёным. Строки с локальными константами изначально записаны зеленым шрифтом. Строка с оператором становится зеленой (шрифт с обычного черного изменяется на жирный зеленый), если оператор полностью размечен, что означает присутствие хотя бы одной формулы для каждого родительского индекса. Если информационный граф находится в «нестандартном» состоянии, при котором имеется несколько входных аргументов, то строка с именем функции и входным аргументом подсвечивается красным цветом и имеет следующий вид:

```
имя_функции << funcdef (входной_арг, доп_арг_1, ... доп_арг_n)
```

где на первой позиции стоит имя функции, потом ключевое слово `funcdef` и в скобках перечислены аргументы, первым стоит обычный входной аргумент, далее — дополнительные.

На панели инструментов для редактирования ИГР представлены следующие функции (приведены в порядке расположения соответствующих кнопок на панели): добавление, удаление и изменение узлов ИГР, добавление/снятие задержки, ручная разметка дуги, автоматизированная разметка оператора интерпретации, автоматическое преобразование кода, автоматическая разметка кода, свёртка графа и генерация условий корректности (только для текущего ИГР).

При создании и изменении элемента открывается дополнительное диалоговое окно, своё для каждого типа узла. В данном окне пользователь указывает имя и входные аргументы для операторов или значение для константы. Нельзя добавить или удалить строку с именем функции и возвращаемое значение, эти элементы можно только редактировать. Если в графе присутствуют дополнительные аргументы, то их можно удалить, нажимая на кнопку удаления, при выделенном заголовке функции. Нельзя удалять константу задержанных списков, предварительно требуется снять задержку. Если удаляемый узел графа поступает на вход другим операторам, то пользователю выводится соответствующее предупреждение.

Диалоговое окно создания задержанного списка приведено на рисунке 4.9. По виду оно совпадает с окном создания и редактирования списка и удаления дополнительных аргу-

ментов. В правой части окна приведён список доступных узлов, в левую часть пользователь переносит узлы, которые попадут в задержанный список. На первой позиции необходимо расположить узел, возвращающий значение задержанного списка, который обязательно должен быть параллельным списком. Порядок остальных узлов не имеет значения.

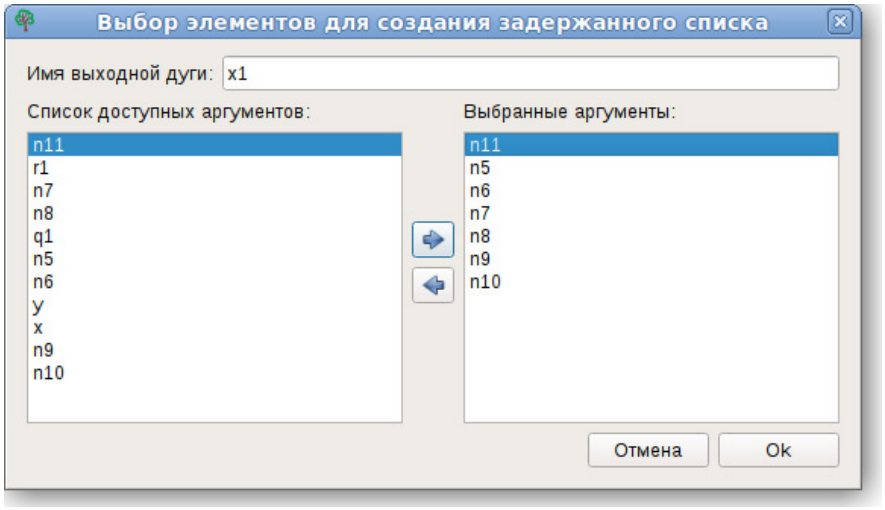


Рисунок 4.9 — Диалоговое окно создания задержанного списка

Функция добавления и удаления разметки доступна, только если выделена строка с кодом из области операторов, не являющаяся задержанным списком. При вызове данной функции производится проверка готовности дуги к разметке. Так как в редакторе используется компактное отображение получаемых ИГР, то дуга готова к разметке, если формулы присутствуют (хотя бы одна) у всех индексов всех входных дуг. Если хотя бы на одной из входной дуг оператора для любого из рассматриваемых в компактном представлении графов разметка отсутствует, то выдаётся соответствующее сообщение о невозможности разметки текущей дуги. Если оператор готов к разметке, то открывается диалоговое окно, приведённое на рисунке 4.10, это редактор ручной разметки дуг. Окно показывает разметку ИГР, представленного в компактном виде (см. раздел 4.4.1). В левой части приведены индексы родительских формул, а справа на вкладках размещаются формулы разметки текущей дуги, для выбранного индекса. Пользователь может добавлять новые формулы, что автоматически изменяет индексы дочерних узлов, непосредственно связанных с редактируемой дугой, в случае, если они уже размечены; разметка для всех размеченных, но опосредованно связанных с дугой узлов, удаляется. Проверка введённой формулы производится при переходе на другую вкладку, изменении индекса или закрытии окна, и, если обнаружена ошибка, то формула не заносится в ИГР, а пользователю выдаётся сообщение о возможной причине ошибки. При удалении формул, удаляются все формулы потомков, дочерние по отношению к текущей формуле, и изменяются индексы оставшихся формул. Если при удалении формул

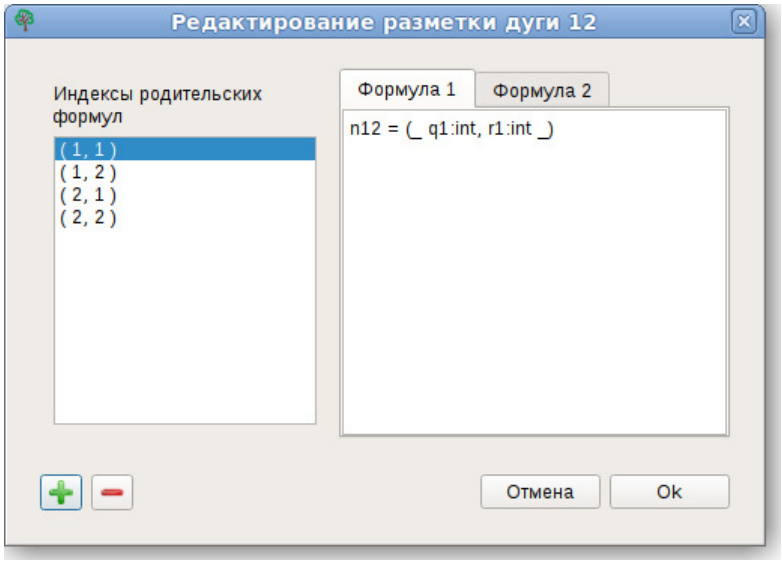


Рисунок 4.10 — Окно редактирования иерархии формул размечаемой дуги

узел становится неполностью размеченным, то удаляются все формулы его дочерних узлов.

Функция автоматизированной разметки оператора интерпретации доступна, только если выделен оператор интерпретации. Если оператор готов к разметке, то в библиотеке аксиом и теорем производится поиск теоремы для функции, поступающей на функциональный вход оператора. Поиск производится автоматически для встроенных и внешних функций, локальных констант целого и булевского типа (соответствуют функции выбора элемента из списка и функции «селектора», см. приложение Д, аксиомы 4.1–4.5, 21.1–21.2). Если на функциональный вход поступает выходная дуга оператора, то система не может автоматически определить применяемую функцию и просит пользователя выбрать нужную функцию из библиотеки. Если при поиске в библиотеке запрашиваемая функция не найдена, то процесс разметки прекращается, и пользователю выдаётся соответствующее сообщение об ошибке. Если функция присутствует в библиотеке, то с помощью модуля ИГР производится сравнение шаблонов аксиом и согласование имён входных дуг с рассматриваемым ИГР. Далее открывается диалоговое окно, приведённое на рисунке 4.11, в котором записано «расширенное» предусловие, полученное с помощью свёртки формул для рассматриваемого оператора интерпретации, и список предусловий найденных теорем. Пользователю необходимо выбрать применимые теоремы и передать их номера системе. Для этого пользователь должен проверить выполнимость условия применимости (2.4). При необходимости формулы можно экспортировать в файл в текстовом виде. После того как система получает номера применимых теорем, производится автоматическая разметка выходной дуги оператора интерпретации формулами $P_i \wedge Q_i$, где P_i — согласованное предусловие , а Q_i — согласованное постусловие i-ой выбранной теоремы, $i = 1, 2, \ldots, n$, n — число выбранных теорем. Так

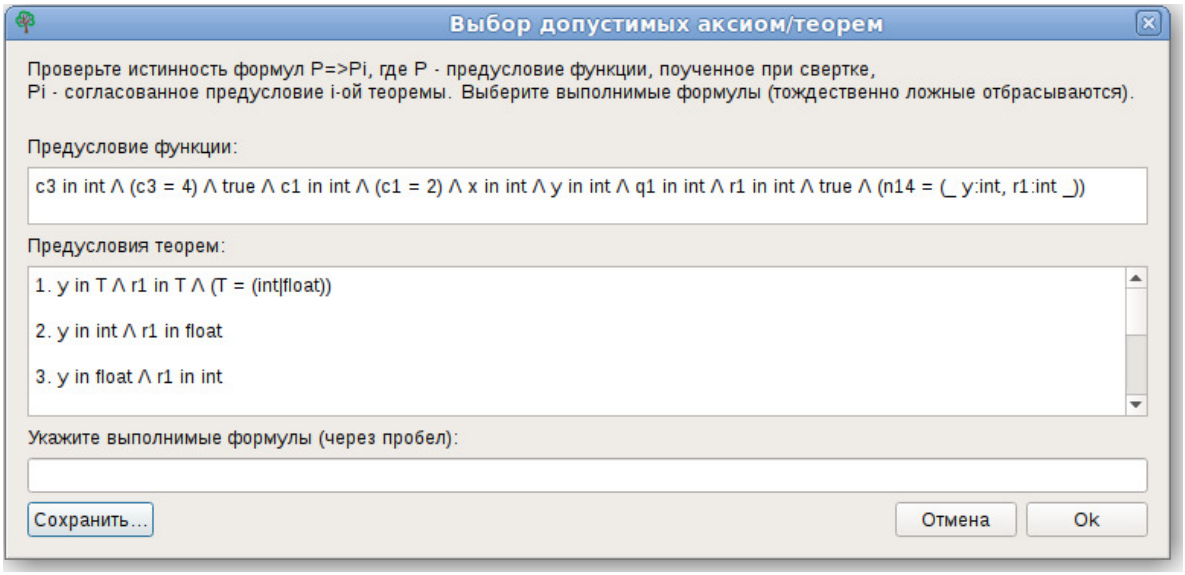


Рисунок 4.11 — Окно выбора применимых теорем

как ИГР находится в компактном виде, то в результате свёртки формул может получиться несколько «расширенных» предусловий, тогда свой набор применимых теорем необходимо выбрать для каждого полученного предусловия.

Функция автоматического преобразования кода использует модуль ИГР, который и осуществляет преобразования. Результат работы данной функции зависит от режима работы системы. Если система находится в режиме ИГР, то в результате преобразования изменяется текущий ИГР. Если система находится в режиме дерева, то преобразованный ИГР добавляется как дочерний к текущему узлу дерева. При этом, если текущий узел не является листом дерева доказательства, то его дочернее поддерево предварительно удаляется. В преобразованиях участвуют только готовые узлы без разметки.

Функция автоматической разметки кода также обращается к модулю ИГР для проведения разметки.

Функция свёртки доступна только в режиме дерева. При проведении свёртки, возможно получение несколько новых ИГР, все они добавляются как дочерние к текущему узлу дерева. Полученные ИГР могут иметь несколько дополнительных аргументов.

Функция генерации условий корректности позволяет применить к ИГР полную свёртку и автоматически сгенерировать формулы, если граф полностью размечен. При успешной генерации открывается диалоговое окно со списком полученных формул, при необходимости формулы можно сохранить в файл в текстовом виде. Пользователю остаётся самостоятельно (вручную или с помощью верификатора формул) доказать тождественную истинность данных формул, чтобы показать, что доказываемая программа корректна.

4.7.2 Редактор узлов дерева доказательства

Редактор узлов дерева доказательства приведён на рисунке 4.7. Он включает область просмотра дерева доказательства и панель инструментов для редактирования узлов дерева.

В области просмотра дерево представлено в виде древовидного списка (аналогичные списки используется для просмотра структуры каталогов). Каждая строка соответствует одному узлу дерева. Строка содержит пиктограмму и список номеров всех узлов в пути от корня дерева к текущему узлу. Тип узла отражает цвет пиктограммы: зелёный соответствует корню дерева, фиолетовый — листу дерева, остальные узлы имеют синюю пиктограмму.

На панели инструментов редактирования дерева представлены следующие функции (приведены в порядке расположения кнопок на панели): добавление нового «пустого» ИГР (как листа к текущему), загрузка существующего ИГР из файла (как листа к текущему), копирование текущего узла (без его поддерева), вставка скопированного узла (как листа к текущему), удаление выделенного узла со всем поддеревом.

4.7.3 Управление библиотекой аксиом и теорем

Функции работы с библиотекой собраны в главном меню в разделе «Библиотека». Функции позволяют загрузить и сохранить библиотеку в файл, посмотреть все теоремы для определённой функции из библиотеки, добавить и удалить теоремы для выбранной функции.

При загрузке библиотеки из файла возможно заменить текущую библиотеку на новую или дополнить новыми теоремами. Во втором случае, если имя загружаемой функции совпадёт с именем присутствующей в библиотеки функции, то загрузка прекратится, а пользователю будет выдано сообщение об ошибке.

Для просмотра всех теорем одной из функций библиотеки пользователь должен указать имя интересующей функции, после этого в редактор загружается список теорем в виде дерева доказательства. Вершина дерева — «пустой» ИГР (может быть произвольным) с именем выбранной функции. Каждый потомок вершины — одна из теорем рассматриваемой функции. Число потомков совпадает с числом теорем функции. Каждая теорема представлена в виде шаблона (см. раздел 4.6.1). Кроме пред- и постусловия ИГР может содержать дополнительные аргументы и иметь разметку у некоторых дуг. В качестве примера на рисунке 4.12 приведены загруженные аксиомы функции «+». В редакторе узлов дерева отображено общее количество аксиом, при выборе одного из узлов в редакторе ИГР отображается соответствующая аксиома. ИГР аксиомы, приведённой на рисунке, имеет два входных аргумента и находится в «нестандартном» состоянии, об этом информирует красная подсветка заголовка функции. На рисунке поверх главного окна расположено окно редактирования

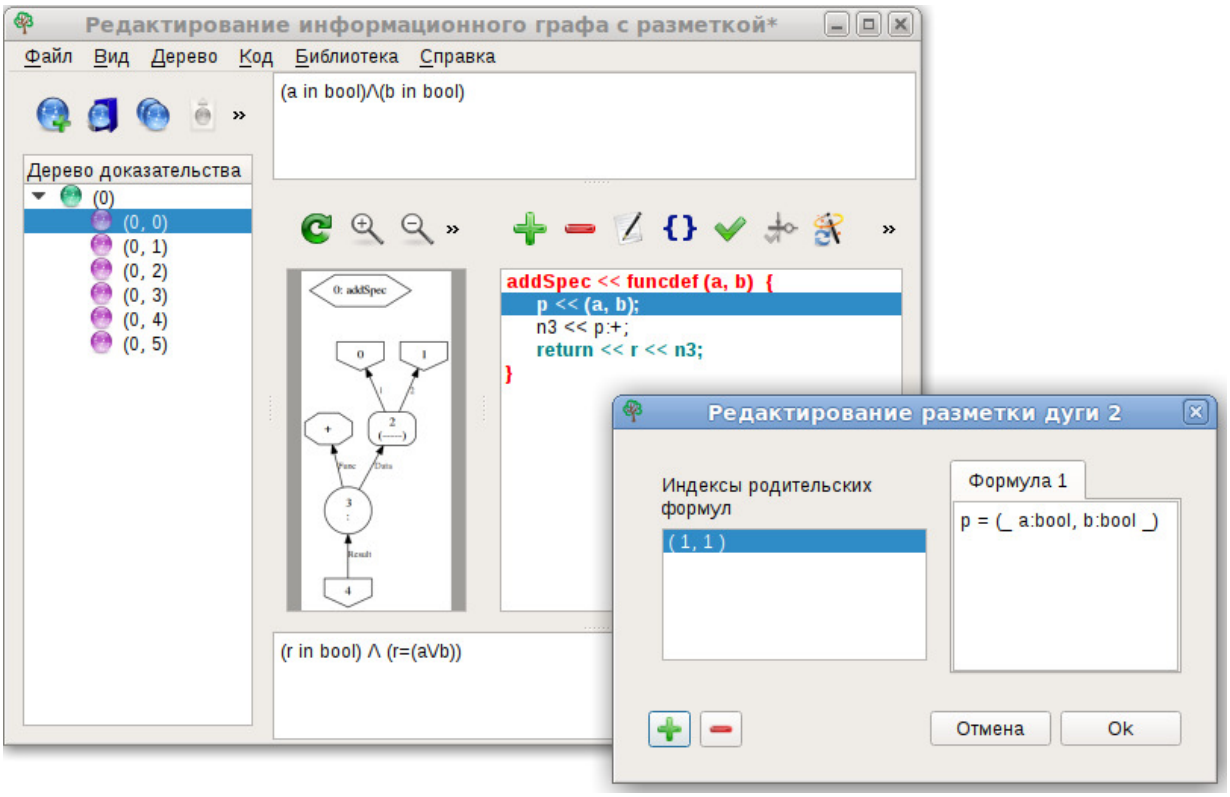


Рисунок 4.12 — Аксиома функции «+», загруженная из библиотеки

формул для дуги p, в котором отображена формула размечающая эту дугу.

Если пользователю требуется добавить в библиотеку теоремы новой функции, то он должен построить дерево доказательства, аналогичное рассмотренному выше. Вершина дерева должна иметь имя добавляемой функции. Каждый её дочерний узел — одна из теорем, представленная в виде шаблона.

Выводы

Для облегчения процесса формальной верификации программ на языке Пифагор разработана архитектура системы поддержки данного процесса. Разработан язык для текстового представления термов языка спецификации программ. Реализован прототип интегрированной системы, включающей в себя:

1. модуль поддержки РИГ, модифицированный автором и интегрированный в систему;

2. модуль поддержки термов, содержащий транслятор термов на языке спецификации из текстового представления во внутреннее;

3. модуль ИГР, отвечающий за внутренне представление ИГР и осуществляющий основные операции по созданию и преобразованию графа, автоматизированной разметке дуг графа;

4. модуль поддержки дерева доказательства, позволяющий сохранять последовательность шагов при доказательстве корректности программы;

5. библиотеку аксиом и теорем, используемую для разметки выходных дуг операторов интерпретации;

6. графическую оболочку, предназначенную для взаимодействия с пользователем и осуществляющую управление взаимодействием между остальными модулями системы.

Предложенное инструментальное средство может использоваться при доказательстве корректности произвольной функции на языке Пифагор. Предварительно необходимо преобразовать рассматриваемую функцию в прямую рекурсию, что требует описания спецификации для множества связанных с ней функций. Для доказательства завершения функции дополнительно требуется задать ограничивающую функцию и дополнить спецификацию рассматриваемой функции условием уменьшения значения ограничивающей функции при рекурсивных вызовах. Дальнейшее доказательство корректности функции проводится в системе в интерактивном режиме. Если корректность функции успешно доказана, то она может быть добавлена в библиотеку аксиом и теорем.

Заключение

Работа посвящена повышению надёжности и корректности функционально-потоковых параллельных программ посредством разработки для них методов формальной верификации. Полученные результаты имеют значение для развития теоретических основ программирования, включающих языки программирования, технологии программирования, автоматизацию программирования, теоретические основы системного и прикладного программного обеспечения.

Получены следующие научные и практические результаты.

1. В результате исследования применимости существующих методов формальной верификации к доказательству корректности ФПП программ, разработан метод верификации ФПП программ на языке Пифагор, основанный на исчислении Хоара, и по сложности сравнимый с методами доказательства корректности для последовательных программ. Для разработки метода потребовалось формализовать семантику языка Пифагор, разработать язык спецификации свойств программ и создать аксиоматическую теорию на базе исчисления Хоара.

2. Предложен метод доказательства завершения программ на языке Пифагор. Метод позволяет изменить спецификацию программы таким образом, что доказательство частичной корректности на базе исчисления Хоара одновременно доказывает и завершение программы.

3. Предложен метод удаления взаимной рекурсии нескольких функций. Метод позволяет преобразовывать любую функцию ФПП программы в функцию с прямой рекурсией. Это упрощает процесс доказательства тем, что позволяет работать только с одной функцией при доказательстве корректности методом, основанным на исчислении Хоара.

4. Разработана архитектура инструментального средства поддержки доказательства корректности ФПП программ. Разработан прототип инструментального средства, позволяющий доказывать корректность программ на языке Пифагор. Данная система позволяет упростить доказательство за счёт визуализации процесса и автоматизации построения дерева доказательства.

Перспективы развития направления исследований. Полученные результаты позволяют проводить формальную верификацию ФПП программ. Из верифицированных программ в перспективе можно сформировать библиотеку неограниченно параллельных программ и, при необходимости, преобразовывать в программы для различных реальных архитектур, которые также будут корректны, если доказать корректность правил преобразования.

Список литературы

1. Непомнящий, В.А. Прикладные методы верификации программ / В.А. Непомнящий, О.М. Рякин; под ред. А.П. Ершова. — М.: Радио и связь, 1988. — 256 с.

2. Карпов, Ю.Г. Model Cheking. Верификация параллельных и распределенных программных систем / Ю.Г. Карпов. — СПб.: БХВ-Петербург, 2010. — 560 с.

3. Beckert, B. Reasoning and Verification / B. Beckert, R. Hahnle // IEEE Intelligent Systems. — 2014. — Vol. 29, N. 1. — P. 20–29.

4. Кларк, М. Верификация моделей программ: Model Checking: пер. с англ. / М. Кларк; под ред. Р. Смелянского. — М.: МЦНМО, 2002. — 416 с.

5. Denney, E. A Theory of Program Refinement: Thesis for the Degree of Doctor of Philosophy / E. Denney. — Edinburgh: University of Edinburgh, 1998. — 244 p.

6. Morgan, C. Programming from Specifications. Second edition / C. Morgan. — Hemel Hempstead: Prentice Hall International, 1994. — 332 с.

7. Barnett, M. Boogie: A modular reusable verifier for object-oriented programs / M. Barnett, B.E. Chang, R. DeLine, B. Jacobs, K.R.M. Leino // Formal Methods for Components and Objects: 4th International Symposium. FMCO 2005. LNCS. — 2006. — Vol. 4111. — P. 364–387.

8. Непомнящий, В.А. Верификация С-программ в мультиязыковой системе СПЕКТР / В.А. Непомнящий, И.С. Ануреев, М.М. Атучин, И.В. Марьясов, А.А. Петров, А.В. Промский // Моделирование и анализ информационных систем. — 2010. — № 17 (4). — С. 88–100.

9. Van den Berg, J. The LOOP compiler for Java and JML / J. Van den Berg, B. Jacobs // Tools and Algorithms for the Construction and Analysis of Systems. LNCS. — 2001. — Vol. 2031. — P. 299–312.

10. Ahrendt, W. The KeY Tool / W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P.H. Schmitt // Software and System Modeling. — 2005. — Vol. 4 (1). — P. 32–54.

11. Rondon, P. Liquid types /P. Rondon, M. Kawaguchi, R. Jhala // SIGPLAN Not. — 2008. — Vol. 43, N. 6. — P. 159–169.

12. Vazou, N. Refinement Types for Haskell / N. Vazou, E.L. Seidel, R. Jhala, D. Vytiniotis, S. Peyton-Jones // SIGPLAN Not. — 2014. — Vol. 49, N. 9. — P. 269–282.

13. Легалов, А.И. Статически типизированная версия языка функционально-потокового параллельного программирования / А.И. Легалов, И.А. Легалов, И.В. Матковский // Параллельные вычислительные технологии — XIV международная конференция, ПаВТ'2020, г. Пермь, 31 марта – 2 апреля 2020 г. Короткие статьи и описания плакатов. — Челябинск: Издательский центр ЮУрГУ, 2020. — С. 185–192.

14. Левин, И.И. Подход к архитектурно-независимому программированию вычислительных систем на основе аспектно-ориентированного языка Set@l / И.И. Левин, А.И. Дордопуло, И.В. Писаренко, А.К. Мельников // — 2018. — № 3 (197). — С. 46–58.

15. Levin, I.I. Approach to Automatic Development of Parallel Applications for Reconfigurable Computer Systems / I.I. Levin, A.I. Dordopulo // Параллельные вычислительные технологии — XIV международная конференция, ПаВТ'2020, г. Пермь, 31 марта – 2 апреля 2020 г. Короткие статьи и описания плакатов. — Челябинск: Издательский центр ЮУрГУ, 2020. — С. 83–93.

16. Легалов, А.И. Функциональный язык для создания архитектурно-независимых параллельных программ / А.И. Легалов // Вычислительные технологии. — 2005. — № 1 (10). — С. 71–89.

17. Легалов, А.И. Функциональная модель параллельных вычислений и язык программирования «Пифагор» / А.И. Легалов, Ф.А. Казаков, Д.А. Кузьмин, Д.В. Привалихин. — Режим доступа: http://www.softcraft.ru/fppp.shtml

18. Удалова, Ю.В. Методы отладки и верификации функционально-потоковых параллельных программ / Ю.В. Удалова, А.И. Легалов, Н.Ю. Сиротинина // Журнал Сибирского федерального университета. Серия: Техника и технологии. — 2011. — Т. 4. № 2. — С. 213-224.

19. Удалова, Ю.В. Отладка и верификация функционально-потоковых параллельных программ: дис. . . . канд. техн. наук: 05.13.11: защищена 21.05.15 / Ю.В. Удалова. — Красноярск, 2015. — 170 с.

20. Васильева, К.А. Верификация автоматных программ с использованием LTL / К.А. Васильева, Е.В. Кузьмин // Моделирование и анализ информационных систем. — 2007. — Т. 14, № 1. — С. 3-14.

21. Parallel Debugging Using Microsoft Visual Studio 2005. — USA: Microsoft Corporation, 2006. — Режим доступа: http://go.microsoft.com/fwlink/?LinkId=55932

22. Кулямин, В.В. Методы верификации программного обеспечения / В.В. Кулямин. — М: Институт системного программирования Российской академии наук, 2008. — 111 с.

23. Meyer, B. Applying Design by Contract / B. Meyer // Computer. — 1992. — Vol. 25, N. 10. — P. 40–51.

24. Мейер, Б. Введение в Контрактное Проектирование / Б. Мейер // Открытые системы. —1998. — № 6.

25. Meyer, B. Object-Oriented Software Construction: 2nd Edition / B. Meyer. — New Jersey: Prentice Hall, 1997. — 1254 p.

26. Сынтульский, С. Методы формально-логической спецификации / С. Сынтульский. — СПб.: Санкт-Петербургский государственный университет, 2007. — Режим доступа: www.math.spbu.ru/user/soloviev/ТВПС ФЛС.pdf

27. Rondon, P. Liquid Types: Thesis for the Degree of Doctor of Philosophy / P. Rondon. — San Diego: University of California, 2012. — 265 p.

28. Floyd, R.W. Assigning meaning to programs / R.W. Floyd // Proc. of Symposium in Applied Mathematics, Mathematical Aspects of Computer Science. — 1967. — N. 19. — P. 19-32.

29. Hoare, C. A. R. An axiomatic basis for computer programming / C. A. R. Hoare // Communications of the ACM. — 1969. — Vol. 10, N. 12. — P. 576-585.

30. Колмогоров, А.Н. Введение в математическую логику: науч. изд. / А.Н. Колмогоров, А.Г. Драгалин. — М.: Изд-во Моск. ун-та, 1982. — 120 с.

31. Ершов, Ю.Л. Математическая логика: учебн. пособие для вузов. — 2-е изд., испр. и доп. / Ю.Л. Ершов, Е.А. Палютин. — М.: Наука. Гл. ред. физ.-мат. лит., 1987. — 336 с.

32. Карри, Х. Основания математической логики: пер.с англ. / Х. Карри. — М.: Мир, 1969. — 568 с.

33. Church, A. A Formulation of the Simple Theory of Types / A. Church // Journal of Symbolic Logic. — 1940. — Vol. 5, N. 2. — P. 56–68.

34. Barendregt, H. Lambda Calculi with Types / H. Barendregt // Handbook of Logic in Computer Science. — 1992. — Vol. 2. — P. 118–309.

35. Барендрегт, Х. Ламбда-исчисление. Его синтаксис и семантика: пер.с англ. / Х. Барендрегт. — М.: Мир, 1985. — 606 с.

36. Лисков, Б. Использование абстракций и спецификаций при разработке программ: пер.с англ. / Б. Лисков, Дж. Гатэг. — М.: Мир, 1989. — 424 с.

37. Столл, Р. Множества. Логика. Аксиоматические теории / Р. Столл; под ред. Ю.А. Шихановича. — М.: Просвещение, 1968. — 232 с.

38. Дудаков, С.М. Математическое введение в информатику: учеб. пособие / С.М. Дудаков. — Тверь: Твер. гос. ун-т, 2003. — 221 с.

39. Coquand, T. The Calculus of Constructions / T. Coquand, G. Huet // Information and Computation — 1988. — Vol. 76, N. 2–3. — P. 95–120.

40. Martin-Lof, P. Intuitionistic Type Theory. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980 / P. Martin-Lof // Journal of Symbolic Logic. — 1984. — Vol. 51, N. 4. — P. 1075–1076.

41. Bertot, Y. Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions / Y. Bertot, P. Casteran. — Springer, 2004. — 472 p.

42. Norell, U. Towards a Practical Programming Language Based on Dependent Type Theory: Thesis for the Degree of Doctor of Philosophy / U. Norell. — Goteborg: Chalmers University of Technology, 2007. — 166 p.

43. Denney, E. Refinement Types for Specification / E. Denney // Programming Concepts and Methods PROCOMET'98. IFIP — The International Federation for Information Processing. — Boston: Springer, 1998. — P. 148–166.

44. Luo, Z. Program Specification and Data Refinement in Type Theory / Z. Luo // Lecture Notes in Computer Science. — 1991. — Vol. 493. — P. 143–168.

45. Luo, Z. The Extended Calculus of Constructions: Thesis for the Degree of Doctor of Philosophy / Z. Luo. — Edinburgh: University of Edinburgh, 1990. — 138 p.

46. Дейкстра, Э. Дисциплина программирования / Э. Дейкстра. — М.: Мир, 1978. — 275 с.

47. Грис, Д. Наука программирования: пер. с англ. / Д. Грис. — М.: Мир, 1984. — 416 с.

48. Back, R.J.R. A Calculus of Refinements for Program Derivations / R.J.R. Back // Acta Informatica. — 1988. — Vol. 25. — P. 593–624.

49. Freeman, T. Refinement types for ML / T. Freeman, F. Pfenning // SIGPLAN Not. — 1991. — Vol. 26, N. 6. — P. 268–277.

50. Turing, A.M. On Computable Numbers, with an Application to the Entscheidungsproblem / A.M. Turing // Proceedings of the London Mathematical Society. — 1937. — Vol. 42. — P. 230–265.

51. Кушниренко, А.Г. Программирование для математиков / А.Г. Кушниренко, Г.В. Лебедев. — М.: Наука. Гл. ред. физ.-мат. лит., 1988. — 384 с.

52. Роганов, Е.А. Основы информатики и программирования: учебное пособие / Е.А. Роганов. — М.: МГИУ, 2001. — 315 с.

53. Мальцев, А.И. Алгоритмы и рекурсивные функции / А.И. Мальцев. — М.: Наука. Гл. ред. физ.-мат. лит., 1986. — 368 с.

54. Баррон, Д. Рекурсивные методы в программировании / Д. Баррон. — М.: Мир, 1974. — 80 с.

55. Boyer, R.S. A computational logic / R.S. Boyer, J.S. Moore. — New York: Academic Press, 1979. — 420 p.

56. Giesl, J. Automated Termination Proofs with Measure Functions / J. Giesl // Advances in Artificial Intelligence. KI 1995. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence). — 1995. — Vol. 981. — P. 149–160.

57. Hoare, C. A. R. Procedures and parameters: An axiomatic approach / C. A. R. Hoare // Lecture Notes in Mathematics. — 1971. — Vol. 188. — P. 102–116.

58. Walther, C. On Proving the Termination of Algorithm by Machine / C. Walther // Artificial Intelligence. — 1994. — Vol. 71, N. 1. — P. 101–157.

59. Giesl, J. Termination Analysis for Functional Programs Using Term Orderings / J. Giesl // In Proc. SAS'95. LNCS. — 1995. — Vol. 983 — P.154–171.

60. Jones, N.D. Termination Analysis of the Untyped λ-Calculus / N.D. Jones, N. Bohr // Rewriting Techniques and Applications. RTA 2004. Lecture Notes in Computer Science. — 2004. — Vol. 3091. — P. 1–23.

61. Sereni, D. Termination Analysis of Higher-Order Functional Programs / D. Sereni, N.D. Jones // Programming Languages and Systems. APLAS 2005. Lecture Notes in Computer Science. — 2005. — Vol 3780. — P 281–297.

62. Barthe, G. Type-based Termination of Recursive definitions / G. Barthe, M.J. Frade, E. Gimenez, L. Pinto, T. Uustalu // Mathematical Structures in Computer Science. — 2004. — Vol. 14. — P. 97–141.

63. Hughes, J. Proving the Correctness of Reactive Systems Using Sized Types / J. Hughes, L. Pareto, A. Sabry // Conference Record of the Annual ACM Symposium on Principles of Programming Languages. — 1996. — N. 1. — P. 410–423.

64. Xi, H. Dependent Types for Program Termination Verification / H. Xi // Higher-Order and Symbolic Computation. — 2002. — Vol. 15. — P. 91–131.

65. Nielson, F. Termination Analysis based on Operational Semantics: Technical Report / F. Nielson, H.R. Nielson. — Denmark: Aarhus University, 1995. — 42 p.

66. Manoury, P. Automatizing Termination Proofs of Recursively Defined Functions / P. Manoury, M. Simonot // Theoretical Computer Science. — 1994. — Vol. 135. — P. 319–343.

67. Giesl, J. Termination Analysis for Functional Programs / J. Giesl, C. Walther, J. Brauburger // Automated Deduction — A Basis for Applications. Applied Logic Series. — 1998. — Vol. 10. — P. 135–164.

68. Arts, T. Termination of term rewriting using dependency pairs / T. Arts, J. Giesl // Theoretical Computer Science. — 2000. — Vol. 236. — P. 133–178.

69. Marche, C. The Termination Competition / C. Marche, H. Zantema // Term Rewriting and Applications. RTA 2007. Lecture Notes in Computer Science. — 2007. — Vol. 4533. — P. 303–313.

70. Giesl, J. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages / J. Giesl, S. Swiderski, P. Schneider-Kamp, R. Thiemann // Term Rewriting and Applications. RTA 2006. Lecture Notes in Computer Science. — 2006. — Vol. 4098. — P. 297–312.

71. Giesl, J. Automated termination proofs for Haskell by term rewriting / J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, R. Thiemann // ACM Transactions on Programming Languages and Systems (TOPLAS). — 2011. — Vol. 33, N. 2. — P. 7:1–7:39.

72. Giesl, J. Proving termination of programs automatically with AProVE 2014 / J. Giesl, M. Brockschmidt, F. Emmes et al. // International Joint Conference on Automated Reasoning 2014. Automated Reasoning. — 2014. – Vol. 8562. — P. 184–191.

73. Хювёнен Э., Сеппянен Й. Мир Лиспа. Том 1. Введение в язык Лисп и функциональное программирование: пер. с англ. / Э. Хювёнен, Й. Сеппянен. — М: Мир, 1990.

74. Milner, R. The Definition of Standard ML / R. Milner, M. Tofte, R. Harper. — London: MIT Press, 1990. — 99 p.

75. Augustsson, L. Cayenne — A Language with Dependent Types / L. Augustsson // Lecture Notes in Computer Science. — 1998. — Vol. 1608. — P. 240–267.

76. Damas, L. Principal Type-Schemes for Functional Programs / L. Damas, R. Milner // Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — New Mexico: Association for Computing Machinery, 1982. — P. 207–212.

77. Agerwala, T. Assertion Graphs for Verifying and Synthesizing Programs / T. Agerwala, J. Misra. — Austin: University of Texas, 1978. — 20 p.

78. Ou, X. Dynamic Typing with Dependent Types / X. Ou, G. Tan, Y. Mandelbaum, D. Walker // Exploring New Frontiers of Theoretical Informatics. IFIP International Federation for Information Processing. — 2004. — Vol. 155. — P. 437–450.

79. Vazou, N. Abstract Refinement Types / N. Vazou, P.M. Rondon, R. Jhala // Felleisen M., Gardner P. (eds) Programming Languages and Systems. ESOP 2013. Lecture Notes in Computer Science. — 2013. — Vol. 7792. — P. 209–228

80. Rondon, P. CSolve. Verifying C with Liquid Types / P. Rondon, A. Bakst, M. Kawaguchi, R. Jhala // Lecture Notes in Computer Science. — 2012. — Vol. 7358. — P. 744–750.

81. Vytiniotis, D. Equality Proofs and Deferred Type Errors: A Compiler Pearl / D. Vytiniotis, S.P. Jones, J.P. Magalhães // SIGPLAN Not. — 2012. — Vol. 47, N. 9. — P. 341–352.

82. Meyer, B. Eiffel: The Language / B. Meyer. — Englewood Cliffs: Prentice Hall, 1991.

83. Tschannen, J. Automatic verification of Eiffel programs: Master's Thesis / J. Tschannen. — Zurich: ETH, 2009. — 81 p.

84. Tschannen, J. Verifying Eiffel Programs with Boogie / J. Tschannen, C.A. Furia, M. Nordio, B. Meyer // CoRR. — 2011. — Vol. abs/1106.4700.

85. Eiffel. Documentation. — Режим доступа: https://www.eiffel.org/documentation

86. Meyer, B. Systematic Concurrent Object-Oriented Programming / B. Meyer // Commun. ACM. — 1993. — Vol. 36, N. 9. — P. 56–80.

87. Morandi, B. Prototyping a Concurrency Model / B. Morandi, M. Schill, S. Nanz, B. Meyer // Proceedings of the 2013 13th International Conference on Application of Concurrency to System Design. — 2013. — P. 170–179.

88. Meyer, B. On the Verification of SCOOP Programs / B. Meyer, G. Caltais // Science of Computer Programming. — 2017. — Vol. 133, Part 2. — P. 194–215

89. DeLine, R. BoogiePL: A typed procedural language for checking object-oriented programs: Technical Report MSR-TR-2005-70 / R. DeLine, K.R.M. Leino. — Microsoft Research, 2005. — 12 p.

90. Rustan, K. This is Boogie 2: Technical report / K. Rustan, M. Leino. — Redmond: Microsoft Research, 2008. — 52 p.

91. Detlefs, D. Simplify: a Theorem Prover for Program Checking / D. Detlefs, G. Nelson, J.B. Saxe // ACM. — 2005. — Vol. 52 (3). — P. 365–473.

92. The Java Modeling Language (JML). JML Home Page. — Режим доступа: http://www.eecs.ucf.edu/ leavens/JML//index.shtml

93. Larch Home Page. — Режим доступа: http://www.sds.lcs.mit.edu/spd/larch/

94. Hatcliff, J. Behavioral Interface Specification Languages / J. Hatcliff, G. Leavens, K.R.M. Leino, P. Müller, M. Parkinson // ACM Comput. Surv. — 2012. — Vol. 44, N. 3. — 68 p.

95. Leavens, G.T. Preliminary Design of JML: A Behavioral Interface Specification Language for Java / G.T. Leavens, A.L. Baker, C. Ruby // SIGSOFT Softw. Eng. Notes. — 2006. — Vol. 31, N. 3. — P. 1–38.

96. Leavens, G.T. Design by Contract with JML (2005) Draft / G.T. Leavens, Y. Cheon. — Режим доступа: http://www.jmlspecs.org/jmldbc.pdf

97. Chalin, P. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2 / P. Chalin, J.R. Kiniry, G.T. Leavens, E. Poll // Lecture Notes in Computer Science. — 2006. — Vol. 4111. — P. 342–363.

98. Leavens, G.T. JML Reference Manual (DRAFT), May, 2013 / G.T. Leavens, E. Poll, C. Clifton et al.— Режим доступа: http://www.jmlspecs.org/refman/jmlrefman.pdf

99. Ahrendt, W. Deductive Software Verification — The KeY Book: From Theory to Practice / W. Ahrendt, B. Beckert, R. Bubel et al. // Lecture Notes in Computer Science. — 2016. — Vol. 10001. — 734 p.

100. The KeY Progect. — Режим доступа: https://www.key-project.org/

101. Krakatoa and Jessie: verification tools for Java and C programs. — Режим доступа: http://krakatoa.lri.fr/

102. Filliatre, JC. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification / JC. Filliatre, C. Marche// Lecture Notes in Computer Science. — 2007. — Vol. 4590. — P. 173–177.

103. Jacobs, B. Java Program Verification at Nijmegen: Developments and Perspective / B. Jacobs, E. Poll // Lecture Notes in Computer Science. — 2004. — Vol. 3233. — P. 134–153.

104. Cok, D.R. Reasoning with Specifications Containing Method Calls and Model Fields / D.R. Cok // Journal of Object Technology. — 2005. — Vol. 4. — P. 77–103.

105. OpenJML. — Режим доступа: http://www.openjml.org/

106. Filliatre, J.C. Multi-prover Verification of C Programs / J.C. Filliatre, C. Marche // Formal Methods and Software Engineering. ICFEM 2004. Lecture Notes in Computer Science. — 2004. — Vol. 3308. — P. 15–29.

107. Alkassar, E. The Verisoft Approach to Systems Verification / E. Alkassar, M.A. Hillebrand, D. Leinenbach, N.W. Schirmer, A. Starostin // Verified Software: Theories, Tools, Experiments. VSTTE 2008. Lecture Notes in Computer Science. — 2008. — Vol. 5295. — P. 209–224.

108. Why3. Where Programs Meet Provers. — Режим доступа: http://why3.lri.fr/

109. Cohen, E. VCC. A Practical System for Verifying Concurrent C / E. Cohen, M. Dahlweid et al. // Theorem Proving in Higher Order Logics. TPHOLs 2009. Lecture Notes in Computer Science. — 2009. — Vol. 5674. — P. 23–42.

110. Анурeев, И.С. Верификация С-программ на основе смешанной аксиоматической семантики / И.С. Анурeев, И.В. Марьясов, В.А. Непомнящий // Моделирование и анализ информационных систем. — 2010 — Т. 17, № 3. — С. 5–28.

111. Шелехов, В.И. Исчисление вычислимых предикатов: Препр. / В.И. Шелехов. — Новосибирск: ИСИ СО РАН, 2007; № 143. — 24 с.

112. Карнаухов, Н.С. Язык предикатного программирования Р. Версия 0.14 / Н.С. Карнаухов, Д.Ю. Першин, В.И. Шелехов. — Новосибирск, 2018. — 41 с.

113. Чушкин, М.С. Система дедуктивной верификации предикатных программ / М.С. Чушкин // Программная инженерия. — 2016. — Т. 7, № 5. — С. 202–210.

114. Шелехов, В.И. Предикатное программирование: учеб. пособие / В.И. Шелехов. — Новосибирск: Новосиб. гос. ун-т., 2009. — 109 с.

115. Чушкин, М.С. Генерация и доказательство условий корректности предикатных программ / М.С. Чушкин, В.И. Шелехов. — Новосибирск: ИСИ, 2012. — 50 с.

116. SMT-LIB. The Satisfability Modulo Theories Library. — Режим доступа: http://smtlib.cs.uiowa.edu/

117. Cok, D.R. The SMT-LIBv2 Language and Tools. A Tutorial: Technical report / D.R. Cok. — GrammaTech, Inc., 2011. — 71 p.

118. CVC3. — Режим доступа: https://cs.nyu.edu/acsys/cvc3/

119. CVC4. The SMT Solver. — Режим доступа: http://cvc4.cs.stanford.edu/web/

120. Z3. — Режим доступа: https://github.com/Z3Prover/z3/wiki

121. Yices. — Режим доступа: http://yices.csl.sri.com/

122. Tuerk, T. Interactive Theorem Proving (ITP) Course / T. Tuerk. — Режим доступа: https://www.kth.se/social/group/interactive-theorem-/page/lecture-slides-51/

123. The Seventeen Provers of the World / ed. F. Wiedijk // LNCS. — 2016. — Vol. 3600. — 162 p.

124. HOL. — Режим доступа: https://hol-theorem-prover.org/

125. Gordon, M.J.C. Introduction to HOL. A theorem proving environment for higher order logic / M.J.C. Gordon, T.F. Melham. — Cambridge: Cambridge University Press, 1993. — 471 p.

126. Isabelle. — Режим доступа: http://isabelle.in.tum.de/

127. Nipkow, T. Isabelle/HOL: A Proof Assistant for Higher-Order Logic / T. Nipkow, L.C. Paulson, M. Wenzel // LNCS. — 2002. — Vol. 2283. — 226 p.

128. Coq. — Режим доступа: http://coq.inria.fr/

129. PVS. — Режим доступа: http://pvs.csl.sri.com/index.shtml

130. Crow, J. A Tutorial Introduction to PVS / J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas // Workshop on Industrial-Strength Formal Specification Techniques. — 1995. — 116 p. — Режим доступа: http://www.csl.sri.com/papers/wift-tutorial/

131. HOL Light. — Режим доступа: http://www.cl.cam.ac.uk/ jrh13/hol-light/

132. ProofPower. — Режим доступа: http://www.lemma-one.com/ProofPower/doc/doc.html

133. HOL Zero. — Режим доступа: http://proof-technologies.com/holzero/index.html

134. Owre, S. A tutorial on using PVS for hardware verification / S. Owre, J.M. Rushby, N. Shankar, M.K. Srivas // Theorem Provers in Circuit Design. TPCD 1994. Lecture Notes in Computer Science. — 1995. — Vol. 901. — P. 258–279.

135. Shankar, N. The PVS Proof Checker: A Reference Manual / N. Shankar, S. Owre, J.M. Rushby. — Menlo Park: SRI International, Computer Science Laboratory, 1993. — 119 p.

136. Шилов, Н.В. Основы синтаксиса, семантики, трансляции и верификации программ / Н.В. Шилов. — Новосибирск: Издательство СО РАН, 2009. — 340 с.

137. Strachey, C. Fundamental Concepts in Programming Languages / C. Strachey // Higher-Order and Symbolic Computation. — 2000. — Vol. 13. — P. 11–49.

138. Васильев, В.С. Трансформация функционально-потоковых параллельных программ в императивные / В.С. Васильев, А.И. Легалов, А.И. Зыков // Моделирование и анализ информационных систем. — 2021. — Т. 28, № 2. — С. 198–214.

139. Непомнящий, О.В. Метод архитектурно-независимого высокоуровневого синтеза СБИС / О.В. Непомнящий, И.Н. Рыженко, А.И. Легалов // Известия ЮФУ. Технические науки. — 2018. — № 8 (202). — С. 38–47.

140. Легалов, А.И. На пути к переносимым параллельным программам / А.И. Легалов, Д.А. Кузьмин, Ф.А. Казаков, Д.В. Привалихин // Открытые системы. — 2003. — № 5. — С. 36–42.

141. Легалов, А.И. Особенности семантики статически типизированного языка функционально-потокового параллельного программирования / А.И. Легалов, И.А. Легалов, И.В. Матковский // Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции (23–28 сентября 2019 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2019. — С. 489–500.

142. Нечаев, В.И. Числовые системы. Пособие для студентов пед. ин-тов / В.И. Нечаев. — М.: «Просвещение», 1975. — 199 с.

143. Kreisel, G. Elements of Mathematical Logic. Studies in Logic and the Foundations of Mathematics / G. Kreisel, J.L. Krivine. — Amsterdam: North-Holland Publishing Company, 1967. — 221 p.

144. Manna, Z. Combining decision procerures / Z. Manna, C. Zarba // LNCS. — 2003. — Vol. 2787. — P. 453–468.

145. Enderton, H.B. A Mathematical Introduction to Logic / H.B. Enderton. — 2nd ed. — Massachusetts: Harcourt/Academic Press, 2001. — 326 p.

146. Скобелев, В.В. Автоматы на алгебраических структурах. Модели и методы их исследования / В.В. Скобелев. — Донецк: ИПММ НАН Украины, 2013. — 307 с.

147. Oppen, D.C. Reasoning about recursively defined data structures / D.C. Oppen // Journal of the ACM. — 1980. — Vol. 27, N. 3. — P. 403–411.

148. Melham, T. Automating recursive type definitions in higher order logic / T. Melham // Current Trends in Hardware Verification and Automated Theorem Proving. — New York: Springer, 1989. — P. 341–386.

149. Легалов, А.И. Использование асинхронно поступающих данных в потоковой модели вычислений / А.И. Легалов // Третья сибирская школа-семинар по параллельным вычислениям. — Томск: Изд-во Томского ун-та, 2006. — С 113–120.

150. Чень, Л. Математическая логика и автоматическое доказательство теорем / Л. Чень; под ред. С.Ю. Маслова. — М: Наука, 1983. — 358 с.

151. Harrison, J. Handbook Of Practical Logic And Automated Reasoning / J. Harrison. — New York: Cambridge University Press, 2009. — 781 p.

152. Головешкин, В.А. Теория рекурсии для программистов / В.А. Головешкин, М.В. Ульянов. — М.: Физматлит, 2006. — 296 с.

153. Plumer, L. Termination Proofs for Logic Programs / L. Plumer // Lecture Notes in Artificial Intelligence. — 1990. — Vol. 446. — 142 p.

154. Giesl, J. Termination of nested and mutually recursive algorithms / J. Giesl // Automat. Reason. — 1997. — Vol. 19. — P. 1–29.

155. Bevers, E. Proving Termination of Conditional Rewrite Systems / E. Bevers, J. Lewi // Acta Informatica. — 1993. — Vol. 30. — P. 537–568.

156. Удалова, Ю.В. Библиотека математических функций для языка функционально-потокового параллельного программирования Пифагор / Ю.В. Удалова // Вестник Бурятского государственного университета. Математика, информатика. — 2019. — № 4. — С. 57–64.

157. Удалова, Ю.В. Библиотека обработки строк для языка функционально-потокового параллельного программирования Пифагор / Ю.В. Удалова // Международный научно-исследовательский журнал. — 2020. — № 12-1 (102). — С. 83–87.

158. Буч, Г. Язык UML. Руководство пользователя. 2-е изд.: пер. с англ. Мухин Н. / Г. Буч, Д. Рамбо, И. Якобсон. — М.: ДМК Пресс, 2006. — 496 с.

159. Легалов, А.И. Событийная модель вычислений, поддерживающая выполнение функционально-потоковых параллельных программ / А.И. Легалов, Г.В. Савченко, В.С. Васильев // Системы. Методы. Технологии. — 2012. — № 1 (13). — С. 113–119.

160. Матковский, И.В. Параллельная событийная машина для функционально-потокового языка «Пифагор» / И.В. Матковский // Информационные и математические технологии

в науке и управлении: Сборник трудов XVVII Байкальской Всероссийской конференции с международным участием (Иркутск – Байкал, 30 июня – 9 июля 2012 г.). Часть II. — Иркутск: ИСЭМ СО РАН, 2012. — С. 186–193.

161. Матковский, И.В. Инструментальная поддержка трансляции и выполнения функционально-потоковых параллельных программ / И.В. Матковский, А.И. Легалов // Ползуновский вестник. — 2013. — № 2. — С. 49–52.

162. Матковский, И.В. Транслятор для функционально-потоковых параллельных программ / И.В. Матковский // Языки программирования и компиляторы — 2017: труды конференции / Южный федеральный университет; под ред. Д.В. Дуброва. — Ростов-на-Дону: Издательство Южного федерального университета, 2017. — С. 194–197.

163. Graphviz — Graph Visualization Software. — Режим доступа: http://graphviz.org/

164. Легалов, А.И. Свидетельство о государственной регистрации программы для ЭВМ № 2013611618 Российская Федерация. Модуль формирования реверсивного информационного графа / А.И. Легалов, О.В. Непомнящий, Н.Ю. Сиротинина, И.В. Матковский; заявитель и правообладатель Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Сибирский федеральный университет» (СФУ). — № 2012660518; заявл. 03.12.2012; опубл 29.01.2013. — 1 с.

165. Матковский, И.В. Свидетельство о государственной регистрации программы для ЭВМ RU 2018666577 Российская Федерация. Программа для трансляции функционально-потокового языка параллельного программирования / И.В. Матковский, А.И. Легалов, В.С. Васильев, А.И. Постников; заявитель и правообладатель Федеральное государственное автономное образовательное учреждение высшего образования «Сибирский федеральный университет» (СФУ). — № 2018663780; заявл. 03.12.2018; опубл 18.12.2018. — 1 с.

Список публикаций по теме диссертации

166. Legalov, A.I. A Toolkit For The Development Of Data-Driven Functional Parallel Programmes / A.I. Legalov, V.S. Vasilyev, I.V. Matkovskii, M.S. Ushakova // Communications in Computer and Information Science. — 2018. — Vol. 910. — P. 16–30.

167. Легалов, А.И. Особенности разработки и преобразования функционально-потоковых параллельных программ / А.И. Легалов, М.С. Ушакова // Суперкомпьютерные дни в России: труды международной конференции (Москва, 24-25 сентября 2018 г.). Суперкомпьютерный консорциум университетов России, Российская академия наук. — М.: Московский государственный университет имени М.В. Ломоносова, 2018. — С. 999–1000.

168. Удалова, Ю.В. Об отладке и верификации программ на функционально-потоковом параллельном языке Пифагор / Ю.В. Удалова, М.С. Кропачева (Ушакова) // Материалы V всероссийской научно-технической конференции «Молодёжь и наука: начало XXI века». — Красноярск, 2009. — С. 40–43.

169. Кропачева (Ушакова), М.С. Верификация функционально-потоковых параллельных программ методом поиска слабейшего предусловия / М.С. Кропачева (Ушакова) // Материалы XI Всероссийской научно-практической конференции «Проблемы информатизации региона» (ПИР-2009). — Красноярск: РИЦ СибГТУ, 2009. — С. 136–139.

170. Кропачева (Ушакова), М.С. Формальная верификация параллельных программ / М.С. Кропачева (Ушакова) // Труды XII Международной научной конференции «Интеллект и наука». — Красноярск: Центр информатизации, 2012. — С. 130–131.

171. Ушакова, М.С. Использование SMT-решателей для доказательства условий корректности программ на языке Пифагор / М.С. Ушакова // Проспект Свободный — 2018: материалы Международной студенческой конференции (Красноярск, 23-27 апреля 2018 г.). — Красноярск: СФУ, 2018. — С. 847–850.

172. Ушакова, М.С. Инструментальная поддержка формальной верификации программ, написанных на языке функционально-потокового параллельного программирования / М.С. Ушакова, А.И. Легалов // Вестник Южно-Уральского государственного университета. Серия Вычислительная математика и информатика. — 2015. — Т. 4, № 2. — С. 58–68.

173. Кропачева (Ушакова), М.С. Формализация семантики функционально-потокового языка параллельного программирования Пифагор / М.С. Кропачева (Ушакова) // Материалы XII Всероссийской научно-практической конференции «Проблемы информатизации региона» (ПИР-2011). — Красноярск: Сибирский федеральный университет, 2011. — С. 144–148.

174. Ушакова, М.С. Семантика типов данных функционально-потокового языка параллельного программирования Пифагор / М.С. Ушакова // Образовательные ресурсы и технологии. — 2016. — № 2 (14). — С. 263–269.

175. Легалов, А.И. Динамически изменяющийся параллелизм с асинхронно-последовательными потоками данных / А.И. Легалов, И.В. Матковский, М.С. Ушакова, Д.С. Романова // Моделирование и анализ информационных систем. — 2020. — Т. 27, № 2. — С. 164–179. — Перевод изд.: Dynamically Changing Parallelism with Asynchronous Sequential Data Flows / A.I. Legalov, I.V. Matkovskii, M.S. Ushakova, D.S. Romanova // Automatic Control and Computer Sciences. — 2021. — Vol. 55, N. 7. — P. 636–646.

176. Кропачева (Ушакова), М.С. Формальная верификация программ, написанных на функционально-потоковом языке параллельного программирования / М.С. Кропачева (Ушакова), А.И. Легалов // Моделирование и анализ информационных систем. — 2012. — Т. 19, № 5. — С. 81–99. — Перевод изд.: Formal Verification of Programs in the Functional Data-Flow Parallel Language / M.S. Kropacheva (Ushakova), A.I. Legalov // Automatic Control and Computer Sciences. — 2013. — Vol. 47, N. 7. — P. 373–384.

177. Кропачева (Ушакова), М.С. Аксиоматический подход к формальной верификации рекурсивных программ на функционально-потоковом языке параллельного программирования / М.С. Кропачева (Ушакова) // Параллельные вычислительные технологии (ПаВТ'2013): труды международной научной конференции (г. Челябинск, 1–5 апреля 2013 г.). — Челябинск: Издательский центр ЮУрГУ, 2013. —С. 421–431.

178. Kropacheva (Ushakova), M.S. Formal Verification of Programs in the Pifagor Language / M.S. Kropacheva (Ushakova), A.I. Legalov // Parallel Computing Technologies (PaCT-2013) 12th International Conference, September 30 – October 4, 2013. Saint-Petersburg, Russia. LNCS. — 2013. — Vol. 7979. — P. 80–89.

179. Ushakova, M.S. Automation of Formal Verification of Programs in the Pifagor Language / M.S. Ushakova, A.I. Legalov // Modeling and Analysis of Information Systems. — 2015. — Vol. 22, N. 4. — P. 578–589.

180. Удалова, Ю.В. Верификация и доказательство завершения функционально-потоковых параллельных программ / Ю.В. Удалова, М.С. Ушакова // Языки программирования и компиляторы — 2017: труды конференции / Южный федеральный университет; под ред. Д.В. Дуброва. — Ростов-на-Дону: Изд-во ЮФУ, 2017. — С. 248–251.

181. Кропачева (Ушакова), М.С. Формальная верификация параллельных программ / М.С. Кропачева (Ушакова) // Исследования наукограда. — 2012. — № 2. — С. 35–38.

182. Легалов, А.И. Преобразование хвостовых рекурсий в функционально-потоковых параллельных программах / А.И. Легалов, О.В. Непомнящий, И.В. Матковский, М.С. Кропачева (Ушакова) // Моделирование и анализ информационных систем. — 2012. — Т. 19, № 5. — С. 48–58. — Перевод изд.: Tail Recursion Transformation in Functional Dataflow Parallel Programs / A.I. Legalov, O.V. Nepomnyaschy, I.V. Matkovsky, M.S. Kropacheva (Ushakova) // Automatic Control and Computer Sciences. — 2013. — Vol. 47, N. 7. — P. 366–372.

183. Ушакова, М.С. Верификация программ со взаимной рекурсией на языке Пифагор / М.С. Ушакова, А.И. Легалов // Моделирование и анализ информационных систем. — 2018. — Т. 25, № 4. — С. 358–381. — Перевод изд.: Verification Of Programs With Mutual

Recursion In Pifagor Language / M.S. Ushakova, A.I. Legalov // Automatic Control and Computer Sciences. — 2018. — Vol. 52, N. 7. — P. 850–866.

184. Кропачева (Ушакова), М.С. Система автоматизированного доказательства корректности функционально-потоковых параллельных программ / М.С. Кропачева (Ушакова) // Молодёжь и наука: сборник материалов VIII Всероссийской научно-технической конференции. — Красноярск: Сиб. федер. ун-т., 2012. — С. 62–65.

185. Кропачева (Ушакова), М.С. Общая концепция и архитектура программного средства инструментальной поддержки методов формальной верификации функционально-потоковых параллельных программ / М.С. Кропачева (Ушакова) // Многоядерные процессоры, параллельное программирование, ПЛИС, системы обработки сигналов: сб. ст. — Барнаул: Барнаул, 2013. — С. 113–116.

186. Ушакова, М.С. Свидетельство о государственной регистрации программы для ЭВМ № 2021663755 Российская Федерация. Инструментальное средство для формальной верификации функционально-потоковых параллельных программ на языке Пифагор на основе исчисления Хоара / Ушакова М.С.; заявитель и правообладатель Федеральное государственное автономное образовательное учреждение высшего образования «Сибирский федеральный университет» (СФУ). — № 2021662716; заявл. 09.08.2021; опубл. 23.08.2021. — 1 с.

187. Легалов, А.И. Инструментальная поддержка создания и трансформации функционально-потоковых параллельных программ / А.И. Легалов, В.С. Васильев, И.В. Матковский, М.С. Ушакова // Труды Института системного программирования РАН. — 2017. — Т. 29, № 5. — С. 165–184.

188. Легалов, А.И. Свидетельство о государственной регистрации программы для ЭВМ № 2013611434 Российская Федерация. Синтаксический анализатор текстового представления реверсивного информационного графа / А.И. Легалов, О.В. Непомнящий, И.В. Матковский, М.С. Кропачева (Ушакова); заявитель и правообладатель Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Сибирский федеральный университет» (СФУ). — опубл. 09.01.2013. — 1 с.

Список сокращений

АПМ	—	анализатор программных моделей;
БНФ	—	Бэкуса-Наура форма;
ИГР	—	информационный граф с разметкой;
ПМ	—	программная модель;
ПО	—	программное обеспечение;
РИГ	—	реверсивный информационный граф;
УК	—	условия корректности;
ФПП	—	функционально-потоковый параллельный;
ФПМПВ	—	функционально-потоковая модель параллельных вычислений

Приложение А

Акты о внедрении

СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ | SIBERIAN FEDERAL UNIVERSITY

МИНОБРНАУКИ РОССИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Сибирский федеральный университет»
660041, Красноярский край,
г. Красноярск, проспект Свободный, д. 79
телефон: (391) 244-82-13, тел./факс: (391) 244-86-25
http://www.sfu-kras.ru, e-mail: office@sfu-kras.ru
ОКПО 02067876; ОГРН 1022402137460;
ИНН/КПП 2463011853/246301001

УТВЕРЖДАЮ
Ректор ФГАОУ ВО СФУ
_____ М. В. Румянцев
«12» _____ 2021 г.

_____ № _____
на № _____ от _____

АКТ о внедрении

в учебный процесс в ФГАОУ ВО Сибирском федеральном университете результатов кандидатской диссертационной работы Ушаковой Марии Сергеевны «Методы и инструментальные средства формальной верификации функционально-потоковых параллельных программ»

Настоящим подтверждаем, что результаты диссертационного исследования Ушаковой М.С. на тему «Методы и инструментальные средства формальной верификации функционально-потоковых параллельных программ», включающие метод верификации на базе исчисления Хоара, метод доказательства завершения и метод удаления взаимной рекурсии нескольких функций для функционально-потоковых параллельных программ, и инструментальное средство, обеспечивающее поддержку формальной верификации функционально-потоковых параллельных программ (свидетельство о государственной регистрации программы для ЭВМ №2021663755) обладают актуальностью, предоставляют научный и практический интерес и внедрены в учебный процесс на кафедре вычислительной техники и кафедре высокопроизводительных вычислений Института космических и информационных технологий СФУ. Эти материалы используются при подготовке магистров по направлению 09.04.01 – «Информатика и вычислительная техника», при изучении дисциплин «Технологии разработки программного обеспечения», «Параллельное программирование», «Высокопроизводительные вычисления на графических процессах» и при выполнении выпускных квалификационных работ.

И. о. директора института космических и информационных технологий СФУ, к.т.н. Д. В. Капулин

Заведующий кафедрой вычислительной техники, к.т.н. О. В. Непомнящий

Заведующий кафедрой высокопроизводительных вычислений, к.т.н. Д. А. Кузьмин

Акционерное общество
«ИНФОРМАЦИОННЫЕ СПУТНИКОВЫЕ СИСТЕМЫ»
имени академика М.Ф. Решетнёва»

РЕШЕТНЕВ
АО «И С С»

ул. Ленина, д. 52, г. Железногорск, ЗАТО Железногорск, Красноярский край, Российская Федерация, 662972
Тел. (3919) 76-40-02, 72-24-39, Факс (3919) 72-26-35, 75-61-46, e-mail: office@iss-reshetnev.ru, http: //www.iss-reshetnev.ru
ОГРН 1082452000290, ИНН 2452034898

УТВЕРЖДАЮ
Заместитель генерального конструктора
по электрическому проектированию и
системам управления КА
_____ С.Г. Кочура
«03» _____ 2022 г.
М.П.

АКТ
внедрения результатов диссертационных исследований

Настоящим актом подтверждается, что результаты диссертационных исследований по теме: Методы и инструментальные средства формальной верификации функционально-потоковых параллельных программ
(тема диссертации)

выполненной Ушаковой Марией Сергеевной,
(Ф.И.О Аспиранта/исследователя - автора)

использованы в: Акционерное Общество «Информационные спутниковые системы» имени академика М. Ф. Решетнёва».
(Полное наименования предприятия)

1. Вид внедренных результатов: Комплекты сложно-функциональных блоков СБИС в составе приборов гражданского назначения: 751ВМ.2513-0 БУ БРК, 765.1512-0-01 БУ БКУ.
2. Форма внедрения: Экспериментальное внедрение при выполнении НИР и СЧ ОКР
3. Новизна результатов научно-исследовательских работ:
качественно новые
(принципиально новые, качественно новые, модификация)

4. Опытно-промышленная проверка в ОКР/СЧ ОКР/НИР вн. №№ 20717, 20718.
5. Научно-технический эффект: Предположительное сокращение сроков выполнения работ, сокращение собственного энергопотребления, сокращение требуемой площади кристалла за счет переносимости полученного кода и возможности рассмотрения модельного ряда для вариантов полученных решений.

Заместитель начальника отдела
проектирования и испытаний РЭА _____ И.Н. Тульский

Приложение Б

Таблица Б.1 — Семантика встроенных функций языка Пифагор

№	Выражение	Значения выражения	Результат
p:.			
1	true	true	**p** (пустая операция, аргумент не изменяется)
.:f			
1	true	true	**f()** (вызов функции f без аргумента)
p:type			
1	true	true	**signal**, если p – сигнал **int**, если p – целое число **float**, если p – число с плавающей точкой **char**, если p – символ **bool**, если p – true или false **func**, если p – предопределенная или пользоват функция **error**, если p – ошибка **datalist**, если p – список данных **user_type**, где user_type – имя пользовательского типа, а p – имеет этот тип **TYPEERROR**, если p элемент множества {int, float, .. , user_type }, то есть p – идентификатор типа
p:|			
1	p:type	datalist	**<int, n>**, где $p=(p_1,..,p_n)$
		else	**BASEFUNCERROR**
p:<int, b>			
1	p:type	datalist	→ 2
		else	**BASEFUNCERROR**
2	b равно нулю или выражение (b,0):=	да или значение выражения true	**.** (пустое значение)
		else	→ 3

№	Выражение	Значения выражения	Результат
3	абсолютное значение b не превышает длину списка p или выражение ((p:|,b):>=, (b,0):>, (p:|,b:-):>=, (b,0):<)	да и b>0 или значение выражения (true, true, false, false)	**p_b**, где p=(p_1,..., p_n), n>=b
		да и b<0 или значение выражения (false, false, true, true)	**(p_1,..., p_{b-1}, p_{b+1},..., p_n)**, где p=(p_1,..., p_n), n>=|b|
		else	**BOUNDERROR**
p:+			
1	p:type	int	**<int, p>**
		float	**<float, p>**
		datalist	→ 2 (переходим к пункту 2)
		else	**BASEFUNCERROR**
2	p:|	2	→ 3
		else	**BASEFUNCERROR**
3	(p:1:type , p:2:type)	(bool, bool)	**<bool, p:1 ∨ p:2>**
		(int, int)	→ 4
		(int , float) или (float, int) или (float, float)	→ 5
		else	**BASEFUNCERROR**
4	Происходит переполнение	нет	**<int, p:1 + p:2>**
		else	**INTERROR**
5	Происходит переполнение	нет	**<float, p:1 + p:2>**
		else	**REALERROR**
p:-			
1	p:type	int	**<int, -p>**
		float	**<float, -p>**
		bool	**<bool, ¬p>**
		datalist	→ 2
		else	**BASEFUNCERROR**
2	p:|	2	→ 3
		else	**BASEFUNCERROR**

№	Выражение	Значения выражения	Результат
3	(p:1:type, p:2:type)	(bool, bool)	**<bool, p:1 ⊕ p:2>** «исключающее или» или «сложение по модулю 2»
		(int, int)	→ 4
		(int, float) или (float, int) или (float, float)	→ 5
		else	**BASEFUNCERROR**
4	Происходит переполнение	нет	**<int, p:1 - p:2>**
		else	**INTERROR**
5	Происходит переполнение	нет	**<float, p:1 - p:2>**
		else	**REALERROR**
p:*			
1	(p:type, p:|)	(datalist, 2)	→ 2
		else	**BASEFUNCERROR**
2	(p:1:type, p:2:type)	(bool, bool)	**<bool, p:1 ∧ p:2>**
		(int, int)	→ 3
		(int, float) или (float, int) или (float, float)	→4
		else	**BASEFUNCERROR**
3	Происходит переполнение	нет	**<int, p:1 * p:2>**
		else	**INTERROR**
4	Происходит переполнение	нет	**<float, p:1 * p:2>**
		else	**REALERROR**
p:/			
1	(p:type, p:|)	(datalist, 2)	→ 2
		else	**BASEFUNCERROR**
2	(p:1:type, p:2:type)	(int, int) или (int, float) или (float, int) или (float, float)	→ 3
		else	**BASEFUNCERROR**
3	p:2	0	**ZERODIVIDE**
		else	→ 4
4	Происходит переполнение	нет	**<float, p:1 / p:2>**
		else	**REALERROR**

№	Выражение	Значения выражения	Результат
p:%			
1	(p:type, p:|)	(datalist, 2)	→ 2
		else	**BASEFUNCERROR**
2	(p:1:type, p:2:type)	(int, int)	→ 3
		else	**BASEFUNCERROR**
3	p:2	0	**ZERODIVIDE**
		else	**(<int, a>,<int, b>)**, где a — результат целочисленного деления p:1 на p:2, b — остаток от деления p:1 на абсолютное значение p:2, то есть |b|= |p:1| mod |p:2|, b>0, если p:1>0, b<0, если p:1<0
p:= **p:!=**			
1	(p:type, p:|)	(datalist, 2)	→ 2
		else	**BASEFUNCERROR**
2	(p:1, p:2)	(<type, t1>, <type, t2>), где t1, t2 – встроенный или пользовательский тип	**<bool, p:1=p:2>** **<bool, p:1!=p:2>**
		else	→ 3
3	(p:1:type, p:2:type)	(int, int) или (int, float) или (float, int) или (float, float) или (char, char) или (bool, bool) или (func, func)	**<bool, p:1=p:2>** **<bool, p:1!=p:2>**
		else	**BASEFUNCERROR**
p:< **p:>** **p:<=** **p:>=**			
1	(p:type, p:|)	(datalist, 2)	→ 2
		else	**BASEFUNCERROR**
2	(p:1:type, p:2:type)	(int, int) или (int, float) или (float, int) или (float, float) или (char, char) или (bool, bool)	**<bool, p:1<p:2>** **<bool, p:1>p:2>** **<bool, p:1<=p:2>** **<bool, p:1>=p:2>**
		else	**BASEFUNCERROR**

№	Выражение	Значения выражения	Результат
p:?			
1	p:type	datalist	→ 2
		else	**BASEFUNCERROR**
2	(p:1:type, … , p:[p:|]:type)	(bool, … , bool)	**.** (пустое значение), при p:i=false, i=1, ... ,n или **[<int, i_1>, … <int, i_k>]**, где $0 < i_1 < i_2 < \ldots < i_k <= p:|$ и $p:i_s$ = true, s=1,2,...k
		else	**BASEFUNCERROR**
p:#			
1	p:type	datalist	→ 2
		else	**BASEFUNCERROR**
2	(p:1:type, … , p:[p:|]:type)	(datalist, … , datalist)	**((p_{11},...,p_{m1}),...,(p_{1n},...)...)** если p=((p_{11},..,p_{1n}), … , (p_{m1},..., p_{mk})) то есть происходит транспонирование подсписков списка
		else	**BASEFUNCERROR**
p:() p:datalist			
1			**(p)**
p:[] p:parlist			
1	p:type	datalist	**[p_1,...,p_n]**, где p=(p_1,... ,p_n)
		else	**p**
p:..			
1	p:type	datalist	→ 2
		else	**BASEFUNCERROR**
2	p:|	2	→ 3
		3	→ 5
		else	**BASEFUNCERROR**
3	(p:1:type, p:2:type)	(int, int)	→ 4
		else	**BASEFUNCERROR**
4	(p:1,p:2):<=	true	**(<int, p:1>, <int, p:1+1>,..., <int, p:1+k>)**, где k такое, что p:2=p:1+k
		else	**BOUNDERROR**

№	Выражение	Значения выражения	Результат
5	(p:1:type, p:2:type, p:3:type)	(int, int, int)	→ 6
		(int, float, int) или (int, float, float) или (float, int, int) или (float, int, float) или (float, float, int) или (float, float, float)	→ 7
		else	**BASEFUNCERROR**
6	отсутствуют расхождения между границами отрезка и шагом или выражение ((p:2,p:1):-,p:3):*	да или значение выражения больше нуля, то есть разность конца и начала отрезка имеет тот же знак что и шаг.	**(<int, p:1>, <int, p:1+p:3>, … , <int, p:1+k*p:3>)**, где k такое, что p:2>=p:1+k*p:3 и p:2<p:1+(k+1)*p:3, если p:1<p:2 или p:2<=p:1+k*p:3 и p:2>p:1+(k+1)*p:3, если p:1>p:2.
		else	**BOUNDERROR**
7	отсутствуют расхождения границ отрезка с шагом или выражение ((p:2,p:1):-,p:3):*	да или значение выражения больше нуля, то есть разность конца и начала отрезка имеет тот же знак что и шаг.	**(<float, p:1>, <float, p:1+p:3>, … , <float, p:1+k*p:3>)**, где k такое, что p:2 >=p:1+k*p:3 и p:2<p:1+(k+1)*p:3, если p:1<p:2 или p:2<=p:1+k*p:3 и p:2>p:1+(k+1)*p:3, если p:1>p:2.
		else	**BOUNDERROR**
p:<bool, b>			
1	значение селектора b равно true или выражение (b,true):=	да или значение выражения true	**p**
		else	**.** (пустое значение)
p:dup			
1	p:type	datalist	→ 2
		else	**BASEFUNCERROR**
2	p:|	2	→ 3
		else	**BASEFUNCERROR**
3	p:2:type	int	→ 4
		else	**BASEFUNCERROR**
4	(p:2,0):>	true	**(p_1, ..., p_n)**, где **p_i=p:1, i=1,..,n, n=p:2**
		else	**BASEFUNCERROR**

№	Выражение	Значения выражения	Результат
p:int			
	p:type	bool	**1**, если p=true **0**, если p=false
		char	Имеем отображение EncodingTable, которое каждому символу ставит в соответствие число EncodingTable : char → int с множеством значений (или множеством допустимых кодов символов) E. **EncodingTable(<char, p>)** = <int, p:int>, где p:int из множества допустимых кодов символов
		float	**<int, p1>** , где p1 получено из p округлением по математическим правилам
		int	**p**
		else	**BASEFUNCERROR**
p:float			
	p:type	bool	**true**, p!=0 **false**, p=0
		char	EncodingTable : int → char **EncodingTable(<char, p>):float** = <int, p:int>:float = <float, p:float>
		int	**<float, p>**
		float	**p**
		else	**BASEFUNCERROR**
p:char			
1	p:type	int	→ 2
		char	**p**
		else	**BASEFUNCERROR**
2	p∈E то есть p является допустимым кодом символа	true	Обратное отображение EncodingTable^(-1) : E → char **EncodingTable^(-1) (p)** = <char, p:char>
		else	**BASEFUNCERROR**
p:bool			
1	p:type	int или float	→ 2
		bool	**p**
		else	**BASEFUNCERROR**

№	Выражение	Значения выражения	Результат
2	(p,0):!=	true	**true**
		else	**false**
p:signal			
1			**.** (пустое значение)
p:in			
1	p:type	datalist	→ 2
		else	**BASEFUNCERROR**
2	p:|	2	→ 3
		else	**BASEFUNCERROR**
3	p:2:type	user_type, где user_type – пользовательский тип	**b**, где b ϵ{true, false} – значение, полученное в результате выполнения предиката, заданного в описании пользовательского типа user_type
		else	**BASEFUNCERROR**
p:user_type			
	p:in	true	**<user_type, p>**, где user_type – имя пользовательского типа
		else	**TYPEERROR**
p:value			
	p:type	user_type то есть тип пользователя	**p1**, если p=<user_type, p1>
		else	**VALUEERROR**

Приложение B

Стандартные теории языка спецификации

Теория стандартной модели (типа bool и ind).

Сигнатура теории стандартной модели содержит следующие константы:

$\Rightarrow$: bool→bool→bool,

$=$: $(\Pi\, A : \mathrm{Type}.\, A{\to}A{\to}\mathrm{bool})$,

$\varepsilon : (\Pi\, A : \mathrm{Type}.\, (A{\to}\mathrm{bool}){\to}A)$,

true, false : bool,

$\forall$, $\exists$, $\exists!$: $(\Pi\, A : \mathrm{Set}.\, (A{\to}\mathrm{bool}){\to}\mathrm{bool})$,

$\neg$: bool→bool,

$\wedge$, $\vee$, $\Leftrightarrow$: bool→bool→bool,

OneOne, Onto : $\Pi(A, B : \mathrm{Set}).\, (A{\to}B){\to}\mathrm{bool}$.

Используются следующие обозначения.

Для кванторов $\forall$, $\exists$ и $\exists!$:

$$\forall\, \sigma : \mathrm{Set}\, (\lambda x : \sigma.\, t) \equiv \forall x : \sigma.\, t,$$

$$\forall x_1.\, (\forall x_2.\, \ldots (\forall x_n.\, t) \ldots) \equiv \forall x_1\, x_2 \ldots x_n.\, t,$$

$$\forall a_1 : A.\, \forall a_2 : A.\, \ldots \forall a_n : A.\, f(a_1, \ldots, a_n) \equiv \forall (a_1, a_2, \ldots, a_n : A).\, f(a_1, \ldots, a_n),$$

$$\forall (a_1, a_2 : A).\, \forall b : B.\, f(a_1, a_2, b) \equiv \forall (a_1, a_2 : A)(b : B).\, f(a_1, a_2, b).$$

Для операций $\wedge$, $\vee$ и $\Leftrightarrow$:

$$(\wedge\, t_1\, t_2) \equiv (t_1 \wedge t_2).$$

Для отношений OneOne и Onto:

$$\mathrm{OneOne}\, A\, B\, f \equiv \mathrm{OneOne}\, f.$$

Для отрицания равенства:

$$\neg(A = B) \equiv A \neq B.$$

Аксиомы теории.

Определение введённых констант через $\Rightarrow$, $=$ и ε:

$\vdash \mathrm{true} = ((\lambda x : \mathrm{bool}.\, x) = (\lambda x : \mathrm{bool}.\, x))$,

$\vdash \mathrm{false} = \forall b : \mathrm{bool}.\, b$,

$\vdash \forall A : \mathrm{Set} = (\lambda P : A{\to}\mathrm{bool}.\, P = (\lambda x : \mathrm{bool}.\, \mathrm{true}))$,

$\vdash \exists A : \mathrm{Set} = (\lambda P : A{\to}\mathrm{bool}.\, P(\varepsilon\, P))$,

$\vdash \exists! A : \mathrm{Set} = (\lambda P : A{\to}\mathrm{bool}.\, (\exists x : A.\, P) \wedge (\forall (x, y : A).\, P\, x \wedge P\, y \Rightarrow (x = y)))$.

$\vdash \neg = \lambda b : \mathrm{bool}.\, b \Rightarrow \mathrm{false}$,

$\vdash \wedge = \lambda(b_1, b_2 : \mathrm{bool}).\, \forall b : \mathrm{bool}.\, (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b)$,

$\vdash \vee = \lambda(b_1, b_2 : \mathrm{bool}).\, \forall b : \mathrm{bool}.\, (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b)$,

$\vdash \forall(A, B : \mathrm{Set}).\, \mathrm{OneOne}\, A\, B = (\lambda f : A \to B.\, \forall(x_1, x_2 : A).\, (f\, x_1 = f\, x_2) \Rightarrow (x_1 = x_2))$,

$\vdash \forall(A, B : \mathrm{Set}).\, \mathrm{Onto}\, A\, B = (\lambda f : A \to B.\, \forall y : B.\, \exists x : A.\, y = f\, x)$,

Аксиомы.

$\vdash \forall b : \mathrm{bool}.\, (b = \mathrm{true}) \vee (b = \mathrm{false})$,

$\vdash \forall(b_1, b_2 : \mathrm{bool}).\, (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)$,

$\vdash \forall(A, B : \mathrm{Set})(f : A \to B).\, (\lambda x : A.\, f\, x) = f$,

$\vdash \forall(P : A \to \mathrm{bool})(x : A).\, P\, x \Rightarrow P(\varepsilon\, P)$ (аксиома выбора)

$\vdash \exists f : \mathrm{ind} \to \mathrm{ind}.\, \mathrm{OneOne}\, f \wedge \neg(\mathrm{Onto}\ \ f)$ (аксиома бесконечности)

Теория натуральных чисел (с нулём).

Данная теория является расширением теории стандартной модели. Её сигнатура $\Sigma_{\mathbb{N}}$ включает следующие константы:

$\mathbb{N} : \mathrm{Set}$,

$0 : \mathbb{N}$,

$\mathrm{SUC} : \mathbb{N} \to \mathbb{N}$ — функция следования,

$1, 2, 3, \ldots : \mathbb{N}$ — константы, обозначающие числа,

$+\ , \cdot\ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ — символы операций,

Аксиомы теории.

1. Аксиомы теории с равенством [146]:

   $\vdash \forall(x, y : \mathrm{Type}).\, ((x = y) \Rightarrow (y = x))$,

   $\vdash \forall(x, y, z : \mathrm{Type}).\, (((x = y) \wedge (y = z)) \Rightarrow (x = z))$.

2. Аксиомы Дж. Пеано [136, 145]:

   $\vdash \forall n : \mathbb{N}.\, \neg(\mathrm{SUC}\, n = 0)$,

   $\vdash \forall(n, m : \mathbb{N}).\, (\mathrm{SUC}\, m = \mathrm{SUC}\, n) \Rightarrow (m = n)$,

   $\vdash \forall P : (\mathbb{N} \to \mathrm{bool}).\, P\, 0 \wedge (\forall n : \mathbb{N}.\, Pn \Rightarrow P(\mathrm{SUC}\, n)) \Rightarrow (\forall n : \mathbb{N}.\, P\, n)$. (аксиома индукции)

3. Бесконечное множество аксиом, определяющих числа [125]:

   $\vdash 1 = \mathrm{SUC}\, 0$, $\vdash 2 = \mathrm{SUC}\, 1$, $\vdash 3 = \mathrm{SUC}\, 2$, $\ldots$

4. Аксиомы, определяющие операции сложения и умножения:

   $\vdash (\forall m : \mathbb{N}.\, 0 + m = m) \wedge (\forall(m, n : \mathbb{N}).\, m + (\mathrm{SUC}\, n) = \mathrm{SUC}(m + n))$,

   $\vdash (\forall n : \mathbb{N}.\, 0 \cdot n = 0) \wedge (\forall(m, n : \mathbb{N}).\, (\mathrm{SUC}\, m) \cdot n = (m \cdot n) + n)$,

Теория целых чисел.

Аксиоматическая система для целых чисел [136, 142, 144], строится на основе аксиоматики натуральных чисел. Сигнатура $\Sigma_{\mathbb{Z}}$ включает следующие константы:

$\mathbb{Z} : \mathrm{Set}$,

$0, 1 : \mathbb{Z}$,

$- : \mathbb{Z} \to \mathbb{Z}$,

$+, \cdot \ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$,

$-, \quad \mathrm{mod}\ , /\ : \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$,

$<\ , >\ , \leqslant\ , \geqslant\ : \mathbb{Z} \to \mathbb{Z} \to \mathrm{bool}$ — символы отношений,

Аксиомы теории.

1. Аксиомы коммутативного кольца [142]:

   $\vdash \forall (a, b : \mathbb{Z}).\, \exists! c : \mathbb{Z}.\, a + b = c$,

   $\vdash \forall (a, b : \mathbb{Z}).\, a + b = b + a$,

   $\vdash \forall (a, b, c : \mathbb{Z}).\, (a + b) + c = a + (b + c)$,

   $\vdash \forall a : \mathbb{Z}.\, a + 0 = a$,

   $\vdash \forall a : \mathbb{Z}.\, \exists (-a) : \mathbb{Z}.\, a + (-a) = 0$,

   $\vdash \forall (a, b : \mathbb{Z}).\, \exists! c : \mathbb{Z}.\, a \cdot b = c$,

   $\vdash \forall (a, b : \mathbb{Z}).\, a \cdot b = b \cdot a$,

   $\vdash \forall (a, b, c : \mathbb{Z}).\, (a \cdot b) \cdot c = a \cdot (b \cdot c)$,

   $\vdash \forall a : \mathbb{Z}.\, a \cdot 1 = a$,

   $\vdash \forall (a, b, c : \mathbb{Z}).\, ((a + b) \cdot c = a \cdot c + b \cdot c) \wedge (c \cdot (a + b) = c \cdot a + c \cdot b)$.

2. Аксиомы, связывающие множество целых чисел со множеством натуральных чисел:

   2.1. Множество целых чисел $\mathbb{Z}$ содержит подмножество, изоморфное множеству натуральных чисел $\mathbb{N}$.

   2.2. Если M — подмножество $\mathbb{Z}$, такое, что операция вычитания не выводит за пределы M. Тогда M совпадает со всем $\mathbb{Z}$ (аксиома минимальности).

   Ввиду такого изоморфизма, далее считаем, что константы 1, 2, 3, ...обозначают положительные целые числа.

   Замечание. В формулах прямое и обратное преобразование элементов множеств $\mathbb{N}$ и $\mathbb{Z}$ будет подразумеваться и явно не прописываться. Это сделано ввиду отсутствия типа $\mathbb{N}$ в языке Пифагор, в результате чего типы $\mathbb{N}$ и $\mathbb{Z}$ относительно него неразличимы.

3. Аксиомы, определяющие $-$ (бинарная операция), $\mod$, $/$:

   $\vdash \forall(m, n : \mathbb{Z}).\, m - n = m + (-n)$,

   $\vdash \forall(k, n : \mathbb{Z}).\, (k > 0) \wedge (n > 0) \wedge ((k \mod n) = \varepsilon n : \mathbb{Z}.\, \exists q : \mathbb{Z}.\, (k = (q \cdot n) + r) \wedge (r < n))$,

   $\vdash \forall(k, n : \mathbb{Z}).\, ((-k \mod n) = -(k \mod n)) \wedge ((k \mod -n) = -(k \mod n))$,

   $\vdash \forall(k, n : \mathbb{Z}).\, (k/n) = \varepsilon q : \mathbb{Z}.\, k = (q \cdot n) + (k \mod n)$.

4. Аксиомы полного дискретного упорядочения без первого или последнего элемента [143]:

   $\vdash \forall a : \mathbb{Z}.\, \neg(a < a)$,

   $\vdash \forall(a, b, c : \mathbb{Z}).\, ((a < b) \wedge (b < c)) \Rightarrow (a < c)$,

   $\vdash \forall(a, b : \mathbb{Z}).\, (a = b) \vee (a < b) \vee (b < a)$,

   $\vdash \forall(a, b : \mathbb{Z}).\, (a < b) \Leftrightarrow ((b = a + 1) \vee (a + 1 < b))$,

   $\vdash \forall a : \mathbb{Z}.\, \exists b : \mathbb{Z}.\, a = (b + 1)$.

5. Аксиомы, определяющие $>$, $\leqslant$, $\geqslant$ через $<$ и $=$ [125]:

   $\vdash \forall(m, n : \mathbb{Z}).\, (m > n) = (n < m)$,

   $\vdash \forall(m, n : \mathbb{Z}).\, (m \leqslant n) = ((m < n) \vee (m = n))$,

   $\vdash \forall(m, n : \mathbb{Z}).\, (m \geqslant n) = ((m > n) \vee (m = n))$.

Теория действительных чисел.

Теория действительных чисел является расширением теории натуральных чисел. Сигнатура $\Sigma_{\mathbb{R}}$ содержит следующие константы:

$\mathbb{R} : \mathrm{Set}$,

$0, 1 : \mathbb{R}$,

$+, \cdot, -, / \ : \mathbb{R} \to \mathbb{R} \to \mathbb{R}$,

$- \ : \mathbb{R} \to \mathbb{R}$,

$< , > , \leqslant , \geqslant \ : \mathbb{R} \to \mathbb{R} \to \mathrm{bool}$,

$\mathrm{NR} : \mathbb{N} \to \mathbb{R}$,

$\mathrm{ZR} : \mathbb{Z} \to \mathbb{R}$,

$\mathrm{RZ} : \mathbb{R} \to \mathbb{Z}$.

Аксиомы теории [142, 143]:

1. Аксиомы коммутативного поля:

   аксиомы кольца (аналогичны аксиомам целых чисел),

   $\vdash \forall a : \mathbb{R}.\, \exists a^{-1} : \mathbb{R}.\, \neg(a = 0) \Rightarrow (a \cdot a^{-1} = 1)$,

$\vdash \neg(1 = 0)$.

2. Аксиомы линейного порядка, согласованного с операциями поля:

   $\vdash \forall a : \mathbb{R}.\, \neg(a < a)$,

   $\vdash \forall (a, b, c : \mathbb{R}).\, ((a < b) \wedge (b < c)) \Rightarrow (a < c)$,

   $\vdash \forall (a, b : \mathbb{R}).\, (a = b) \vee (a < b) \vee (b < a)$,

   $\vdash \forall (a, b, c : \mathbb{R}).\, (a < b) \Rightarrow (a + c < b + c)$,

   $\vdash \forall (a, b, c : \mathbb{R}).\, (a < b \wedge 0 < c) \Rightarrow (a \cdot c < b \cdot c)$.

3. Аксиомы, определяющие $-$, $/$, $>$, $\leqslant$, $\geqslant$ через $=$, $+$, $\cdot$, $<$, $-$ (унарный).

   $\vdash \forall (a, b : \mathbb{R}).\, a - b = a + (-b)$,

   $\vdash \forall (a, b : \mathbb{R}).\, a \leqslant b = \neg(a < b)$,

   $\vdash \forall (a, b : \mathbb{R}).\, a > b = b < a$,

   $\vdash \forall (a, b : \mathbb{R}).\, a \geqslant b = b \leqslant a$,

   $\vdash \forall (a, b : \mathbb{R}).\, a/b = a \cdot (b^{-1})$.

4. Аксиома непрерывности (полноты):

   Каковы бы ни были непустые множества $A \subset \mathbb{R}$ и $B \subset \mathbb{R}$, такие, что для любых двух элементов $a \in A$ и $b \in B$ выполняется неравенство $a \leqslant b$, существует такое число $\xi \in \mathbb{R}$, что для всех $a \in A$ и $b \in B$ имеет место соотношение $a \leqslant \xi \leqslant b$.

5. Аксиома включения множества натуральных чисел во множество действительных, как гомоморфизма NR. Аналогичный гомоморфизм существует и для целых чисел, он пределен функцией ZR. Функция RZ является обратной функцией к ZR и определяется следующей аксиомой:

   $\forall x : \mathbb{R}.\, \exists n : \mathbb{Z}.\, (\mathrm{ZR}(n) \leqslant x \leqslant \mathrm{ZR}(n + 1)) \wedge (\mathrm{RZ}(x) = n)$.

Теория списков.

Списки являются составными (полиморфными типами) типами [41, 125]. Они представляют собой последовательность элементов некоторого типа. Тип элементов списка является параметром.

Сигнатура Σ_L содержит следующие символы:

$\mathrm{list} : \mathrm{Type} \to \mathrm{Type}$,

$\mathrm{nil} : \Pi\, A : \mathrm{Type}.\, A \to \mathrm{list}\, A$,

$\mathrm{cons} : \Pi\, A : \mathrm{Type}.\, A \to \mathrm{list}\, A \to \mathrm{list}\, A$,

$\mathrm{head} : \Pi\, A : \mathrm{Type}.\, \mathrm{list}\, A \to A$,

$\mathrm{tail} : \Pi\, A : \mathrm{Type}.\, \mathrm{list}\, A \to \mathrm{list}\ A$,

$\mathrm{length} : \Pi\, A : \mathrm{Type}.\, \mathrm{list}\, A \to \mathbb{N}$,

$\mathrm{elem} : \Pi\, A : \mathrm{Type}.\, \mathbb{N} \to \mathrm{list}\, A \to A$.

Аксиомы теории списков [125, 147].

1. Аксиомы конструирования списка:

   $< B : \mathrm{Type} > .\ \vdash \forall (A : \mathrm{Type})(x : B)(\mathrm{f} : B \to A \to \mathrm{list}\, A \to B).\, \exists !(\mathrm{fn} : \mathrm{list}\, A \to B).$

   $$(\mathrm{fn}\, \mathrm{nil} = x) \wedge (\forall (h : A)(t : \mathrm{list}\, A).\, \mathrm{fn}\, (\mathrm{cons}\, h\, t) = \mathrm{f}(\mathrm{fn}\, t)\, h\, t),$$

   где B — ранее определённый подходящий тип, подтипу которого будет изоморфен новый введенный тип $\mathrm{list}\, A$ [125, 148]. Из этой аксиомы могут быть выведены все последующие аксиомы конструирования списка.

   $\vdash \forall (A : \mathrm{Type})(P : \mathrm{list}\, A \to \mathrm{bool}).\, P\, \mathrm{nil} \wedge (\forall (t : \mathrm{list}\, A).\, P\, t \Rightarrow \forall h : A.\, P(\mathrm{cons}\, h\, t)) \Rightarrow$

   $$\Rightarrow (\forall (x : \mathrm{list}\, A).\, P\, x), \qquad \text{(принцип индукции для списков)}$$

   $\vdash \forall (A : \mathrm{Type})(l : \mathrm{list}\, A).\, (l = \mathrm{nil}) \vee (\exists (t : \mathrm{list}\, A)(h : A).\, l = \mathrm{cons}\, h\, t)$,

   $\vdash \forall (A : \mathrm{Type})(h, g : A)(t, q : \mathrm{list}\, A).\, (\mathrm{cons}\, h\, t = \mathrm{cons}\, g\, q) = ((h = g) \wedge (t = q))$,

   $\vdash \forall (A : \mathrm{Type})(h : A)(t : \mathrm{list}\, A).\, \neg(\mathrm{nil} = \mathrm{cons}\, h\, t)$,

   $\vdash \forall (A : \mathrm{Type})(h : A)(t : \mathrm{list}\, A).\, \neg(\mathrm{cons}\, h\, t = \mathrm{nil})$.

2. Аксиомы, определяющие функции над списками:

   $\vdash \forall (A : \mathrm{Type})(h : A)(t : \mathrm{list}\, A).\, \mathrm{head}(\mathrm{cons}\, h\, t) = h$,

   $\vdash \forall (A : \mathrm{Type})(h : A)(t : \mathrm{list}\, A).\, \mathrm{tail}(\mathrm{cons}\, h\, t) = t$,

   $\vdash \forall A : \mathrm{Type}.\, (\mathrm{length}\, \mathrm{nil} = 0) \wedge (\forall (h : A)(t : \mathrm{list}\, A).\, \mathrm{length}(\mathrm{cons}\, h\, t) = \mathrm{SUC}(\mathrm{length}\, t))$,

   $\vdash \forall (A : \mathrm{Type})(l : \mathrm{list}A).\, (\mathrm{elem}\, 1\, l = (\mathrm{head}\, l)) \wedge$

   $$\wedge (\forall n : \mathbb{N}.\, (1 < n) \Rightarrow (\mathrm{elem}\, (\mathrm{SUC}\ n)\, l = \mathrm{elem}\, n\, (\mathrm{tail}\, l))).$$

Приложение Г
Описание условий корректности ФПП программ с помощью логики HOL

Таблица Г.1 — Соответствие типов языка спецификации ФПП программ типам в логики HOL

Типы языка спецификации ФПП программ	**Типы HOL (теория HOL)**
bool, ind, $\mathbb{N}$	bool, ind, num (BASIC-HOL)
целые числа $\mathbb{Z}$	integer (integer)
действительные числа $\mathbb{R}$	real (reals)
символы char	ascii (string)
множество сигналов signal	—
множество констант ошибок error	—
множество констант func	—
множество задержанных списков delaylist	—
списки list	*list (list)
гетерогенные списки datalist, parlist	—

В таблице Г.1 приведено соответствие теорий языка спецификации ФПП программ имеющимся теориям системы HOL. Не все типы языка спецификации ФПП программ имеют аналоги в логике HOL. Логика системы HOL может быть дополнена недостающими типами. Один из способов — это создание новой теории «pifagor», описанной ниже. Теория включает в себя определение констант и типов языка спецификации ФПП программ, отсутствующих в HOL. Для описания списков данных (параллельных списков) вводится два типа. Первый тип datalist (parlist) использует встроенный тип HOL *list, второй тип — datalist_P (parlist_P) использует встроенный тип (*,**)prod.

Для системы HOL88 листинг, определяющий теорию pifagor имеет следующий вид:

```
% Создание теории pifagor %
new_theory(‘pifagor‘);;

% Загрузка необходимых библиотек %
load_library(‘integer‘);;
load_library(‘reals‘);;
load_library(‘string‘);;

%Определение отношений >, >=, <, <= для целых чисел %
% < %
let integer_lt =new_infix_definition (‘integer_lt‘,
 "integer_lt (a:integer) (b:integer) = NEG (a minus b)");;

% <= %
let integer_le =new_infix_definition (‘integer_le‘,
 "$integer_le (a:integer) (b:integer) = ~(b integer_lt a)");;
```

```
% > %
let integer_gt =new_infix_definition (‘integer_gt‘,
 "integer_gt (a:integer) (b:integer) = b integer_lt a");;

% >= %
let integer_ge =new_infix_definition (‘integer_ge‘,
 "integer_ge (a:integer) (b:integer) = b integer_le a");;


% Определение констант %
new_constant(‘MaxInt‘,":integer");;
new_constant(‘MinInt‘,":integer");;
new_constant(‘MaxFloat‘,":real");;
new_constant(‘MinFloat‘,":real");;

% Определение недостающих типов %
% Тип ошибок error %
let error_Axiom = define_type ‘error_Axiom‘
                  ‘error = BASEFUNCERROR | BOUNDERROR | INTERROR |
                   REALERROR | ZERODIVIDE | TYPEERROR‘;;

% Тип имен функций Пифагор func %
let func_Axiom = define_type ‘func_Axiom‘
                  ‘func = type_p | len_p | plus_p | minus_p | dot_p |
                  div_p | mod_p | eq_p | neq_p | ls_p | gr_p | leq_p |
                  geq_p | quest_p | hash_p | datalist_p | parlist_p |
                  dup_p | dd_p | int_p | float_p | char_p | bool_p |
                  signal_p‘;;

% Тип signal %
%-------------- %
let signal_Axiom = define_type ‘signal_Axiom‘ ‘signal = SIGNAL‘;;

% Тип delaylist определяется как обертка над типом string %
%-------------- %
let delaylist_Axiom = define_type ‘delaylist_Axiom‘
                      ‘delaylist = DELAYLIST string‘;;

% Функция получения строки из задержанного списка %
let dest_delayList = new_recursive_definition false delaylist_Axiom
                      ‘dest_delayList‘ "(DEST_DELAYLIST (DELAYLIST (t) )= t)";;

% Тип datalist определяется через тип list %
%-------------- %
let datalist_Axiom = define_type ‘datalist_Axiom‘ ‘datalist = DATALIST (*list)‘;;

% Функция получения списка *list из списка данных *)%
let dest_datalist = new_recursive_definition false datalist_Axiom ‘dest_datalist‘
                     "(dest_datalist (DATALIST (l:*list) )= l)";;

% Тип datalist определяется через тип pair  %
let datalist_Axiom_P = define_type ‘datalist_Axiom_P‘
```

```
                      `datalist_P = DATALIST_P ((*,**)prod)`;;

% Функция получения пары (*,**)prod из списка данных  %
let dest_datalist_P = new_recursive_definition false datalist_Axiom_P
                      `dest_datalist_P`
                      "(dest_datalist_P (DATALIST_P (l:(*,**)prod) )= l)";;

% Тип parlist определяется через тип list  %
%--------------%
let parlist_Axiom = define_type `parlist_Axiom` `parlist = PARLIST (*list)`;;

% Функция получения списка *list из параллельного списка *) %
let dest_parlist =  new_recursive_definition false parlist_Axiom `dest_parlist`
                    "(dest_parlist (PARLIST (l:*list) )= l)";;

% Тип parlist определяется через тип pair %
let parlist_Axiom_P = define_type `parlist_Axiom_P`
                      `parlist_P = PARLIST_P ((*,**)prod)`;;

% Функция получения пары (*,**)prod из параллельного списка  %
let dest_parlist_P = new_recursive_definition false parlist_Axiom_P
                     `dest_parlist_P`
                     "(dest_parlist_P (PARLIST_P (l:(*,**)prod) )= l)";;

close_theory();;
```

Приведём несколько примеров перевода формул языка спецификации Пифагор на язык системы HOL.

Пример 1. Задана тройка Хоара для функции нахождения значения арифметического выражения $(a \cdot b + c)$:

$$\boxed{x = (a\colon \text{int}, b\colon \text{int}, c\colon \text{int})}\ \texttt{((a, b):*,c):+}\ \to r\ \boxed{(r \in \text{int}) \wedge (r = a \cdot b + c)}.$$

Финальная формула при доказательстве корректности данной тройки:

$$((a, b, c, (a \cdot b), (a \cdot b + c) \in \text{int}) \wedge (r = a \cdot b + c)) \Rightarrow (r = a \cdot b + c).$$

Данная формула записанная на языке системы HOL имеет вид:

```
(
  (((a:integer) times (b:integer)) integer_lt MaxInt)/\
  ((a times b) integer_gt MinInt)/\((a times b plus c) integer_lt MaxInt)/\
  ((a times b plus c) integer_gt MinInt)/\(r = (a times b) plus c)
)
==>(r = ((a times b) plus c))
```

Пример 2. Тройка Хоара для функции `abs`, находящей абсолютное значение целого числа, имеет вид:

$$\boxed{(arg \in \text{int})}\ \texttt{(arg:-,arg):[((arg,0):[<,>=]):?]:.}\ \to r\ \boxed{\begin{array}{l}(r \in \text{int}) \wedge ((r = arg) \wedge (arg \geqslant 0)) \\ \vee((r = -arg) \wedge (arg < 0))\end{array}}.$$

Ниже приведена одна из финальных формул, получаемых при доказательстве корректности программы:

$$\Big((arg \in \text{int}) \wedge (a_1 \in \text{bool}) \wedge (a_1 = (arg < 0)) \wedge (a_2 \in \text{bool}) \wedge (a_2 = (arg \geqslant 0)) \wedge$$
$$\wedge((a_1, a_2) = (\text{true}, \text{false})) \wedge (a_3 = [1]) \wedge (r_1 \in \text{int}) \wedge (r_1 = (-arg))) \wedge (r = r_1)\Big) \Rightarrow$$
$$\Rightarrow \Big((r \in \text{int}) \wedge ((r = arg) \wedge (arg \geqslant 0)) \vee ((r = -arg) \wedge (arg < 0))\Big).$$

Данная формула, записанная на языке системы HOL:

```
(
    ((arg:integer) integer_lt MaxInt)/\(arg integer_gt MinInt)/\
    (a1 = (arg integer_lt (INT 0)))/\
    (a2 = (arg integer_ge (INT 0)))/\
    (a1 = T)/\(a2 = F)/\(a3 = PARLIST ([1]))/\
    (r1 integer_lt MinInt)/\(r1 integer_gt MinInt)/\
    (r1 = (neg arg))/\(r = r1)
)
==>(r integer_lt MaxInt)/\(r integer_gt MinInt)/\
   (
       ((r = arg)/\(arg integer_ge (INT 0)))\/
       ((r = (neg arg))/\(arg integer_lt (INT 0)))
   )
```

Пример 3. Для функции `order`, находящей число десятичных знаков у натурального числа n, определены предусловие P и постусловие Q следующим образом:

$$P \equiv (n, p, k \in int) \wedge (n, p, k > 0) \wedge (p = 10^k) \wedge \left((n \geqslant 10^k) \vee ((n < 10^k) \wedge (n \geqslant 10^{k-1}))\right),$$
$$Q \equiv (res \in int) \wedge (n \geqslant 10^{res-1}) \wedge (n < 10^{res}).$$

На языке системы HOL формулы можно записать следующим образом:

```
(   (n>0)/\(p>0)/\(k>0)/\
    (n< FST(REP_integer MaxInt))/\(p<FST(REP_integer MaxInt))/\
    (k<FST(REP_integer MaxInt))/\(p=(EXP 10 k))/\
    ( (n>= (EXP 10 k))\/ (n< (EXP 10 k))/\(n>=(EXP 10 (k-1)) ))/\
    (n<p)/\(k=res)    )

((n >= (EXP 10 (res-1)))/\(n< (EXP 10 res)))
```

Пример 4. Имеем функцию `parse`, входным аргументом которой является строка символов *str*. В заданной строке *str* можно выделить две подстроки str_1 и *tail* таких, что $str = str_1 \circ tail$, где «$\circ$» обозначает конкатенацию двух строк. Строка str_1 удовлетворяет правилу, записанном в нотации Бэкуса-Наура:

```
<выражение> ::= x | +<выражение><выражение> | *<выражение><выражение> .
```

Строка *tail* содержит произвольные символы, в том числе, она может быть пустой. В результате работы функция `parse` возвращает строку *tail*.

Предусловие $P(str)$ функции `parse` на языке логики:

$$F(n_1, f) \equiv \Big(f(1) = \text{“x”}\Big) \wedge \Big(f(2) = \text{“+xx”} \vee f(2) = \text{“*xx”}\Big) \wedge$$
$$\wedge \left(\bigvee_{i=1}^{n_1-1} \Big(f(n_1) = \text{“+”} \circ f(i) \circ f(n_1 - i)\Big) \vee \Big(f(n_1) = \text{“*”} \circ f(i) \circ f(n_1 - i)\Big) \right),$$

$$P(str) \equiv \exists (n, m : \mathbb{N})(f : (\Pi\, n : \mathbb{N}.\, \text{datalist}\, L_{(2n-1)})).\, F(n, f) \wedge (str \in \text{datalist}\, L_{(2n-1+m)}) \wedge$$
$$\wedge \big(\exists str_1 : \text{datalist}\, L_{(2n-1)}.\, (str_1 = f(n)) \wedge (str_1 = \text{sublist}(str, 1, 2n-1))\big) \wedge$$
$$\wedge \big(\exists tail : \text{datalist}\, L_{(m)}.\, (tail = \text{sublist}(str, 2n, 2n-1+m))\big),$$

где $L_{(k)}$ — список, содержащий k элементов char; F — вспомогательная функция с типом $\big(\mathbb{N} \to (\Pi\, n_1 : \mathbb{N}.\, \text{datalist}\, L_{(2n_1-1)}) \to \text{bool}\big)$; $\text{sublist}(s, p, q)$ — функция, возвращающая подстроку строки s, начиная с позиции p до q, функция возвращает пустую строку, если $p = q$.

Запишем предусловие функции `parse` на языке системы HOL. Для представления строк используем тип string. Для записи формулы необходимо доопределить несколько функций, работающих со строками. Это функции определения длины строки, объединения двух строк и получения подстроки. Ниже приведён листинг, дополняющий теорию pifagor в системе HOL необходимыми функциями:

```
% Функция LEN нахождения длины строки %
let LEN =
new_recursive_definition false string_Axiom `LEN`
"(LEN `` = 0) /\ (LEN (STRING a s) = (LEN s) + 1)";;

% Функция FIRST_CHAR получения первого символа %
let FIRST_CHAR =
new_recursive_definition false string_Axiom `FIRST_CHAR`
"(FIRST_CHAR `` = ``) /\ (FIRST_CHAR (STRING a s) = STRING a ``)";;

% Функция APP_CHAR добавления символа в конец строки %
let APP_CHAR =
new_recursive_definition false string_Axiom `APP_CHAR`
"((APP_CHAR (``) (a:ascii) ) = (STRING a ``)) /\
 ((APP_CHAR  (STRING a1 s) (a:ascii)) = (STRING (a1) (APP_CHAR s a) ))";;

% Функция CONCAT_STR объединения двух строк %
let CONCAT_STR =
new_recursive_definition false string_Axiom `CONCAT_STR`
"((CONCAT_STR (s1:string) (``) )=s1) /\
 ((CONCAT_STR (s1:string) (STRING a s2) )= (CONCAT_STR (APP_CHAR s1 a) (s2)))";;

% Функция POP_STR_FRONT удаления первого символа в строке %
let POP_STR_FRONT =
new_recursive_definition false string_Axiom `POP_STR_FRONT`
"
```

```
 ((POP_STR_FRONT `` )= ``)/\
 ((POP_STR_FRONT (STRING a s) )= s)
";;

% Функция CUT_STR_HEAD удаления n символов с начала строки %
let CUT_STR_HEAD =
new_prim_rec_definition (
`CUT_STR_HEAD`,
"((CUT_STR_HEAD 0 s:string )= s)/\
 ((CUT_STR_HEAD (SUC n) s:string )= (CUT_STR_HEAD n (POP_STR_FRONT s)))"
);;

% Функция POP_STR_BACK удаления первого символа с конца строки %
let POP_STR_BACK =
new_recursive_definition false string_Axiom `POP_STR_BACK`
"
 ((POP_STR_BACK `` )= ``)/\
 ((POP_STR_BACK (STRING a s) )= (STRING (a) (POP_STR_BACK s) ))
";;

% Функция CUT_STR_TAIL удаления n символов с конца строки %
let CUT_STR_TAIL =
new_prim_rec_definition (
`CUT_STR_TAIL`,
"((CUT_STR_TAIL 0 s:string )= s)/\
 ((CUT_STR_TAIL (SUC n) s:string )= (CUT_STR_TAIL n (POP_STR_BACK s)))"
);;

% Функция SUB_STR получения подстроки %
let SUB_STR =
new_definition (
`SUB_STR`,
"(SUB_STR (n:num) (m:num) (s:string) )= CUT_STR_TAIL m (CUT_STR_HEAD n s)"
);;
```

Также для описания предусловия функции `parse` требуется определение нескольких вспомогательных функций:

```
% Функция дизъюнкции от 0 до n f(i) %
let REC_DISJUNCITON =
new_prim_rec_definition (
`REC_DISJUNCITON`,
"((REC_DISJUNCITON 0 (f:(num->bool)) )= (f 0))/\
((REC_DISJUNCITON (SUC n) (f:(num->bool)) )= ((f n)\/(REC_DISJUNCITON n f)))
");;

% Вспомогательная функция aux_f принимает натуральное число num,  %
% функцию num->string и возвращает функцию num->bool  %
new_definition (
`aux_f`,
"(aux_f:(num->(num->string)->(num->bool))) (n:num) (f:(num->string)) =
    \i:num.(
```

```
              (  (f n) = ( CONCAT_STR ‘+‘ (CONCAT_STR (f i) (f (n-i))) ) )
              \/
              (  (f n) = ( CONCAT_STR ‘*‘ (CONCAT_STR (f i) (f (n-i))) ) )
           )"
);;

% Вспомогательная функция FF определяет принадлежность функции (num->string) %
% ко множеству функций, распознающих выражение %
% <выражение> ::= x | +<выражение><выражение> | *<выражение><выражение> %
new_definition (
‘FF‘, "(FF:(num->(num->string)->bool)) n f  =
(
    ((f 1)=‘x‘)/\
    (   ((f 2)=‘+xx‘)  \/ ((f 2)=‘*xx‘)  )/\
    (REC_DISJUNCITON (n-2) (aux_f n f))

)"
);;
```

В результате предусловие программы `parse` может быть записано на языке логики HOL следующим образом:

```
? (n:num) (m:num) (f:(num->string)).
   (FF n f)/\(LEN str = (2*n-1+m))/\
   (? (str1:string).
         (LEN str1 = (2*n-1))/\
         (str1= (f n))/\
         (str1 = (SUB_STR 1 (2*n-1) str))
    ) /\
    (? (tail:string).
         (LEN tail = m)/\
         (tail = (SUB_STR (2*n) (2*n-1+m) str))
    )
```

Приложение Д

Аксиомы встроенных функций языка Пифагор

1. $(p \in T) \wedge (T \in \mathrm{Set})$ `p:.` $\to r$ $(r \in T) \wedge (r = p)$.

2.1. $(p \in T) \wedge (T \in \mathrm{Set})$ `p:type` $\to r$ $(r \in \mathrm{Set}) \wedge (r = T)$.

2.2. $(p \in \mathrm{Set})$ `p:type` $\to r$ $(r \in \mathrm{error}) \wedge (r =$ `TYPEERROR`$)$.

3.1. $(p \in \mathrm{datalist}\ L) \wedge (L \in \mathrm{list\,Set})$ `p:|` $\to r$ $(r \in \mathrm{int}) \wedge (r \geqslant 0) \wedge (r = \mathrm{hlength}(p))$.

3.2. $(p \notin \mathrm{datalist}\ L) \wedge (L \in \mathrm{list\,Set})$ `p:|` $\to r$ $(r \in \mathrm{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

4.1. $(b \in \mathrm{int}) \wedge (p \in \mathrm{datalist}\ L) \wedge (L \in \mathrm{list\,Set}) \wedge (b = 0)$ `p:<int, b>` $\to r$ $(r \in \mathrm{signal}) \wedge (r = \bullet)$.

4.2. $(b \in \mathrm{int}) \wedge (p \in \mathrm{datalist}\ L) \wedge (L \in \mathrm{list\,Set}) \wedge$
$(n = \mathrm{hlength}(p)) \wedge (b > 0) \wedge (n \geqslant b)$
`p:<int, b>` $\to r$ $(r \in L[b]) \wedge (r = p[b])$.

4.3. $(b \in \mathrm{int}) \wedge (p \in \mathrm{datalist}\ L) \wedge$
$(L \in \mathrm{list\,Set}) \wedge$
$(n = \mathrm{hlength}(p)) \wedge$
$(b < 0) \wedge (n \geqslant (-b))$
`p:<int, b>` $\to r$
$(r \in \mathrm{datalist}\, L_1) \wedge (L_1 \in \mathrm{list\,Set}) \wedge$
$(n = \mathrm{hlength}(p)) \wedge (\mathrm{hlength}(r) = (n-1)) \wedge$
$(\forall i \in \mathbb{N}.\, ((1 \leqslant i < (-b)) \Rightarrow (r[i] \in L_1[i]) \wedge (r[i] = p[i])) \wedge$
$(((-b) < i \leqslant n) \Rightarrow (r[i-1] \in L_1[i]) \wedge (r[i-1] = p[i]))$.

4.4. $(b \in \mathrm{int}) \wedge (p \in \mathrm{datalist}\ L) \wedge (L \in \mathrm{list\,Set}) \wedge$
$(n = \mathrm{hlength}(p)) \wedge \big((b > 0 \wedge n < b) \vee (b < 0 \wedge n < (-b))\big)$
`p:<int, b>` $\to r$ $(r \in \mathrm{error}) \wedge (r =$ `BOUNDERROR`$)$.

4.5. $(b \in \mathrm{int}) \wedge (p \notin \mathrm{datalist}\ L) \wedge (L \in \mathrm{list\,Set})$ `p:<int, b>` $\to r$ $(r \in \mathrm{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

5.1. $p = (a : \mathrm{bool}, b : \mathrm{bool})$ `p:+` $\to r$ $(r \in \mathrm{bool}) \wedge r = (a \vee b)$.

5.2. $p = (a : \mathrm{int}, b : \mathrm{int}) \wedge (a + b) \in \mathrm{int}$ `p:+` $\to r$ $(r \in \mathrm{int}) \wedge (r = a + b)$.

5.3. $p = (a : \mathrm{int}, b : \mathrm{int}) \wedge (a + b) \notin \mathrm{int}$ `p:+` $\to r$ $(r \in \mathrm{error}) \wedge (r =$ `INTERROR`$)$.

5.4. $p = \big((a : \mathrm{float}, b : \mathrm{int}) \mid (a : \mathrm{int}, b : \mathrm{float}) \mid$
$(a : \mathrm{float}, b : \mathrm{float})\big) \wedge ((a + b) \in \mathrm{float})$
`p:+` $\to r$ $(r \in \mathrm{float}) \wedge (r = a + b)$.

5.5. $p = \big((a : \mathrm{float}, b : \mathrm{int}) \mid (a : \mathrm{int}, b : \mathrm{float}) \mid$
$(a : \mathrm{float}, b : \mathrm{float})\big) \wedge ((a + b) \notin \mathrm{float})$
`p:+` $\to r$ $(r \in \mathrm{error}) \wedge (r =$ `REALERROR`$)$.

5.6. $p \neq \big((a : \mathrm{bool}, b : \mathrm{bool}) \mid (a : \mathrm{int}, b : \mathrm{int}) \mid$
$(a : \mathrm{float}, b : \mathrm{int}) \mid (a : \mathrm{int}, b : \mathrm{float}) \mid$
$(a : \mathrm{float}, b : \mathrm{float})\big)$
`p:+` $\to r$ $(r \in \mathrm{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

6.1. $p \in \mathrm{bool}$ `p:-` $\to r$ $(r \in \mathrm{bool}) \wedge r = \neg p$.

6.2. $p \in \mathrm{int}$ `p:-` $\to r$ $(r \in \mathrm{int}) \wedge r = (-p)$.

6.3. $p \in \mathrm{float}$ `p:-` $\to r$ $(r \in \mathrm{float}) \wedge r = (-p)$.

6.4. $p = (a : \mathrm{bool}, b : \mathrm{bool})$ `p:-` $\to r$ $(r \in \mathrm{bool}) \wedge r = \big((a \vee b) \wedge \neg(a \wedge b))\big)$.

6.5. $p = (a : \mathrm{int}, b : \mathrm{int}) \wedge (a - b) \in \mathrm{int}$ `p:-` $\to r$ $(r \in \mathrm{int}) \wedge (r = a - b)$.

6.6. $p = (a : \mathrm{int}, b : \mathrm{int}) \wedge (a - b) \notin \mathrm{int}$ `p:-` $\to r$ $(r \in \mathrm{error}) \wedge (r =$ `INTERROR`$)$.

6.7. $p = \big((a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big) \wedge ((a-b) \in \text{float})$ `p:-` $\to r$ $(r \in \text{float}) \wedge (r = a - b)$.

6.8. $p = \big((a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big) \wedge ((a-b) \notin \text{float})$ `p:-` $\to r$ $(r \in \text{error}) \wedge (r =$ `REALERROR`$)$.

6.9. $p \neq \big((a:\text{bool}, b:\text{bool}) \mid (a:\text{int}, b:\text{int}) \mid (a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big) \wedge (p \notin (\text{bool} \mid \text{int} \mid \text{float}))$ `p:-` $\to r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

7.1. $p = (a:\text{bool}, b:\text{bool})$ `p:*` $\to r$ $(r \in \text{bool}) \wedge r = (a \wedge b)$.

7.2. $p = (a:\text{int}, b:\text{int}) \wedge (a \cdot b) \in \text{int}$ `p:*` $\to r$ $(r \in \text{int}) \wedge (r = a \cdot b)$.

7.3. $p = (a:\text{int}, b:\text{int}) \wedge (a \cdot b) \notin \text{int}$ `p:*` $\to r$ $(r \in \text{error}) \wedge (r =$ `INTERROR`$)$.

7.4. $p = \big((a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big) \wedge ((a \cdot b) \in \text{float})$ `p:*` $\to r$ $(r \in \text{float}) \wedge (r = a \cdot b)$.

7.5. $p = \big((a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big) \wedge ((a \cdot b) \notin \text{float})$ `p:*` $\to r$ $(r \in \text{error}) \wedge (r =$ `REALERROR`$)$.

7.6. $p \neq \big((a:\text{bool}, b:\text{bool}) \mid (a:\text{int}, b:\text{int}) \mid (a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big)$ `p:*` $\to r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

8.1. $p = \big((a:\text{int}, b:\text{int}) \mid (a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big) \wedge (p[2] \neq 0) \wedge (p[1]/p[2] \in \text{float})$ `p:/` $\to r$ $(r \in \text{float}) \wedge (r = a/b)$.

8.2. $p = \big((a:\text{int}, b:\text{int}) \mid (a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big) \wedge (p[2] \neq 0) \wedge (p[1]/p[2] \notin \text{float})$ `p:/` $\to r$ $(r \in \text{error}) \wedge (r =$ `REALERROR`$)$.

8.3. $p = \big((a:\text{int}, b:\text{int}) \mid (a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big) \wedge (p[2] = 0)$ `p:/` $\to r$ $(r \in \text{error}) \wedge (r =$ `ZERODIVIDE`$)$.

8.3. $p \neq \big((a:\text{int}, b:\text{int}) \mid (a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float}) \mid (a:\text{float}, b:\text{float})\big)$ `p:/` $\to r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

9.1. $p = (a:\text{int}, b:\text{int}) \wedge (p[2] \neq 0)$ `p:%` $\to r$ $((r = ((a/b):\text{int}, (a \mod b):\text{int})$.

9.2. $p = (a:\text{int}, b:\text{int}) \wedge (p[2] = 0)$ `p:%` $\to r$ $(r \in \text{error}) \wedge (r =$ `ZERODIVIDE`$)$.

9.3. $p \neq (a:\text{int}, b:\text{int})$ `p:%` $\to r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

10.1. $p = (a:T, b:T) \wedge (T:\text{Set})$ `p:=` $\to r$ $(r \in \text{bool}) \wedge (r = (a = b))$.

10.2. $p = (a:\text{int}, b:\text{float})$ `p:=` $\to r$ $(r \in \text{bool}) \wedge (r = (\text{ZR}(a) = b))$.

10.3. $p = (a:\text{float}, b:\text{int})$ `p:=` $\to r$ $(r \in \text{bool}) \wedge (r = (a = \text{ZR}(b)))$.

10.4. $p \neq ((a:T, b:T) \mid (a:\text{float}, b:\text{int}) \mid (a:\text{int}, b:\text{float})) \wedge (T:\text{Set})$ `p:=` $\to r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

11. Аналогичны 10.1-10.4, за исключением того, что `=` заменяется на `!=`, а выражения вида $r = (a = b)$ на $r = (a \neq b)$.

12.1. $p = (a:T, b:T) \wedge T = (\text{int} \mid \text{float})$ `p:<` $\to r$ $(r \in \text{bool}) \wedge (r = (a < b))$.

12.2. $p = (a:\text{int}, b:\text{float})$ `p:<` $\to r$ $(r \in \text{bool}) \wedge (r = (\text{ZR}(a) < b))$.

12.3. $\boxed{p = (a : \text{float}, b : \text{int})}$ `p:<` $\to r$ $\boxed{(r \in \text{bool}) \wedge (r = (a < \text{ZR}(b)))}$.

12.4. $\boxed{p = (a : \text{char}, b : \text{char})}$ `p:<` $\to r$ $\boxed{(r \in \text{bool}) \wedge (r = (\text{CharInt}(a) < \text{CharInt}(b))}$.

12.5. $\boxed{p = (a : \text{bool}, b : \text{bool})}$ `p:<` $\to r$ $\boxed{\begin{array}{l}(r \in \text{bool}) \wedge \exists (n, m : \mathbb{N}).\ \big((r = (n < m)) \wedge \\ (((a = \text{true}) \Rightarrow (n = 1)) \vee ((a = \text{false}) \Rightarrow (n = 0))) \wedge \\ (((b = \text{true}) \Rightarrow (m = 1)) \vee ((b = \text{false}) \Rightarrow (m = 0)))\big)\end{array}}$.

12.6. $\boxed{\begin{array}{l}p \neq ((a : T, b : T) \mid (a : \text{float}, b : \text{int}) \mid (a : \text{int}, b : \text{float})) \wedge \\ (T = (\text{int} \mid \text{float} \mid \text{char} \mid \text{bool}))\end{array}}$ `p:<` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

13. Аналогичны 12.1-12.6, за исключением того, что `<` заменяется на `>`, в выражениях $<$ заменяется на $>$.

14. Аналогичны 12.1-12.6, за исключением того, что `<` заменяется на `<=`, в выражениях $<$ заменяется на $\leqslant$.

15. Аналогичны 12.1-12.6, за исключением того, что `<` заменяется на `>=`, в выражениях $<$ заменяется на $\geqslant$.

16.1. $\boxed{\begin{array}{l}(p \in \text{datalist } L) \wedge (L \in \text{list Set}) \wedge \\ (\forall i : \mathbb{N}.\ (0 < i \leqslant \text{length}(L)) \Rightarrow (L[i] = \text{bool}))\end{array}}$ `p:?` $\to r$ $\boxed{(r \in \text{parlist genZ}(\text{countT}(L, p)) \wedge (r = f(L, p, 1))}$.

Где countT, genZ и f — вспомогательные функции, определённые ниже. Функция countT принимает список данных, состоящий из булевских констант, и подсчитывает количество значений true. Расчёт проводится рекурсивно: для пустого списка возвращается ноль, для списка с одним элементом true — единица, для произвольного списка l производится рекурсивный вызов функции для хвоста списка l, и к полученному значению прибавляется единица, если голова списка l имеет значение true.

$\text{countT} : \Pi\, L : \text{list Type}.\, \text{datalist } L \to \mathbb{N}$,

$$\vdash \forall (L : \text{list Set})(i : \mathbb{N})(l : \text{datalist } L).\ ((0 < i \leqslant \text{length}(L)) \wedge (L[i] = \text{bool})) \Rightarrow ((\text{countT}(\text{nil}, \text{hnil}) = 0) \wedge$$
$$\wedge (\text{countT}(\text{cons}(\text{bool}, L), \text{hcons}(\text{true}, l)) = 1 + \text{countT}(L, l)) \wedge$$
$$\wedge (\text{countT}(\text{cons}(\text{bool}, L), \text{hcons}(\text{false}, l)) = \text{countT}(L, l))).$$

Функция genZ принимает целое число n в возвращает список длины n, у которого все элементы одинаковые и имеют значение «$\mathbb{Z}$»:

$\text{genZ} : \mathbb{N} \to \text{list Set}$,

$$\vdash (\text{genZ}(0) = \text{nil}) \wedge (\text{genZ}(1) = (\mathbb{Z})) \wedge \big(\forall (n : \mathbb{N}).\ (n > 1) \Rightarrow (\text{genZ}(n) = \text{cons}(\mathbb{Z}, \text{genZ}(n - 1)))\big).$$

Функция f семантически аналогична функции «?». Она принимает список данных, состоящие из булевских констант, и возвращает параллельный список, содержащий номера позиций, на которых находятся значения true; дополнительный аргумент — целое число, которое служит для сохранения номера позиции рассматриваемого элемента при рекурсивном вызове функции. Функция f имеет следующий тип и аксиомы:

$f : \Pi\, (L : \text{list Set})(l : \text{datalist } L).\, \mathbb{Z} \to \text{parlist genZ}(\text{countT}(L, l)))$,

$$\vdash \forall n : \mathbb{Z}.\ \ (f((\text{bool}), (\text{true} : \text{bool}), n) = [n : \mathbb{Z}]) \wedge\ (f((\text{bool}), (\text{false} : \text{bool}), n) = [\,]) \wedge$$
$$\wedge\ \ \forall (L : \text{list Type})(i : \mathbb{N})(l : \text{datalist } L).\ ((0 < i \leqslant \text{length}(L)) \wedge (L[i] = \text{bool})) \Rightarrow$$
$$\Rightarrow ((f(\text{cons}(\text{bool}, L), \text{hcons}(\text{true}, l), n) = \text{phcons}(n, f(L, l, n + 1))) \wedge$$
$$\wedge (f(\text{cons}(\text{bool}, L), \text{hcons}(\text{false}, l), n) = f(L, l, n + 1))).$$

16.2. $\boxed{\begin{array}{l}(p \in \text{datalist } L) \wedge (L \in \text{list Set}) \wedge \\ \neg(\forall i : \mathbb{N}.\ (0 < i \leqslant \text{length}(L)) \Rightarrow (L[i] = \text{bool}))\end{array}}$ `p:?` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

16.3. $\boxed{(p \notin \text{datalist } L) \wedge (L \in \text{list bool})}$ `p:?` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

17.1. $(p \in \text{datalist}\, L) \wedge (L \in \text{list Set}) \wedge (\forall i : \mathbb{N}.\, (0 < i \leqslant \text{length}(L)) \Rightarrow ((p[i] \in \text{datalist}\, L_i) \wedge (L_i \in \text{list Set}))) \wedge (\forall j : \mathbb{N}.\, (1 < j \leqslant \text{length}(L)) \Rightarrow (\text{length}(L_j) = \text{length}(L_1)))$ `p:#` $\to r$ $(r \in \text{datalist}\, M) \wedge (M \in \text{list Set}) \wedge (\text{length}(M) = \text{length}(L_1)) \wedge (\forall i : \mathbb{N}.\, (0 < i \leqslant \text{length}(M)) \Rightarrow \big((r[i] \in \text{datalist}\, M_i) \wedge (M_i \in \text{list Set}) \wedge (\text{length}(M[i]) = \text{length}(L)) \wedge (\forall j : \mathbb{N}.\, (0 < j \leqslant \text{length}(L)) \Rightarrow (r[i][j] = p[j][i]))\big)$.

17.2. $\neg\big((p \in \text{datalist}\, L) \wedge (L \in \text{list Set}) \wedge (\forall i : \mathbb{N}.\, (0 < i \leqslant \text{length}(L)) \Rightarrow ((p[i] \in \text{datalist}\, L_i) \wedge (L_i \in \text{list Set}))) \wedge (\forall j : \mathbb{N}.\, (1 < j \leqslant \text{length}(L)) \Rightarrow (\text{length}(L_j) = \text{length}(L_1)))\big)$ `p:#` $\to r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

18. `p:()`$\equiv$`p:datalist`,

$(p \in T) \wedge (T : \text{Set})$ `p:()` $\to r$ $r = (p : T)$.

19. `p:[]`$\equiv$`p:parlist`,

19.1. $(p \in \text{datalist}\, L) \wedge (L \in \text{list Set})$ `p:[]` $\to r$ $(r \in \text{parlist}\, L) \wedge (\forall i : \mathbb{N}.\, (0 < i \leqslant \text{hlength}(p)) \Rightarrow (\text{phelem}(r, i) = \text{helem}(p, i)))$.

19.2. $(p \notin \text{datalist}\, L) \wedge (L \in \text{list Set})$ `p:[]` $\to r$ $r = p$.

20.1. $(b \in \text{bool}) \wedge (b = \text{true}) \wedge (p \in T) \wedge (T \in \text{Set})$ `p:<bool,b>` $\to r$ $(r = p)$.

20.2. $(b \in \text{bool}) \wedge (b = \text{false}) \wedge (p \in T) \wedge (T \in \text{Set})$ `p:<bool,b>` $\to r$ $(r = \bullet)$.

21.1. $p = (a : \text{int}, b : \text{int}) \wedge (a \leqslant b)$ `p:..` $\to r$ $(r \in \text{datalist}\, L) \wedge (L \in \text{list Set}) \wedge (\text{length}(L) = b - a + 1) \wedge (\forall i : \mathbb{N}.\, (0 < i \leqslant \text{length}(L)) \Rightarrow ((L[i] = \text{int}) \wedge (r[i] = a + i - 1)))$.

21.2. $p = (a : T_1, b : T_2) \wedge ((T_1 \neq \text{int}) \vee (T_2 \neq \text{int}))$ `p:..` $\to r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

21.3. $p = (a : \text{int}, b : \text{int}) \wedge (a > b)$ `p:..` $\to r$ $(r \in \text{error}) \wedge (r =$ `BOUNDERROR`$)$.

21.4. $p = (a : \text{int}, b : \text{int}, h : \text{int}) \wedge ((b - a) \cdot h > 0)$ `p:..` $\to r$ $(r \in \text{datalist}\, L) \wedge (L \in \text{list Set}) \wedge (\exists k : \mathbb{N}.\, (\text{length}(L) = k + 1) \wedge (((b \geqslant (a + k \cdot h)) \wedge (b < a + (k + 1) \cdot h) \wedge (a < b)) \vee ((b \leqslant (a + k \cdot h)) \wedge (b > a + (k + 1) \cdot h) \wedge (a > b))) \wedge (\forall i : \mathbb{N}.\, (0 < i \leqslant k + 1) \Rightarrow ((L[i] = \text{int}) \wedge (r[i] = a + (i - 1) \cdot h))))$.

21.5. $p = (a : \text{int}, b : \text{int}, h : \text{int}) \wedge ((b - a) \cdot h \leqslant 0))$ `p:..` $\to r$ $(r \in \text{error}) \wedge (r =$ `BOUNDERROR`$)$.

21.6. $p = (a : \text{float}, b : \text{int}, h : \text{float}) \wedge ((\text{ZR}(b) - a) \cdot h > 0)$ `p:..` $\to r$ $(r \in \text{datalist}\, L) \wedge (L \in \text{list Set}) \wedge (\exists k : \mathbb{N}.\, (\text{length}(L) = k + 1) \wedge (((\text{ZR}(b) \geqslant (a + k \cdot h)) \wedge (\text{ZR}(b) < a + (k + 1) \cdot h) \wedge (a < \text{ZR}(b))) \vee ((\text{ZR}(b) \leqslant (a + k \cdot h)) \wedge (\text{ZR}(b) > a + (k + 1) \cdot h) \wedge (a > \text{ZR}(b)))) \wedge (\forall i : \mathbb{N}.\, (0 < i \leqslant k + 1) \Rightarrow ((L[i] = \text{float}) \wedge (r[i] = a + (i - 1) \cdot h))))$.

21.7. $p = (a : \text{float}, b : \text{int}, h : \text{float}) \wedge ((\text{ZR}(b) - a) \cdot h \leqslant 0))$ `p:..` $\to r$ $(r \in \text{error}) \wedge (r =$ `BOUNDERROR`$)$.

21.8.-21.17 Аналогичны 21.6 - 21.7, за исключением того, что в них p принимает одно из следующих значений: $(a : \text{int}, b : \text{float}, c : \text{int})$, $(a : \text{int}, b : \text{float}, c : \text{float})$, $(a : \text{float}, b : \text{int}, c : \text{int})$, $(a : \text{float}, b : \text{float}, c : \text{int})$, $(a : \text{float}, b : \text{float}, c : \text{float})$; а в выражении, описывающем результат, в нужном месте используется функция ZR для правильного согласования типов.

21.18. $p \notin ((a : \text{int}, b : \text{int}, c : \text{int}) \mid (a : \text{int}, b : \text{float}, c : \text{int}) \mid (a : \text{int}, b : \text{float}, c : \text{float}) \mid (a : \text{float}, b : \text{int}, c : \text{int}) \mid (a : \text{float}, b : \text{int}, c : \text{float}) \mid (a : \text{float}, b : \text{float}, c : \text{int}) \mid (a : \text{float}, b : \text{float}, c : \text{float}))$ `p:..` $\to r$ $(r \in \text{error}) \wedge (r =$ `BASEFUNCERROR`$)$.

21.19. $\boxed{\begin{array}{l}p \in (\text{datalist}\, L) \wedge (L \in \text{list Set}) \wedge \\ (\text{length}(L) \neq 2) \wedge (\text{length}(L) \neq 3)\end{array}}$ `p:..` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

21.20. $\boxed{p \notin (\text{datalist}\, L) \wedge (L \in \text{list Set})}$ `p:..` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

22.1. $\boxed{p = (a : T, b : \text{int}) \wedge (T \in \text{Set}) \wedge (b > 0)}$ `p:dup` $\to r$ $\boxed{\begin{array}{l}(r \in \text{datalist}\, L) \wedge (L \in \text{list Set}) \wedge \\ (\text{length}(L) = b) \wedge \forall i : \mathbb{N}.\, (1 \leqslant i \leqslant b) \Rightarrow (r[i] = a)\end{array}}$.

22.2. $\boxed{p = (a : T, b : \text{int}) \wedge (T \in \text{Set}) \wedge (b \leqslant 0)}$ `p:dup` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

22.3. $\boxed{p \neq (a : T, b : \text{int}) \wedge (T \in \text{Set})}$ `p:dup` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

23.1. $\boxed{p \in \text{bool}}$ `p:int` $\to r$ $\boxed{(r \in \text{int}) \wedge (((p = \text{true}) \wedge (r = 1)) \vee ((p = \text{false}) \wedge (r = 0)))}$.

23.2. $\boxed{p \in \text{char}}$ `p:int` $\to r$ $\boxed{(r \in \text{int}) \wedge (r = \text{CharInt}(p))}$.

23.3. $\boxed{p \in \text{float}}$ `p:int` $\to r$ $\boxed{(r \in \text{int}) \wedge (r = \text{RZ}(p))}$.

23.4. $\boxed{p \in \text{int}}$ `p:int` $\to r$ $\boxed{(r \in \text{int}) \wedge (r = p)}$.

23.5. $\boxed{p \notin (\text{bool} \mid \text{char} \mid \text{float} \mid \text{int})}$ `p:int` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

24.1. $\boxed{p \in \text{bool}}$ `p:float` $\to r$ $\boxed{(r \in \text{float}) \wedge (((p = \text{true}) \wedge (r = 1)) \vee ((p = \text{false}) \wedge (r = 0)))}$.

24.2. $\boxed{p \in \text{char}}$ `p:float` $\to r$ $\boxed{(r \in \text{float}) \wedge (r = \text{ZR}(\text{CharInt}(p)))}$.

24.3. $\boxed{p \in \text{float}}$ `p:float` $\to r$ $\boxed{(r \in \text{float}) \wedge (r = p)}$.

24.4. $\boxed{p \in \text{int}}$ `p:float` $\to r$ $\boxed{(r \in \text{float}) \wedge (r = \text{ZR}(p))}$.

24.5. $\boxed{p \notin (\text{bool} \mid \text{char} \mid \text{float} \mid \text{int})}$ `p:float` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

25.1. $\boxed{(p \in \text{int}) \wedge (\exists c : \text{char}.\, \text{CharInt}(c) = p)}$ `p:char` $\to r$ $\boxed{(r \in \text{char}) \wedge (r = \text{IntChar}(p))}$.

25.2. $\boxed{(p \in \text{int}) \wedge \neg(\exists c : \text{char}.\, \text{CharInt}(c) = p)}$ `p:char` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

25.3. $\boxed{(p \in \text{char})}$ `p:char` $\to r$ $\boxed{(r \in \text{char}) \wedge (r = p)}$.

25.4. $\boxed{p \notin (\text{int} \mid \text{char})}$ `p:char` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

26.1. $\boxed{(p \in (\text{int} \mid \text{float})) \wedge (p \neq 0)}$ `p:bool` $\to r$ $\boxed{(r \in \text{bool}) \wedge (r = \text{true})}$.

26.2. $\boxed{(p \in (\text{int} \mid \text{float})) \wedge (p = 0)}$ `p:bool` $\to r$ $\boxed{(r \in \text{bool}) \wedge (r = \text{false})}$.

26.3. $\boxed{p \in \text{bool}}$ `p:bool` $\to r$ $\boxed{(r \in \text{bool}) \wedge (r = p)}$.

26.4. $\boxed{p \notin (\text{int} \mid \text{float} \mid \text{bool})}$ `p:bool` $\to r$ $\boxed{(r \in \text{error}) \wedge (r = \texttt{BASEFUNCERROR})}$.

27. $\boxed{(p \in T) \wedge (T \in \text{Set})}$ `p:signal` $\to r$ $\boxed{(r \in \text{signal}) \wedge (r = \bullet)}$.

Замечание. Параллельный список не может быть аргументом функции, так как по правилам преобразования, оператор интерпретации применяет функцию к каждому элементу списка, однако, функция может возвращать параллельный список. Задержанный список также не может быть аргументом функции, так как он должен раскрыться на операторе интерпретации.

Приложение E
Примеры верификации рекурсивных функций

Вычисление частного и остатка от деления целых чисел. Рассмотрим специфику доказательства корректности программы для вычисления частного и остатка от деления целых чисел. Данный пример рассматривается в работе [1] при доказательстве корректности императивной программы. Ниже приведён код программы вычисления частного и остатка от деления целых чисел x и y на языке Пифагор:

```
DIV << funcdef arg {
  x<<arg:1;  y<<arg:2;
  (x,y,0,x):div_rec >> return
}

div_rec << funcdef arg {
  x<<arg:1; y<<arg:2; q1<<arg:3; r1<<arg:4;
  ({(x,y,(q1,1):+,(r1,y):-):div_rec},(q1,r1)):
  [((y,r1):[<=, >]):?]:. >> return
}
```

Основная функция `DIV` в качестве входного значения принимает список из двух целых чисел x и y и вызывает рекурсивную функцию `div_rec`, которая вычисляет частное и остаток от деления. Исходя из условия задачи, тройка Хоара для функции `DIV` будет иметь вид (соответствующий ей ИГР приведён на рисунке E.1а):

$$\boxed{(x,\ y \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0)}\ \texttt{(x,y):DIV}\ \rightarrow (q, r)\ \boxed{(x = y \cdot q + r) \wedge (r < y)}\,.$$

Для упрощения изложения присваивание идентификаторов x и y с помощью функции выбора элемента из списка упускается, и аргумент arg сразу представляется в виде списка (x, y).

Вначале докажем корректность функции `DIV`, считая функцию `div_rec` корректной, то есть соответствующей следующей спецификации, записанной в виде тройки Хоара:

$$\boxed{\begin{array}{l}(x,\ y,\ q_1,\ r_1 \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0) \wedge \\ (q_1 \geqslant 0) \wedge (r_1 \geqslant 0) \wedge (x = y \cdot q_1 + r_1)\end{array}}\ \texttt{(x,y,q1,r1):div\_rec}\ \rightarrow (q, r)\ \boxed{\begin{array}{l}(q, r \in \text{int}) \wedge (q \geqslant 0) \wedge (r \geqslant 0) \wedge \\ (x = y \cdot q + r) \wedge (r < y)\end{array}}\,, \quad \text{(E.1)}$$

которая считается теоремой и может быть использована при применении правила прямого прослеживания к тройке Хоара программы `DIV`. Ещё одна тройка $\mathfrak{W}$ функции `div_rec` будет соответствовать некорректным входным аргументам, её предусловие является отрицанием предусловия тройки (E.1) (см. замечание 2 из раздела 2.3.3). Если входные аргументы для функции `div_rec` некорректны, то программа `DIV` сразу же считается некорректной.

Учитывая, что входной аргумент arg преобразуется по правилу эквивалентного преобразования для функции выбора элемента из списка, в функции `DIV` автоматическую разметку получают все дуги, кроме выходной дуги оператора интерпретации, осуществляющего

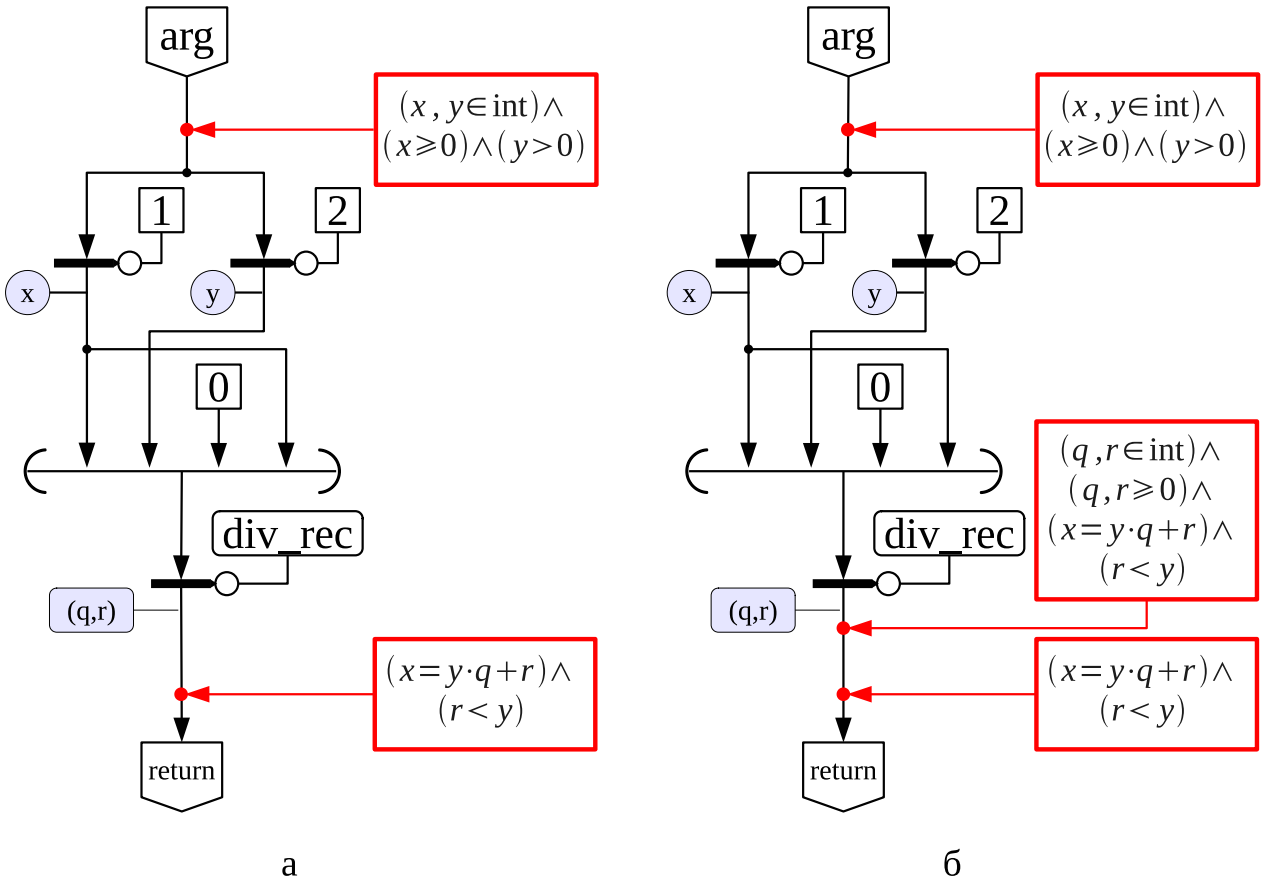


Рисунок Е.1 — Преобразование ИГР функции `DIV`: а — исходный ИГР, б — ИГР после преобразования по правилу прямого прослеживания

применений функции `div_rec`. Проверим выполнение условия (2.4). Функция `DIV` передаёт функции `div_rec` список аргументов `(x, y, 0, x)`, тогда в предусловии функции `div_rec` переменные `q1` и `r1` заменяются на `0` и `x` соответственно. Получаем выражение:

$$P_{DIV}(x,y) \Rightarrow P_{div\_rec}(x,y,0,x) \equiv ((x,\ y \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0)) \Rightarrow$$
$$\Rightarrow ((x,\ y, 0 \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0) \wedge (0 \geqslant 0) \wedge (x \geqslant 0) \wedge (x = y \cdot 0 + x)) \equiv$$
$$\equiv ((x,\ y \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0)) \Rightarrow ((x,\ y \in \text{int}) \wedge (y > 0) \wedge (x \geqslant 0) \wedge$$
$$\wedge (x = x)) \equiv \text{true}.$$

Таким образом, тройка Хоара (Е.1) может быть использована в преобразовании тройки для функции `DIV` по правилу прямого прослеживания (2.1).

Отметим, что предусловие тройки $\mathfrak{W}$ невыводимо из предусловия программы `DIV`, поскольку оно является отрицанием предусловия тройки (Е.1), поэтому тройка $\mathfrak{W}$ отбрасывается.

Предусловие преобразованной тройки Хоара для функции `DIV` на основе теоремы (Е.1)

будет иметь вид:

$$P_{DIV} \wedge P_{div\_rec} \wedge Q_{div\_rec} \equiv$$
$$\equiv (x,\ y \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0) \wedge (q, r \in \text{int}) \wedge (q,\ r \geqslant 0) \wedge (x = y \cdot q + r) \wedge (r < y).$$

Преобразованный ИГР функции `DIV` по правилу прямого прослеживания на основе теоремы (2.4) изображён на рисунке Е.1б. Этот граф является полностью размеченным и в результате свёртки даёт тройку Хоара с «пустой» программой, которую, в свою очередь, можно преобразовать по правилу (2.2) в формулу

$$(x,\ y,\ q,\ r \in \text{int}) \wedge (x,\ q,\ r \geqslant 0) \wedge (y > 0) \wedge (x = y \cdot q + r) \wedge (r < y) \Rightarrow$$
$$\Rightarrow (x = y \cdot q + r) \wedge (r < y).$$

Очевидно, что полученная формула истинна, а значит, корректна и программа `DIV` (при условии корректности `div_rec`).

Докажем корректность рекурсивной функции `div_rec` с тройкой Хоара (Е.1). Для упрощения изложения присваивание идентификаторов элементам списка входного аргумента упускается и arg сразу представляется в виде списка (x, y, q_1, p_1). Информационный граф функции `div_rec` приведён на рисунке Е.2а.

Выделим в коде программы несколько элементов.

1. `(x,y,q1,p1)` — «текущий аргумент». Входной аргумент функции, записанный в виде списка.
2. `(x,y,(q1,1):+,(r1,y):-)` — «рекурсивный аргумент». Выражение, определяющее аргумент для рекурсивного вызова функции.
3. `(y,r1):<=` — «условие рекурсивного вызова». Если значение указанного выражения истинно, то начинает выполняться ветвь программы, приводящая к рекурсивному вызову функции `div_rec`.
4. `(y,r1):>` — «условие завершения функции». Истинность данного выражения приводит к выполнению участка кода, не содержащего рекурсивный вызов функции. Данное выражение истинно тогда и только тогда, когда ложно условие рекурсивного вызова. Следовательно, в этом случае функция гарантированно завершиться (в программе отсутствуют другие рекурсивные вызовы функций).

Выполнение функции `div_rec` начинается с того, что `y` и `r1` входного аргумента формируют список `(y,r1)`. К этому списку применяются функции «<=» и «>». Это встроенные функции, их аксиомы одинаковые и отличаются лишь знаком операции. Они приведены

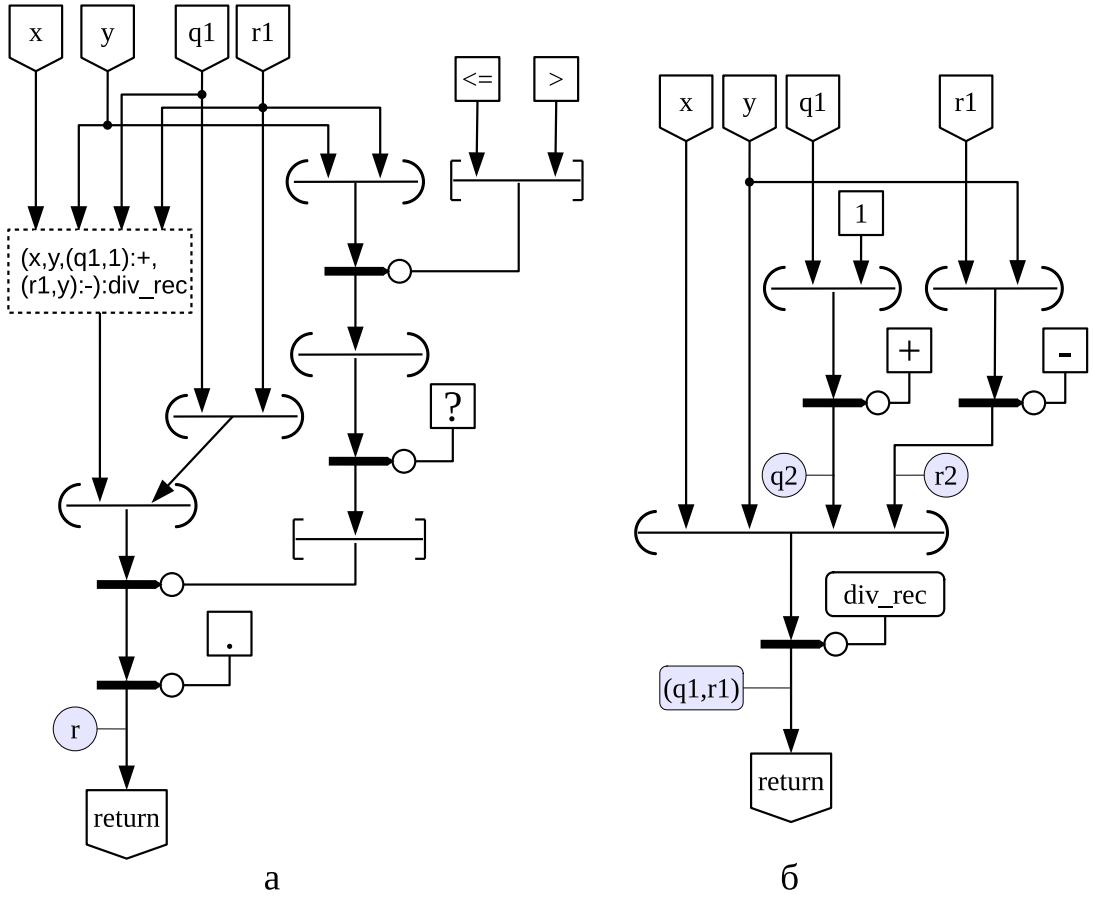


Рисунок Е.2 — Информационные графы функции `div_rec`: а — исходный граф, б — граф с рекурсивной ветвью, полученный в результате расщепления

в приложении Д (аксиомы 14.1–14.6 и 13.1–13.6 соответственно). Условие (2.4) выполнено только для первой аксиомы 14.1. При применении правила прямого прослеживания на её основе получаются формулы, описывающие «условие рекурсивного вызова» и «условие завершения функции»: $(y \leqslant r_1)$ и $(y > r_1)$ соответственно. Значения этих формул — булевские константы, они формируют список из двух элементов, который является входным аргументом функции «?». Согласно своей семантике, функция «?» возвращает порядковый номер того элемента входного списка, чьё значение равно true. Так как «условие рекурсивного вызова» и «условие завершения функции» взаимоисключающие, функция «?» возвращает либо «1», либо «2». Полученная константа используется как функция выбора элемента из списка `({(x,y,(q1,1):+,(r1,y):-):div_rec},(q1,r1))`. Это способ, с помощью которого в языке Пифагор реализуется условный выбор. В данном случае выбирается путь выполнения программы: ветвь с рекурсивным вызовом функции или ветвь без рекурсии, приводящая к завершению программы.

С помощью операции расщепления имеющийся ИГР разделяется на два. Получаем,

что исходная тройка Хоара (Е.1) будет истинной, если истинны две следующие тройки:

$$\boxed{\begin{array}{l}(x,\ y,\ q_1,\ r_1 \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0) \wedge \\ (q_1 \geqslant 0) \wedge (r_1 \geqslant 0) \wedge (x = y \cdot q_1 + r_1) \wedge (y > r_1)\end{array}} \texttt{(q1,r1):.} \rightarrow (q, r) \boxed{\begin{array}{l}(q, r \in \text{int}) \wedge (q \geqslant 0) \wedge (r \geqslant 0) \wedge \\ (x = y \cdot q + r) \wedge (r < y)\end{array}}, \tag{E.2}$$

$$\boxed{\begin{array}{l}(x,\ y,\ q_1,\ r_1 \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0) \wedge \\ (q_1 \geqslant 0) \wedge (r_1 \geqslant 0) \wedge (x = y \cdot q_1 + r_1) \wedge (y \leqslant r_1)\end{array}} \texttt{prog} \rightarrow (q, r) \boxed{\begin{array}{l}(q, r \in \text{int}) \wedge (q \geqslant 0) \wedge (r \geqslant 0) \wedge \\ (x = y \cdot q + r) \wedge (r < y)\end{array}}, \tag{E.3}$$

где `prog` соответствует код «`{(x,y,(q1,1):+,(r1,y):-):div_rec}:.`». Предусловие первой тройки получается конъюнкцией предусловия программы `div_rec` и «условия завершения функции», а в коде вместо «точки ветвления» подставляется ветвь без рекурсии. Вторая тройка получается аналогично: предусловие — конъюнкция предусловия программы и «условия рекурсивного вызова», а в коде вместо «точки ветвления» подставляется ветвь с рекурсией.

Докажем корректность программы `div_rec` с помощью индукции, которая имеет место при условии, что функция завершается (индукция проводится по значениям ограничивающей функции).

База индукции. Требуется доказать корректность всех ветвей программ, для которых выполнено «условие завершения» $y > r_1$ и не происходит рекурсивного вызова программы. Это соответствует доказательству корректности тройки Хоара (Е.2). Функция «`.`» не меняет своего аргумента, поэтому, учитывая, что $(q = q_1)$ и $(r = r_1)$, сразу можно применить правило преобразования тройки в формулу:

$$\begin{array}{l}((x,\ y,\ q_1,\ r_1 \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0) \wedge (q_1 \geqslant 0) \wedge (r_1 \geqslant 0) \wedge \\ \qquad \wedge (x = y \cdot q_1 + r_1) \wedge (y > r_1)) \Rightarrow ((q_1, r_1 \in \text{int}) \wedge (q_1 \geqslant 0) \wedge (r_1 \geqslant 0) \wedge \\ \qquad\qquad \wedge (x = y \cdot q_1 + r_1) \wedge (r_1 < y))\,.\end{array}$$

Очевидно, что формула тождественно истина.

Шаг индукции. Предполагаем, что тройка (Е.1) истинна для всех рекурсивных вызовов функции `divrec`, если «рекурсивный аргумент» удовлетворяет предусловию. Требуется показать, что корректны все ветви, для которых выполнено «условие рекурсивного вызова», содержащие рекурсивный вызовы функции. В случае `divrec` это эквивалентно доказательству корректности тройки (Е.3).

В данной тройке первой выполняется функция «`.`», которая раскрывает задержанный список, предусловие и постусловие не изменяются. Информационный граф второй тройки, после снятия задержки, приведён на рисунке Е.2б. Далее по правилам прямого прослеживания на основе аксиом функций «`-`» и «`+`» (аксиомы 5.1–5.6, 6.1–6.9 приложения Д), применяемых непосредственно к аргументу, выражения `(q,1):+` и `(r,y):-` заменяются значениями $q_2 = (q_1 + 1)$ и $r_2 = (r_1 - y)$ соответственно, а тройка Хоара преобразуется к следующему

виду:

$$\boxed{\begin{array}{l}(x,\ y,\ q_1,\ r_1 \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0) \wedge \\ (q_1 \geqslant 0) \wedge (r_1 \geqslant 0) \wedge (x = y \cdot q_1 + r_1) \wedge \\ (y \leqslant r_1) \wedge (q_2 = q_1 + 1) \wedge (r_2 = r_1 - y)\end{array}} \texttt{(x,y,q2,r2):div\_rec} \to (q, r) \boxed{\begin{array}{l}(q, r \in \text{int}) \wedge (q \geqslant 0) \wedge (r \geqslant 0) \wedge \\ (x = y \cdot q + r) \wedge (r < y)\end{array}}. \quad \text{(E.4)}$$

Теперь непосредственно к аргументу применяется рекурсивный вызов функции `div_rec`, и «рекурсивный аргумент» имеет вид `(x,y,q2,r2)`.

Покажем, что «рекурсивный аргумент» удовлетворяет предусловию (E.1). Для этого требуется проверить истинность условия (2.4), в котором в качестве предусловия функции $P(x)$ берётся предусловие тройки (E.4), а в качестве предусловия $P_i(x)$ — предусловие P_{div_rec} тройки (E.1), применённое к «рекурсивному аргументу» (x, y, q_2, r_2):

$$\big(P_{div\_rec}(x, y, q_1, r_1) \wedge (y \leqslant r_1) \wedge (q_2 = q_1 + 1) \wedge (r_2 = r_1 - y)\big) \Rightarrow P_{div\_rec}(x, y, q_2, r_2).$$

Данная формула истинна, так как из $(q_1 \geqslant 0)$ и $(q_2 = q_1 + 1)$ следует, что $(q_2 \geqslant 0)$; из $(y \leqslant r_1)$ и $(r_2 = r_1 - y)$ следует, что $(r_2 \geqslant 0)$; и $x = y \cdot q_1 + r_1 = y \cdot (q_1 + 1) - y + r_1 = y \cdot q_2 + r_2$.

Тогда по индуктивному предположению будет верна тройка Хоара для функции `div_rec`, применённая к «рекурсивному аргументу», то есть в предусловии тройки (E.1) `q1` и `r1` заменяются на `q2` и `r2` соответственно:

$$\boxed{\begin{array}{l}(x,\ y,\ q_2,\ r_2 \in \text{int}) \wedge (x \geqslant 0) \wedge (y > 0) \wedge \\ (q_2 \geqslant 0) \wedge (r_2 \geqslant 0) \wedge (x = y \cdot q_2 + r_2)\end{array}} \texttt{(x,y,q2,r2):div\_rec} \to (q, r) \boxed{\begin{array}{l}(q, r \in \text{int}) \wedge (q \geqslant 0) \wedge (r \geqslant 0) \wedge \\ (x = y \cdot q + r) \wedge (r < y)\end{array}}.$$

Используем данную тройку в качестве теоремы для доказательства корректности тройки (E.4) с помощью правила прямого прослеживания, считая функцию `div_rec` нерекурсивной. В результате такого преобразования получается тройка с «пустой программой». После упрощения её предусловия и применения правила преобразования тройки в формулу получаем следующую формулу:

$$\begin{aligned}\big(P_{div\_rec}(x, y, q_1, r_1) \wedge (y \leqslant r_1) \wedge (q, r \in \text{int}) \wedge (q, r \geqslant 0) \wedge (x = y \cdot q + r) \wedge (r < y)\big) \Rightarrow \\ \Rightarrow ((q, r \in \text{int}) \wedge (q, r \geqslant 0) \wedge (x = y \cdot q + r) \wedge (r < y))\end{aligned}$$

Полученная формула истинна, а значит, частичная корректность функции `div_rec` доказана.

Докажем завершение функции `div_rec`. Определим фундированное множество $S \equiv \{x : \mathbb{Z} \mid x \geqslant 0\}$, в качестве порядка возьмём отношение «меньше» $<$. Зададим ограничивающую функцию для `div_rec`:

$$\varphi(x, y, q_1, r_1) = x - (y \cdot q_1).$$

Заданная таким образом целочисленная функция целочисленных аргументов определена для всех (x, y, q_1, r_1), удовлетворяющих предусловию функции `div_rec`. Поскольку $(x, q_1, r_1 \geqslant 0)$, $(y > 0)$ и $(x = y \cdot q_1 + r)$, то значения $\varphi(x, y, q_1, r_1) \geqslant 0$ и, следовательно, принадлежат S.

Покажем, что значение ограничивающей функции для «текущего аргумента» будет больше, чем для «рекурсивного». Возьмём аргумент (x, y, q_1, r_1), удовлетворяющий (E.1), для которого значение ограничивающей функции $\varphi(x, y, q_1, r_1) = N$. Значение ограничивающей функции для «рекурсивного аргумента» (x, y, q_2, r_2) равно:

$$\varphi(x, y, q_2, r_2) = \varphi(x, y, q_1 + 1, r_1 - y) = x - y \cdot (q_1 + 1) = \\ = (x - (y \cdot q_1)) - y = \varphi(x, y, q_1, r_1) - y = N - y.$$

Получаем $(N - y) < N$, так как $(y > 0)$. Значит, функция `div_rec` завершается для любого аргумента, удовлетворяющего предусловию.

Вычисление факториала целого неотрицательного числа. Докажем корректность рекурсивной функции `fact`, вычисляющей факториал целого неотрицательного числа x. Код функции `fact` на языке Пифагор:

```
fact << funcdef x {
    fl << ((x,1):[<=,>]):?;
    act << (1,
          { (x, (x,1):-:fact ):*  } );
    return << act:fl:.;
}
```

Пользователь задаёт для функции следующую тройку Хоара (дописывает пред- и постусловие, которые обозначим $P(x)$ и $Q(r)$ соответственно):

$$\boxed{(x \in \text{int}) \wedge (x \geqslant 0) \wedge \left(\left(\prod_{i=1}^{x} i\right) \leqslant \text{MaxInt}\right)} \ \texttt{x:fact} \ \rightarrow r \ \boxed{((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee ((r = 1) \wedge (x = 0))}, \qquad \text{(E.5)}$$

где MaxInt — максимальное целое, которое допускает тип int. Функция Π определена следующим образом:

$$\Pi : \mathbb{Z} \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z},$$
$$\vdash \forall f : (\mathbb{Z} \rightarrow \mathbb{Z}).\ (\Pi(0, f) = 0) \wedge (\Pi(1, f) = f(1)) \wedge \\ \wedge (\forall n : \mathbb{Z}.\ (n > 1) \Rightarrow (\Pi(n + 1, f) = f(n + 1) \cdot \Pi(n, f))),$$
$$\Pi(n, f) \equiv \prod_{i=1}^{n} f(i).$$

Тройка Хоара (E.5) соответствует следующей спецификации программы: если входное значение x — целое число, большее или равное нулю, и произведение чисел от 1 до x не превышает максимального целого, которое допускает тип int, то после выполнения функция `fact` возвращает результат перемножения чисел от 1 до x, если $x > 0$ или 1, если $x = 0$.

Рассмотрим процесс вычисления функции `fact` (рисунок E.3а). При $x = 0$ или 1 должно быть возвращено значение 1, при $x > 1$ должен быть совершен рекурсивный вызов `fact` с аргументом $(x - 1)$. Проверка истинности одного из условий $(x \leqslant 1)$ или $(x > 1)$ выполняется в правой ветви, содержащей функцию «?». Параллельно, в левой ветви формируется список

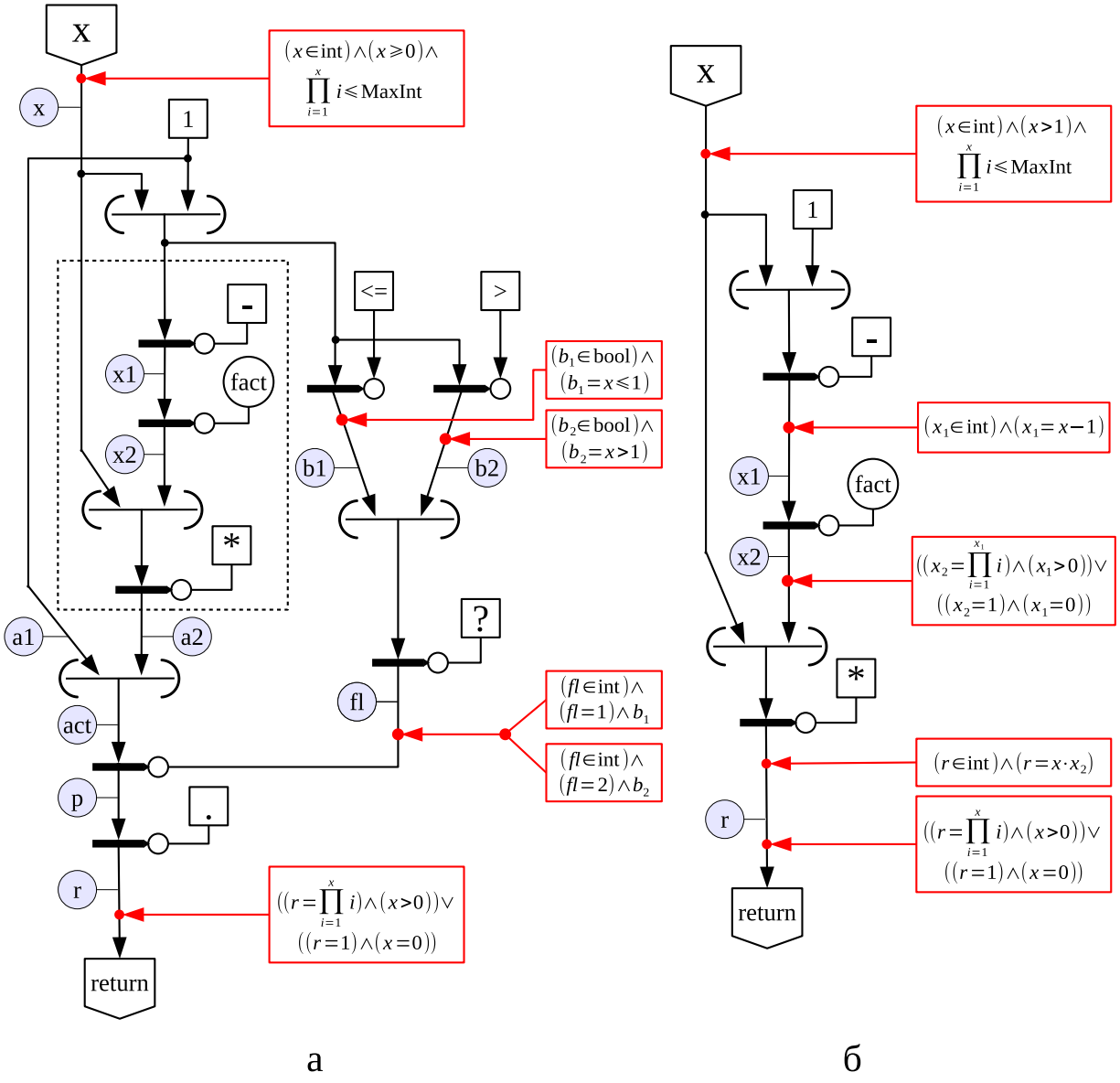


Рисунок Е.3 — Информационный граф тройки Хоара для функции `fact`

act из двух элементов: «1» и «задержанный список», содержащий подграф с рекурсивным вызовом функции `fact`. Операторы задержанного списка выполняются только после снятия задержки, даже если все аргументы готовы. Выбор одного из элементов в списке *act* осуществляется в зависимости от того, какое из условий $x \leqslant 1$ или $x > 1$ верно. Во втором случае выбирается задержанный список, с него снимается задержка функцией «.», и происходит рекурсивный вызов. В случае $x \leqslant 1$ из списка *act* выбирается константа «1», а выполнение операторов задержанного списка вообще не происходит.

Для доказательства корректности программы необходимо доказать истинность тройки (Е.5). Выполнение функции `fact` начинается с того, что входной аргумент `x` и константа «1» формируют список `(x,1)`. К этому списку применяется параллельный список `[<=,>]`.

Производится преобразование по правилам эквивалентных преобразований (правила 3.2 и 4.1 из таблицы 2.1). Полученный информационный граф функции `fact` приведён на рисунке Е.3а. Изначально у графа размечены только входная и выходная дуги, в серых кружках указаны идентификаторы дуг, все идентификаторы должны быть различны, задержанный список выделен пунктиром.

После преобразований получаем, что к списку `(x,1)` применяются функции «<=» и «>». Это встроенные функции, их аксиомы одинаковые и отличаются лишь знаком операции (соответственно аксиомы 14.1–14.6 и 13.1–13.6 приложения Д).

Перед применением правила прямого прослеживания необходимо выбрать те аксиомы, у которых предусловие может следовать из предусловия тройки Хоара (Е.5). Все остальные аксиомы отбрасываются. Формируем формулы, описывающие «условие невыводимости» (предусловия) аксиом 14.1-14.6 из предусловия тройки (Е.5), и проверяем их истинность. Аксиомы, которым соответствуют истинные формулы — отбрасываем. После согласования обозначений аргументов аксиом с рассматриваемой тройкой (Е.5) и введения идентификатора b_1 для результата выполнения функции «<=», получаем что $a = x$, а $b = 1$, тогда условия невыводимости опишутся следующими формулами:

1. $\neg\big(P(x) \Rightarrow (a, b \in \text{int} \mid \text{float})\big)$,
2. $\neg\big(P(x) \Rightarrow (a \in \text{int}) \wedge (b \in \text{float})\big)$,
3. $\neg\big(P(x) \Rightarrow (a \in \text{float}) \wedge (b \in \text{int})\big)$,
4. $\neg\big(P(x) \Rightarrow (a, b \in \text{char})\big)$,
5. $\neg\big(P(x) \Rightarrow (a, b \in \text{bool})\big)$,
6. $\neg\big(P(x) \Rightarrow (a, b \notin (\text{int} \mid \text{float} \mid \text{char} \mid \text{bool})) \vee$
$\vee \neg((a \in \text{float}) \wedge (b \in \text{int})) \wedge \neg((a \in \text{int}) \wedge (b \in \text{float}))\big)$,

Так как $(x \in \text{int})$ согласно предусловию и $(1 \in \text{int})$ по определению, то первая формула является тождественно ложной, а остальные — тождественно истинными, поэтому все аксиомы, кроме первой, отбрасываются, так как выполнение программы не сможет пойти по этим путям.

В результате применения «правила прямого прослеживания» к тройке (Е.5) на основе аксиомы 14.1, получаем следующую тройку Хоара:

$\boxed{P(x) \wedge (b_1 \in \text{bool}) \wedge (b_1 = (x \leqslant 1))}$ `( 1, (x,(x,1):-:fact):* ):[(b1,(x,1):>):?]:.` $\to r$ $\boxed{Q(r)}$,

Далее аналогичным образом применяем «правило прямого прослеживания» на основе аксиом для функции «>», получаем тройку:

$\boxed{P(x) \wedge (b_1 \in \text{bool}) \wedge (b_1 = (x \leqslant 1)) \wedge (b_2 = (x > 1))}$ `( 1, (x,(x,1):-:fact):* ):[(b1,b2):?]:.` $\to r$ $\boxed{Q(r)}$.

Выходная дуга списка `(b1,b2)` размечается автоматически, и к разметке становится готова дуга fl. Условия $(x \leqslant 1)$ и $(x > 1)$ являются взаимоисключающими, поэтому $(b1, b2)$ может принимать только два значения: (true, false) или (false, true). Используем правило прямого прослеживания на основе аксиомы функции «?» (условию применимости удовлетворяет только одна аксиома 16.1, приложение Д):

$$\boxed{\begin{array}{l} P(x) \wedge (b_1 \in \text{bool}) \wedge (b_1 = (x \leqslant 1)) \wedge (b_2 = (x > 1)) \wedge \\ (fl \in \text{int}) \wedge \big(((fl = 1) \wedge (b_1 = \text{true})) \vee ((fl = 2) \wedge (b_2 = \text{true}))\big) \end{array}} \texttt{(1, (x,(x,1):-:fact):*):fl:.} \rightarrow r \boxed{Q(r)},$$

ИГР, полученной тройки, приведён на рисунке Е.3а.

После этого к разметке готова дуга p. Так как на функциональный вход поступает целочисленная константа, то можно провести расщепление графа на два, им соответствуют две следующие тройки:

$$\boxed{P(x) \wedge (x \leqslant 1)} \texttt{x:.} \rightarrow r \boxed{Q(r)}, \tag{E.6}$$

$$\boxed{P(x) \wedge (x > 1)} \texttt{\{(x, (x,1):-:fact):*\}:.} \rightarrow r \boxed{Q(r)}. \tag{E.7}$$

В тройке (Е.6) к аргументу применяется функция «.», которая не меняет своего аргумента, поэтому тройка преобразуется в тройку с «пустой» программой:

$$\boxed{P(x) \wedge (x \leqslant 1) \wedge (r = 1)} \quad \boxed{Q(r)},$$

её можно преобразовать в формулу по правилу (2.2):

$$\left((x \in \text{int}) \wedge (x \geqslant 0) \wedge (\prod_{i=1}^{x} i \leqslant \text{MaxInt}) \wedge (x \leqslant 1) \wedge (r = 1)\right) \Rightarrow$$
$$\Rightarrow \left(((r = \prod_{i=1}^{x} i) \wedge (x > 0)) \vee ((r = 1) \wedge (x = 0))\right).$$

Очевидно, что данная формула истинна, так как из $(x \geqslant 0)$ и $(x \leqslant 1)$, следует, что $x \in \{0, 1\}$, а $(r = 1)$.

Теперь рассмотрим тройку (Е.7). Функция «.» применяется к задержанному списку и «раскрывает» его. Полученный после снятия задержки граф приведён на рисунке Е.3б. Теперь функция «-» — ближайшая к входному аргументу, поэтому выполняется первой. При применении «правила прямого прослеживания» на основе её аксиом к тройке (Е.7) получаем (применима только одна аксиома 6.5, приложение Д):

$$\boxed{P(x) \wedge (x > 1) \wedge (x_1 \in \text{int}) \wedge (x_1 = x - 1)} \text{ (x, x1:fact):*} \rightarrow r \boxed{Q(r)}. \tag{E.8}$$

Далее должен выполняться рекурсивный вызов функции `fact`. Предположим, что программа завершается, и функция корректна для всех рекурсивных аргументов, удовлетворяющих предусловию. Проверим, что рекурсивный аргумент x_1 удовлетворяет предусловию

$P(x)$ исходной тройки (Е.5). Обозначим $P_2(x)$ предусловие тройки (Е.8), тогда:

$$P_2(x) \Rightarrow P(x_1) \equiv \big(P(x) \wedge (x > 1) \wedge (x_1 \in \text{int}) \wedge (x_1 = x - 1)\big) \Rightarrow$$
$$\Rightarrow \left((x_1 \in \text{int}) \wedge (x_1 \geqslant 0) \wedge (\prod_{i=1}^{x_1} i \leqslant \text{MaxInt})\right).$$

Исходя из того что $(x > 1)$, а $(x_1 = x - 1)$, то $(x_1 > 0)$. Если $(\prod_{i=1}^{x} i \leqslant \text{MaxInt})$, то $(\prod_{i=1}^{x_1} i = \prod_{i=1}^{(x-1)} i < \text{MaxInt})$. Значит формула тождественно истинна, и «аргумент рекурсивного вызова» всегда удовлетворяет предусловию функции `fact`. В противном случае программа сразу бы считалась некорректной.

В этом случае, тройка (Е.8) может быть преобразована по правилу прямого прослеживания на основе теоремы (Е.5). Подставим в (Е.5) в качестве входного аргумента «рекурсивный аргумент» x_1, а идентификатор результата заменим на x_2 и используем данную тройку в качестве теоремы при применении «правила прямого прослеживания» к тройке (Е.8), получаем:

$$\boxed{\begin{array}{l} P(x) \wedge (x > 1) \wedge (x_1, x_2 \in \text{int}) \wedge (x_1 = x - 1) \wedge \\ ((x_2 = \prod_{i=1}^{x_1} i) \wedge (x_1 > 0)) \vee ((x_2 = 1) \wedge (x_1 = 0)) \end{array}} \ \text{(x, x2):*} \rightarrow r \ \boxed{Q(r)}.$$

В коде данной тройки осталась только функция умножения «$*$». Применяя «правило прямого прослеживания» на основе аксиом для этой функции, получаем тройку с «пустой» программой (применима только одна аксиома 7.2, приложение Д):

$$\boxed{P(x) \wedge (x \geqslant 1) \wedge (x_2 \in \text{int}) \wedge (x_2 = \prod_{i=1}^{(x-1)} i) \wedge (r = x \cdot x_2)} \quad \boxed{Q(r)}.$$

Далее применяем «правило преобразования тройки в формулу» (2.2), и после упрощения получаем:

$$\Big(P(x) \wedge (x > 1) \wedge (x_2 \in \text{int}) \wedge (x_2 = \prod_{i=1}^{(x-1)} i) \wedge (r = x \cdot x_2)\Big) \Rightarrow Q(r) \equiv$$
$$\equiv \Big((r = x \cdot \prod_{i=1}^{(x-1)} i) \Rightarrow (r = \prod_{i=1}^{x} i)\Big).$$

Полученная формула истинна, а значит, истинна исходная тройка Хоара (Е.5), из чего следует частичная корректность программы.

Докажем завершение программы `fact`. Определим фундированное множество $S \equiv \{x : \mathbb{Z} \mid x \geqslant 0\}$, в качестве порядка возьмём отношение «меньше» $<$. Зададим ограничивающую функцию для `fact`:

$$\varphi(x) = x.$$

Функция определена для всех x, удовлетворяющих предусловию $P(x)$, так как $(x \geqslant 0)$ следовательно, значения $\varphi(x)$ принадлежат S.

Покажем, что значение ограничивающей функции для «текущего аргумента» будет больше, чем для «рекурсивного»:

$$\varphi(x_2) = \varphi(x-1) < \varphi(x).$$

Значит, функция `fact` завершается для любого аргумента, удовлетворяющего предусловию.

Таким образом, тотальная корректность функции `fact` доказана.

Приложение Ж

Частный случай рекурсии с явно выраженным рекуррентным соотношением

Рассмотренный в разделе 3.1.2 метод доказательства завершения рекурсии применим для любой рекурсивной программы на языке Пифагор. Однако его недостатком является необходимость подбора удачной ограничивающей функции и фундированного множества, что является нетривиальной задачей. Даже при выборе подходящей функции и фундированного множества доказательство истинности получающихся формул будет базироваться на теоремах, лежащих в основе алгоритма задачи. Это означает, что доказательство потребует описания (и доказательства) этих теорем на языке спецификации. Это замечание справедливо и при доказательстве частичной корректности.

В некоторых случаях доказательство корректности программы на языке Пифагор можно упростить. Для этого рассмотрим частный случай рекурсии, когда удаётся явно задать рекуррентное соотношение [152].

Пусть с помощью функции $T(n)$ требуется вычислить значения некоторой последовательности t_n, где n — целое положительное число. Функция $T(n)$ задается непосредственно в виде числовых значений для некоторого конечного множества начальных значений аргумента n, $1 \leqslant n \leqslant m$. Задается метод или формула, которые позволяют, зная все значения функции $T(n)$ при $1 \leqslant n \leqslant m$ вычислить её значения при $n > m + 1$. Говорят, что функция задана *рекуррентным соотношением*, если её можно записать в виде формулы:

$$\begin{cases} T(1) = t_1, T(2) = t_2, \ldots, T(m) = t_m; \\ T(n+1) = f(t_n, t_{n-1}, \ldots, t_1), \text{ при } n \geqslant m. \end{cases} \tag{Ж.1}$$

Если для функции есть явно заданное рекуррентное соотношение, то функция завершается. Это доказывается с помощью метода математической индукции. Тогда для доказательства завершения реализации этой функции на языке Пифагор достаточно показать, что программа эквивалентна рекуррентному соотношению. Под эквивалентностью в данном случае понимается выполнение одинаковых математических операций над данными. Если, кроме завершения, доказана частичная корректность рекуррентного соотношения, то эквивалентная программа на языке Пифагор будет тотально корректна.

Таким образом, доказательство корректности и завершения функции отделяется от программного кода и доказывается отдельно, как доказательство тотальной корректности рекуррентного соотношения. А корректность функции на языке Пифагор следует из доказательства её эквивалентности заданному рекуррентному соотношению.

Доказывать эквивалентность можно несколькими способами. Один из способов — изменение тройки Хоара программы на эквивалентную, с изменённым предусловием и посту-

словием. Например, если рекуррентное соотношение имеет вид (Ж.1), то исходную тройку $\boxed{P(x)}$ `Prog` $\boxed{Q(r)}$ можно заменить на

$$\boxed{P(x) \wedge (x \in \mathbb{Z}) \wedge (x \geqslant 1)} \texttt{ Prog } \boxed{r = T(x)}\,.$$

В результате формулы, к которым сводятся ИГР, будут иметь более простой вид, что повышает вероятность автоматического доказательства их истинности.

Другой способ доказательства эквивалентности некоторой функции `fp` на языке Пифагор и рекуррентного соотношения f основан на использовании особенности структуры программы языка Пифагор. Выбор различных вариантов выполнения программы на языке Пифагор реализуется с помощью конструкции:

$$\texttt{(}a_1, \ldots, a_n\texttt{):[(}c_1, \ldots, c_n\texttt{):?]:.} \qquad \text{(Ж.2)}$$

где $(a_1, \ldots, a_n)$ — список вариантов выполнения программы, выбор варианта i определяется условием c_i. При этом, каждый элемент a_i, который не является константой, должен быть задержанным списком. Все условия c_i должны быть взаимоисключающими: для любого входного аргумента x (некоторого типа A), удовлетворяющего предусловию `fp`, одно из условий c_i всегда истинно, а остальные обязательно ложные. Это можно выразить в виде формулы:

$$\forall x : A.\, P(x) \Rightarrow \left((\bigvee_{i=1}^{n} c_i = \text{true}) \wedge (\forall i, j : \mathbb{N}.\, ((1 \leqslant i, j \leqslant n) \wedge (i \neq j)) \Rightarrow (c_i \wedge c_j = \text{false})) \right).$$

Представим рекуррентное соотношение (Ж.1) в виде:

$$f_r(x) = \begin{cases} \widetilde{a}_1(x), \text{ если } \widetilde{c}_1(x); \\ \ldots \\ \widetilde{a}_m(x), \text{ если } \widetilde{c}_m(x); \end{cases}$$

Здесь каждая из функций $\widetilde{c}_i$ не зависит от f_r; каждая из функций $\widetilde{a}_i(x)$ является либо константой, либо зависит от f_r для меньших значений аргумента.

Тогда для доказательства эквивалентности функции `fp` на языке Пифагор рекуррентному соотношению f_r требуется:

1. выделить в функции `fp` элементы, соответствующие (Ж.2), показать, что $n = m$;
2. разметить дуги c_i, $i = 1, \ldots, n$, проверить, что условия c_i взаимоисключающие и показать совпадение отношений (предикатов) c_i и $\widetilde{c}_i$;
3. применить операцию расщепления и разделить исходный граф на n подграфов, разметить дуги a_i, $i = 1, \ldots, n$, и показать равенство функций a_i и $\widetilde{a}_i$.

Замечание. Проверка условия взаимоисключения для c_i обусловлена особенностью языка Пифагор, который, являясь параллельным, допускает одновременное выполнение нескольких ветвей a_i. Рекуррентное же соотношение обычно считается «последовательным алгоритмом», то есть первое выполненное условие $\widetilde{c}_i$ делает ответом $\widetilde{a}_i$, а все последующие варианты игнорируются.

Для проверки совпадения отношений c_i с $\widetilde{c}_i$ и функций a_i с $\widetilde{a}_i$ можно воспользоваться формулой:

$$\bigwedge_{i=1}^{n} \left(\Big(\mathrm{Fold}(res_i) \wedge \widetilde{P}(x) \wedge \widetilde{c}_i(x) \Big) \Rightarrow (res_i = \widetilde{a}_i(x)) \right), \tag{Ж.3}$$

где res_i — идентификатор выходной дуги i-го графа, полученного при расщеплении; $\widetilde{P}$ — область определения (предусловие) рекуррентного соотношения; $\mathrm{Fold}(res_i)$ — формула, полученная при проведении полной свёртки для дуги res_i. Формула $\mathrm{Fold}(res_i)$ включает в себя в качестве конъюнктов предусловие $P(x)$ и формулы, выражающие c_i и res_i через входной аргумент x. Наличие условий $P(x)$ (в составе $\mathrm{Fold}(res_i)$) и $\widetilde{P}$ необходимо, чтобы аргумент x принадлежал области определения как функции так и рекуррентного соотношения. При разметке подграфов, содержащих рекурсивные вызовы, предполагается, что для них доказано совпадение с рекуррентным соотношением, и их выходные дуги размечаются формулами $x_{out} = f_r(x_{in})$, где x_{in} — идентификатор рекурсивного аргумента, а x_{out} — идентификатор выходной дуги рекурсивного вызова, f_r — имя рекуррентного соотношения.

Если удаётся показать истинность формулы (Ж.3), то все res_i можно разметить формулами $res_i = f_r(x)$. И функция будет корректна, если корректно рекуррентное соотношение.

Пример. В примере раздела 3.1.1 для функции `order` нет явного рекуррентного соотношения. Приведём другой алгоритм нахождения количества цифр в натуральном числе, для которого есть явно заданное рекуррентное соотношение:

$$f(n) = \begin{cases} 1, & \text{если } 1 \leqslant n \leqslant 9 \\ f(\lfloor n/10 \rfloor) + 1, & \text{если } n > 9, \end{cases} \tag{Ж.4}$$

где $\lfloor x \rfloor$ — функция нахождения целой части числа x.

Зададим предусловие и постусловие:

$$P(n) \equiv (n \in \mathrm{int}) \wedge (n > 0),$$

$$Q(r) \equiv (res \in \mathrm{int}) \wedge (10^{res-1} \leqslant n) \wedge (n < 10^{res}).$$

Завершение вычислений по данному рекуррентному соотношению очевидно. Покажем, что результат вычислений будет удовлетворять постусловию. Доказательство проведём индукцией по n.

База индукции. $n \in [1,9]$, $f(n) = 1$, $10^0 \leqslant 1 < 10^1$.

Шаг индукции. Предполагаем, что для $k_0 = f(\lfloor n/10 \rfloor)$ верно $10^{k_0-1} \leqslant \lfloor n/10 \rfloor < 10^{k_0}$, требуется показать, что для n будет выполнено $10^{k_0} \leqslant n < 10^{k_0+1}$. Пусть $n = \lfloor n/10 \rfloor \cdot 10 + q$, $0 \leqslant q \leqslant 9$. Домножим обе части неравенства $10^{k_0-1} \leqslant \lfloor n/10 \rfloor$ на 10 и прибавим q:

$$10^{k_0} \leqslant 10^{k_0} + q \leqslant \lfloor n/10 \rfloor \cdot 10 + q = n.$$

Остаётся показать, что $n < 10^{k_0+1}$. Имеем $\lfloor n/10 \rfloor < 10^{k_0}$. Так как $\lfloor n/10 \rfloor$ — целое число, то будет верно неравенство: $\lfloor n/10 \rfloor \leqslant 10^{k_0} - 1$. Домножим обе части последнего неравенства на 10 и сложим с неравенством $q < 10$, получаем:

$$n = \lfloor n/10 \rfloor \cdot 10 + q < (10^{k_0} - 1) \cdot 10 + 10 = 10^{k_0+1},$$

что и требовалось доказать.

Программа на языке Пифагор, реализующая рекуррентное соотношение (Ж.4):

```
orderR << funcdef n{
    np<<(n,9):<=;
    if<<(np,np:-):?;
    act<<( 1, { ((n,10):%:1:orderR,1):+ } );
    return << act:if:.;
}
```

Исходный информационный граф программы приведён на рисунке Ж.1а. Пред- и постусловия программы такие же, как у рекуррентного соотношения.

Докажем эквивалентность программы `orderR` рекуррентному соотношению (Ж.4).

1. Выделим в функции `orderR` элементы соответствующие (Ж.2) :

$$(1, d_1)\texttt{:[}(np, nnp)\texttt{:?]:.}$$

Оба списка данных с условиями и вариантами выполнения программы имеют одинаковую длину, равную двум. В рекуррентном соотношении (Ж.4) также два варианта формул, по которым находится значение f. Первый элемент списка $(1, d_1)$ является константой, а второй элемент d_1 является задержанным списком.

2. Разметим дуги np и nnp, на основе аксиом для встроенных функций «`<=`» и «`-`» (для разметки применимы аксиомы 14.1 и 6.1 соответственно, приложение Д). Формулы разметки приведены на рисунке Ж.1а. Условия np и nnp взаимоисключающие.

3. Расщепление исходного информационного графа даст два ИГР G_1 и G_2:

$$G_1 \equiv \boxed{P(n) \wedge (n \leqslant 9)}\ \texttt{1:.} \to res\ \boxed{Q(res)}\ ,$$

$$G_2 \equiv \boxed{P(n) \wedge \neg(n \leqslant 9)}\ \texttt{\{ ((n,10):\%:1:orderR,1):+ \}:.} \to res\ \boxed{Q(res)}\ .$$

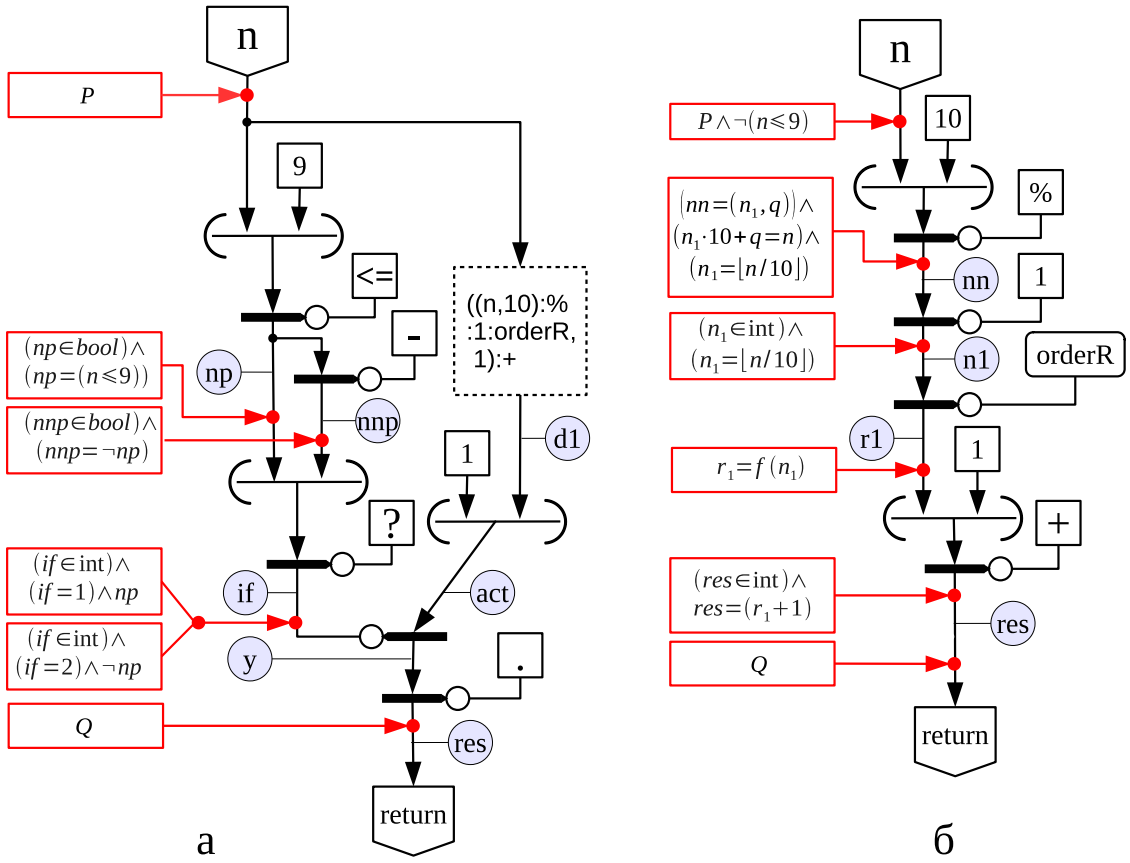


Рисунок Ж.1 — Информационный граф с разметкой для функции orderR; а — исходный информационный граф функции с частичной разметкой, задержанный список представлен как константа; б — полностью размеченный информационный граф, содержащий рекурсивную ветвь

Граф G_1 может быть полностью размечен, при этом к дуге res будет приписана формула $res = 1$, при свёртке получаем тройку с «пустой» программой:

$$\boxed{P(n) \wedge (n \leqslant 9) \wedge (res = 1)} \quad \boxed{Q(res)} \ ,$$

Проверим истинность формулы (Ж.3) для первого варианта выполнения программы:

$$(P(n) \wedge (n \leqslant 9) \wedge (res = 1) \wedge (1 \leqslant n \leqslant 9)) \Rightarrow (1 = 1) \, .$$

Данная формула является тождественно истинной.

В G_2 присутствует задержанный список, операция эквивалентного преобразования при раскрытии задержанного списка даст граф, приведённый на рисунке Ж.1б.

Функция «%», применённая к списку с двумя целыми числами (x, y), возвращает список из двух элементов, на первой позиции будет результат целочисленного деления $\lfloor x/y \rfloor$, а на второй — остаток от деления.

На основе аксиом для функций «%» разметим дугу nn (применима только одна аксиома 9.1, приложение Д). Далее к nn применяется функция выбора элемента из списка,

которая выбирает первый элемент, обозначенный n_1. После этого к разметке готов оператор r_1, который осуществляет рекурсивный вызов функции `orderR`. Предполагаем, что рекурсивный вызов функции соответствует рекуррентному соотношению (Ж.4), и разметим дугу r_1 формулой: $r_1 = f(n_1)$.

После разметки дуги r_1, остаётся одна не размеченная дуга res, которая размечается на основе аксиом для встроенной функции «+» (применима только одна аксиома 5.2, приложение Д). На рисунке Ж.1б приведена формула, размечающая дугу res.

После разметки всех дуг графа можно проверить истинность формулы (Ж.3) для второго варианты выполнения программы:

$$(P(n) \wedge \neg(n \leqslant 9) \wedge (n_1 = \lfloor n/10 \rfloor) \wedge (r_1 = f(n_1)) \wedge (res = r_1 + 1)) \Rightarrow (res = f(\lfloor n/10 \rfloor) + 1)\,.$$

После упрощения:

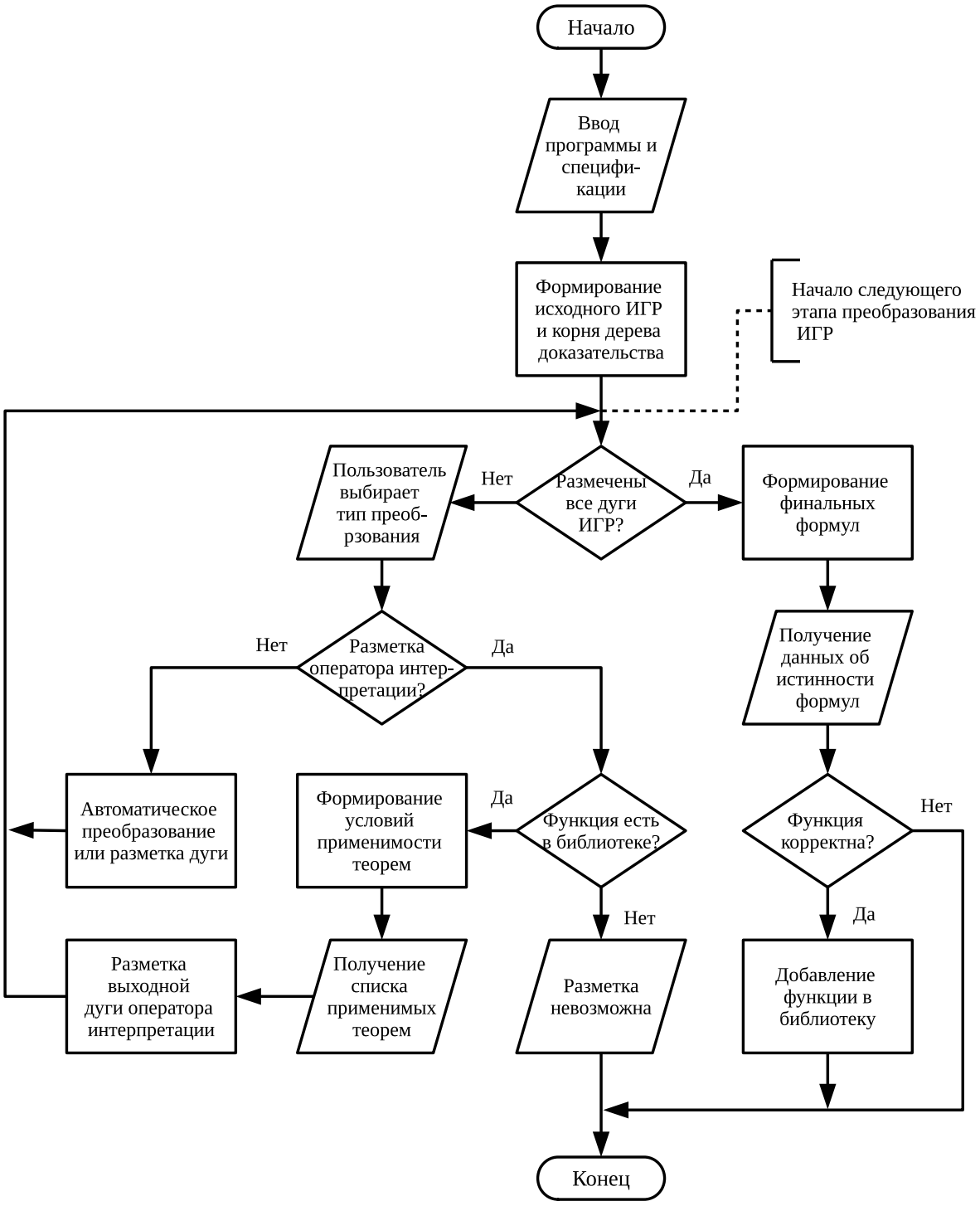
$$(P(n) \wedge (n > 9) \wedge (res = f(\lfloor n/10 \rfloor) + 1) \Rightarrow (res = f(\lfloor n/10 \rfloor) + 1)\,.$$

Данная формула является тождественно истинной.

Таким образом, эквивалентность программы `orderR` рекуррентному соотношению (Ж.4) доказана.

Приложение З

Алгоритм работы системы поддержки доказательства



Алгоритм работы инструментального средства поддержки доказательства корректности ФПП программ приведён в упрощённом виде для отражения общего принципа работы системы.

Приложение И

Пример текстового представления РИГ

Код функции `Func` на языке Пифагор, вычисляющей значение выражения $a \cdot b + c$, где a, b и c — целые числа:

```
Func << funcdef x {
    a << x:1;
    b << x:2;
    c << x:3;
   ((a, b):*,c):+  >>  return
}
```

Текстовое представление РИГ функции `Func` имеет вид:

```
External
    0    Func
Local
    0    1
    1    2
    2    3
id   delay    operation    links        positions
0    0        arg                       pos 1 16 1 17
1    0        :            0 loc:0      pos 2 16 2 17
2    0        :            0 loc:1      pos 3 16 3 17
3    0        :            0 loc:2      pos 4 16 4 17
4    0        (---)        1 2          pos 5 10 5 16
5    0        :            4 *          pos 5 16 5 17
6    0        (---)        5 3          pos 5 9 5 21
7    0        :            6 +          pos 5 21 5 22
8    0        return       7            pos 5 29 5 35
```

Файл с идентификаторами дуг РИГ функции `Func`:

```
Extern    Name
0        Func
Local    Name
0
1
2
Actor    Name
0        x
1        a
2        b
3        c
4
5
6
7
8
```

Приложение К

Синтаксис текстового представления языка спецификации

Для описания синтаксиса языка спецификации используются формы Бэкуса-Наура (БНФ). Нетерминальные символы записываются в угловых скобках «<» и «>», терминальные без скобок. Для описания правил используются метасимволы, которые имеют следующий смысл:

1. «::=» отделяет левую часть правила от правой;
2. каждое правило заканчивается точкой «.»;
3. «|» разделяет несколько альтернатив;
4. парные круглые скобки «(» и «)» используются для ограничения альтернативных конструкций, и означают, что из всех перечисленных внутри скобок альтернатив в данном месте правила может стоять только одна альтернатива;
5. парные квадратные скобки «[» и «]» означают, что заключённые в них символы могут отсутствовать;
6. парные фигурные скобки «{» и «}» означают, что указанные в них символы могут встречаться ноль и более раз;
7. кавычки « и » используются в тех случаях, когда терминальный символ совпадает с одним из метасимволов; если терминальный символ состоит из последовательности символов, один из которых совпадает с метасимволом, то вся последовательность заключается в кавычки.

Для сокращения записи некоторые нетерминальные символы описаны с помощью текстовых комментариев.

```
<терм> ::= <переменная> | <константа> | <применение функции> | <функция> |
           <тип функции> | <сокращения> | «(»<терм>«)» .
<переменная> ::= <идентификатор> .
<идентификатор> ::= <буква>{<буква>|<цифра>} .
```

Идентификаторы переменных не должны совпадать с терминальными символами констант как встроенных, так и пользовательских.

<буква> — строчные и прописные буквы английского алфавита и символ подчёркивания.

```
<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 .
<константа> ::= <встроенный тип> | <элементы встроенных типов> |
                <встроенные операции> | <пользовательские константы> .
```

<встроенный тип> ::= ind | bool | N | R | char | error | func | signal |
delaylist | list | datalist | parlist | Set | Type .

<элементы встроенных типов> ::= 0N | 0 | 1 | «0.0» | «1.0» | ’<символ>’ | . |
<константа ошибки> | <имена функций Пифагор> |
nil | hnil | phnil | MinInt | MaxInt |
MinFloat | MaxFloat .

<символ> ::= <буква> | <цифра> | <специальный знак> .

<специальные знаки> — произвольные символы, не совпадающие с буквами, цифрами и служебными символами (разные виды скобок, точка, запятая, двоеточие, кавычки, вертикальная черта); например, !, $, %, ^ и др.

<константа ошибки> ::= BASEFUNCERROR | BOUNDERROR | INTERROR | REALERROR |
ZERODIVIDE | TYPEERROR .

<имена функций Пифагор> ::= type_p | len_p | plus_p | minus_p | dot_p | div_p |
mod_p | eq_p | neq_p | ls_p | gr_p | leq_p | geq_p |
quest_p | hash_p | datalist_p | parlist_p | dup_p |
dd_p | int_p | float_p | char_p | bool_p | signal_p .

<встроенные операции> ::= <префиксная операция> | <инфиксная операция> .

<префиксная операция> ::= ~ | OneOne | Onto | SUC | minus | NR | ZR | RZ |
cons | head | tail | length | elem |
hcons | hhead | htail | hlength | helem |
phcons | phhead | phtail | phlength | phelem .

<инфиксная операция> ::= /\ | \/ | «=>» | «<=>» | + | * | - | mod | / |
= | != | «<» | «>» | «<=» | «>=» | in | notin .

<пользовательские константы> ::= <идентификатор> | <последовательность знаков> .

<последовательность знаков> ::= <специальный знак>{<специальный знак>} .

Пользователь может добавлять свои идентификаторы констант, при этом они не должны совпадать с уже существующими. Идентификаторы констант могут состоять из букв и цифр или из последовательности специальных знаков.

<применение функции> ::= <терм>«(» <терм> {, <терм>}«)» |
<терм> <инфиксная операция> <терм> .

<функция> ::= (fun | choice | forall | exists[!])<переменная с типом>«.»<терм> .

<переменная с типом> ::= <переменная>:<терм> .

<тип функции> ::= <зависимый тип> | <независимый тип> .

<зависимый тип> ::= dtp <переменная с типом>«.»<терм> .

<независимый тип> ::= <терм> «->» <терм> .

```
<сокращения> ::= <множество> | <строка> | <список> | <число> |
                 <выбор элемента из списка> .
<принадлежность множеству> ::= <терм> ( in | notin ) <множество> .
<множество> ::= <перечисление> | <характеристическая функция> .
<перечисление> ::= «(» <терм> «|» <терм> {«|» <терм>} «)» .
<характеристическая функция> ::= «{» <переменная с типом> «|»
                                     <применение функции> «}») .
<строка> ::= ’’ <символ>{<символ>} ’’ [delaylist] .
<список> ::= «(_» <терм> {,<терм>} «_)» |
             «(_» <терм с типом> {,<терм с типом>} «_)» |
             «[» <терм с типом> {,<терм с типом>} «]» .
<терм с типом> ::= <терм>:<терм> .
<число> ::= <натуральное число> | <целое число> | <действительное число> .
<натуральное число> ::= <цифра>{<цифра>}N .
<целое число> ::= [-]<цифра>{<цифра>} .
<действительное число> ::= [-]<цифра>{<цифра>}«.»<цифра>{<цифра>} .
<выбор элемента из списка> ::= <терм> «[»<терм>«]» .
```

Приложение Л

Входные данные глобального окружения

Файл «globalConstants.txt» содержит стандартную сигнатуру теорий языка спецификации:

```
Type : Type2
-> : Type->Type
Set : Type
ind : Set
bool : Set
N : Set
Z : Set
R : Set
char : Set
error : Set
func : Set
signal : Set
delaylist : Set
list : Type->Type
datalist : list(Set)->Set
parlist : list(Set)->Set
true : bool
false : bool
ON : N
0 : Z
1 : Z
0.0 : R
1.0 : R
BASEFUNCERROR : error
BOUNDERROR : error
INTERROR : error
REALERROR : error
ZERODIVIDE : error
TYPEERROR : error
type_p : func
len_p : func
plus_p : func
minus_p : func
dot_p : func
div_p : func
mod_p : func
eq_p : func
neq_p : func
ls_p : func
gr_p : func
leq_p : func
geq_p : func
quest_p : func
hash_p : func
datalist_p : func
parlist_p : func
dup_p : func
```

```
dd_p : func
int_p : func
float_p : func
char_p : func
bool_p : func
signal_p : func
• : signal
nil : list
hnil : datalist
phnil : parlist
=> : bool->bool->bool
= : dtp A:Type.A->A->bool
!= : dtp A:Type.A->A->bool
choice : dtp A:Type.(A->bool)->A
forall : dtp A:Set.(A->bool)->A
~ : bool -> bool
/\ : bool->bool->bool
\/ : bool->bool->bool
<=> : bool->bool->bool
OneOne : dtp A:Set. dtp B:Set. (A->B)->bool
Onto : dtp A:Set. dtp B:Set. (A->B)->bool
SUC : N->N
+ : Set->Set->Set
* : Set->Set->Set
- : Set->Set->Set
/ : Set->Set->Set
minus : Set->Set
< : Set->Set->bool
> : Set->Set->bool
<= : Set->Set->bool
>= : Set->Set->bool
mod : Z->Z->Z
NR : N->R
ZR : Z->R
RZ : R->Z
CharInt : char->N
IntChar : N->char
cons : dtp A:Type.A->list(A)
head : dtp A:Type.list(A)->A
tail : dtp A:Type.list(A)->list(A)
length : dtp A:Type.list(A) -> N
elem : dtp A:Type.N->list(A) ->A
hcons : dtp A:Set. dtp L:list(Set).A->datalist(L)->datalist(cons(A,L))
hhead : dtp A:Set. dtp L:list(Set).datalist(cons(A,L))->A
htail : dtp A:Set. dtp L:list(Set).datalist(cons(A,L))->datalist(L)
hlength : dtp L:list(Set).datalist(L)->N
helem : dtp A:Set. dtp L:list(Set).N->datalist(L)->A
phcons : dtp A:Set. dtp L:list(Set).A->parlist(L)->parlist(cons(A,L))
phhead : dtp A:Set. dtp L:list(Set).parlist(cons(A,L))->A
phtail : dtp A:Set. dtp L:list(Set).parlist(cons(A,L))->parlist(L)
phlength : dtp L:list(Set).parlist(L)->N
phelem : dtp A:Set. dtp L:list(Set).N->parlist(L)->A
```

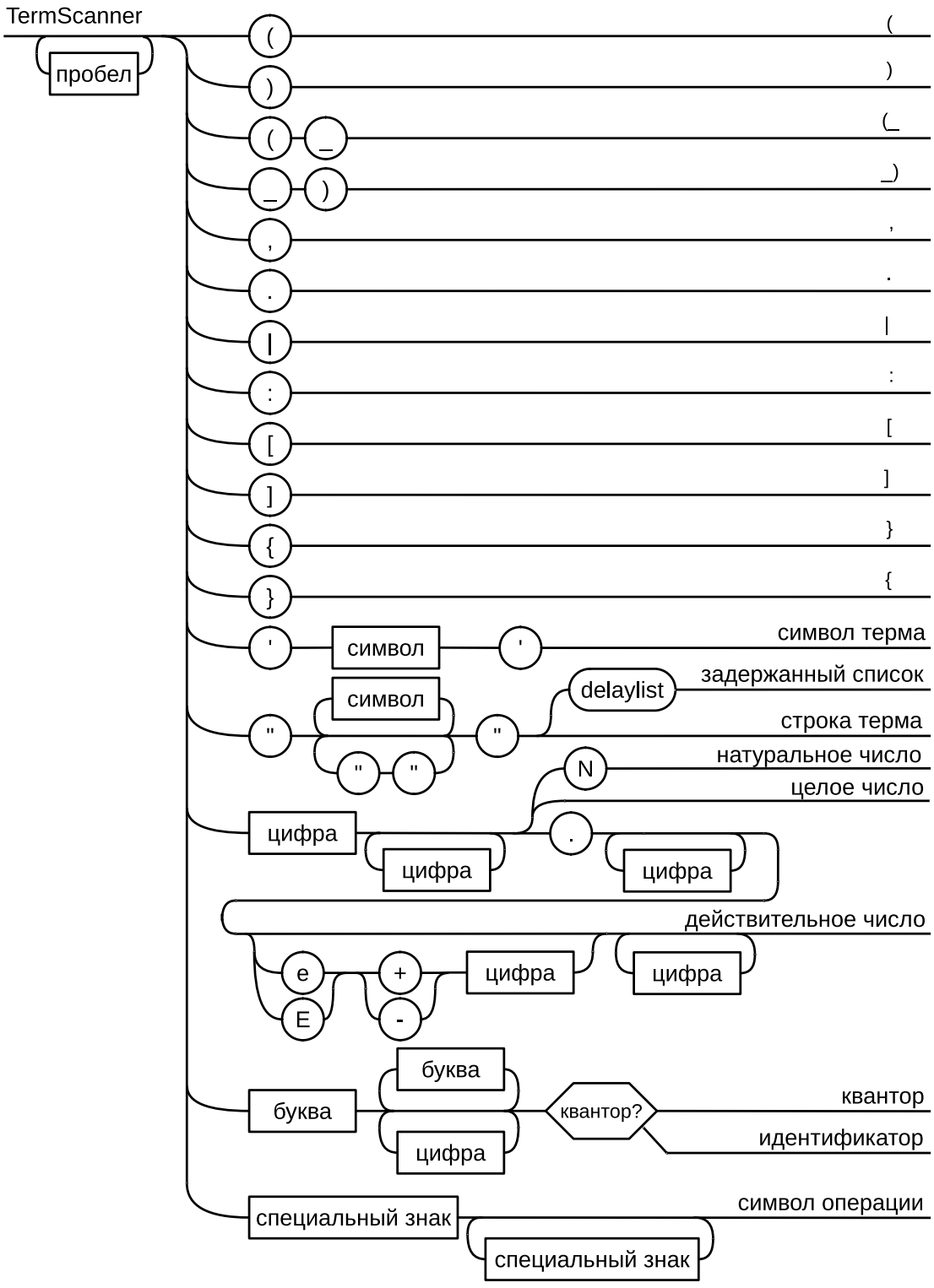
```
in : dtp A:Set. dtp B:Set. B->bool
notin : dtp A:Set. dtp B:Set. B->bool
MaxInt : Z
MinInt : Z
MaxFloat : R
MinFloat : R
int : Set
float : Set
```

Файл «infixOperations.txt» содержит список стандартных инфиксных операций теорий языка спецификации. На первой позиции стоит имя, на второй — приоритет:

```
+     4
-     4
*     3
/     3
mod   3
=     14
!=    14
/\    11
\/    12
=>    14
<=>   14
in    2
notin 2
->    15
>     6
<     6
>=    6
<=    6
```

Приложение М

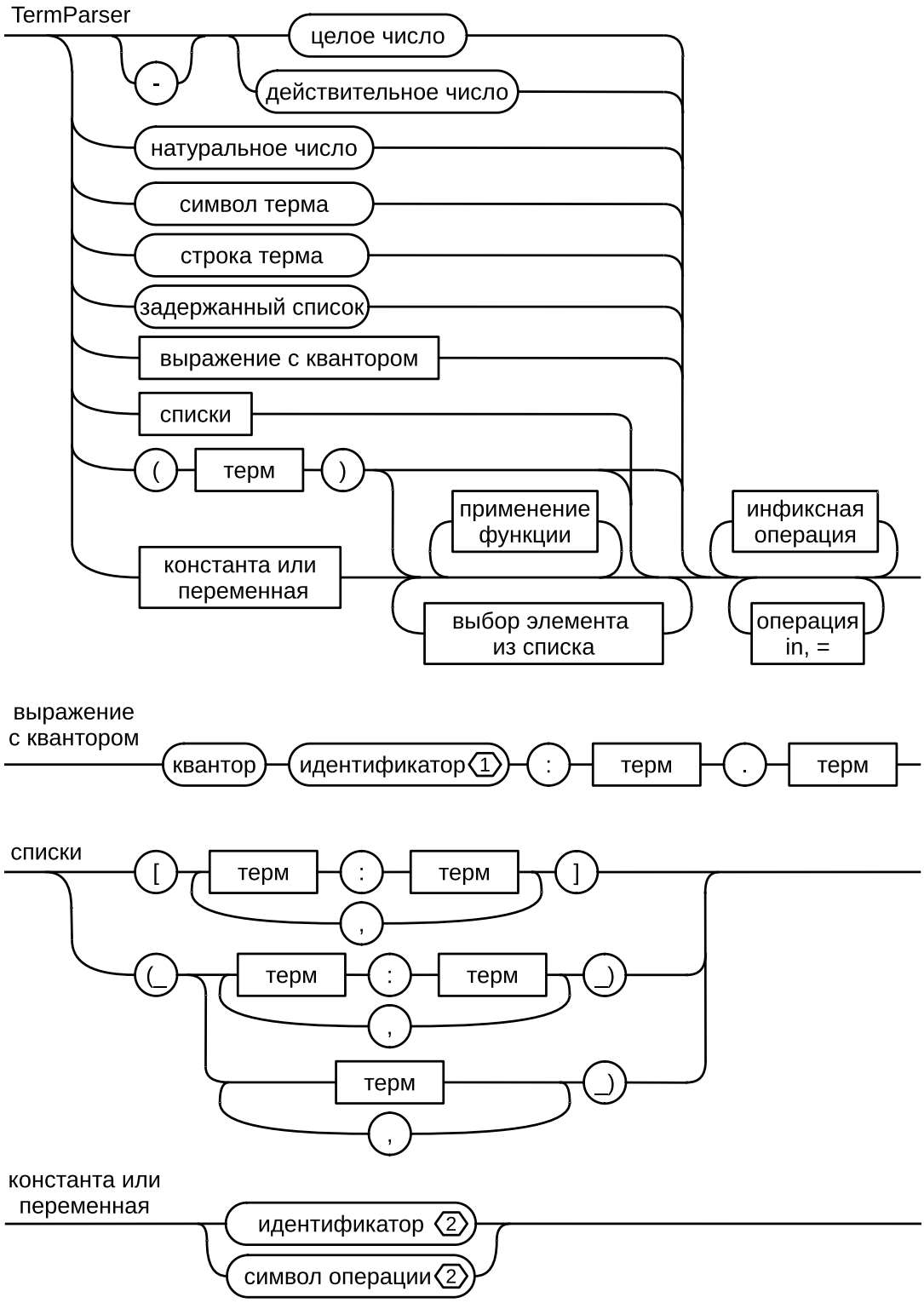
Диаграмма лексического анализа текстового представления термов

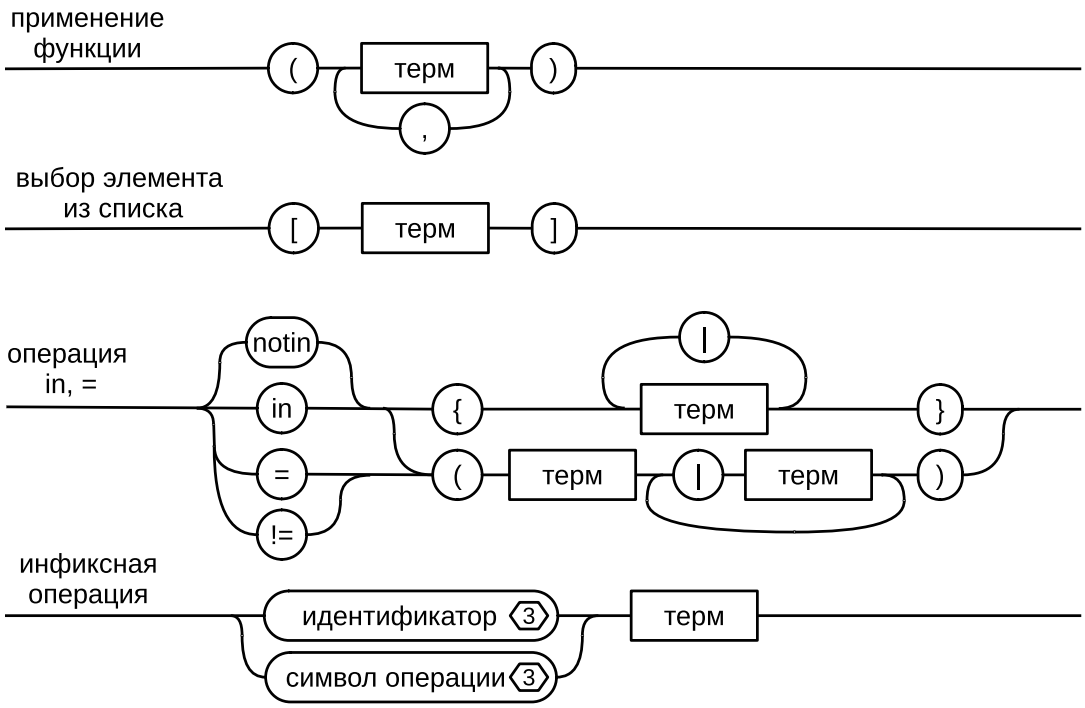


В овалах приведены терминальные символы, поступающие на вход сканеру, в прямоугольниках — нетерминальные символы, шестиугольник обозначает использование семантического анализа для отделения ключевых слов, обозначающих кванторы, от идентификаторов. Над выходными дугами написаны имена генерируемых лексем.

Приложение Н

Диаграмма синтаксического анализа текстового представления термов





В овалах приведены терминальные символы, поступающие на вход парсеру. Ими являются лексемы, полученные от сканера. В прямоугольниках находятся нетерминальные символы. Шестиугольники с числом в некоторых овалах означают, что для поступающей лексемы проводится дополнительный семантический анализ: 1 — идентификатор должен быть идентификатором переменной, ещё не присутствующей в таблицах переменных; 2 — идентификатор или символ операции не является инфиксной операцией; 3 — идентификатор или символ операции является инфиксной операцией.

Приложение О

Пример текстового представления ИГР

На Рисунке О.1 приведён информационный граф с разметкой для функции `Func`, вычисляющей значение выражения $a \cdot b + c$, где a, b и c — целые числа.

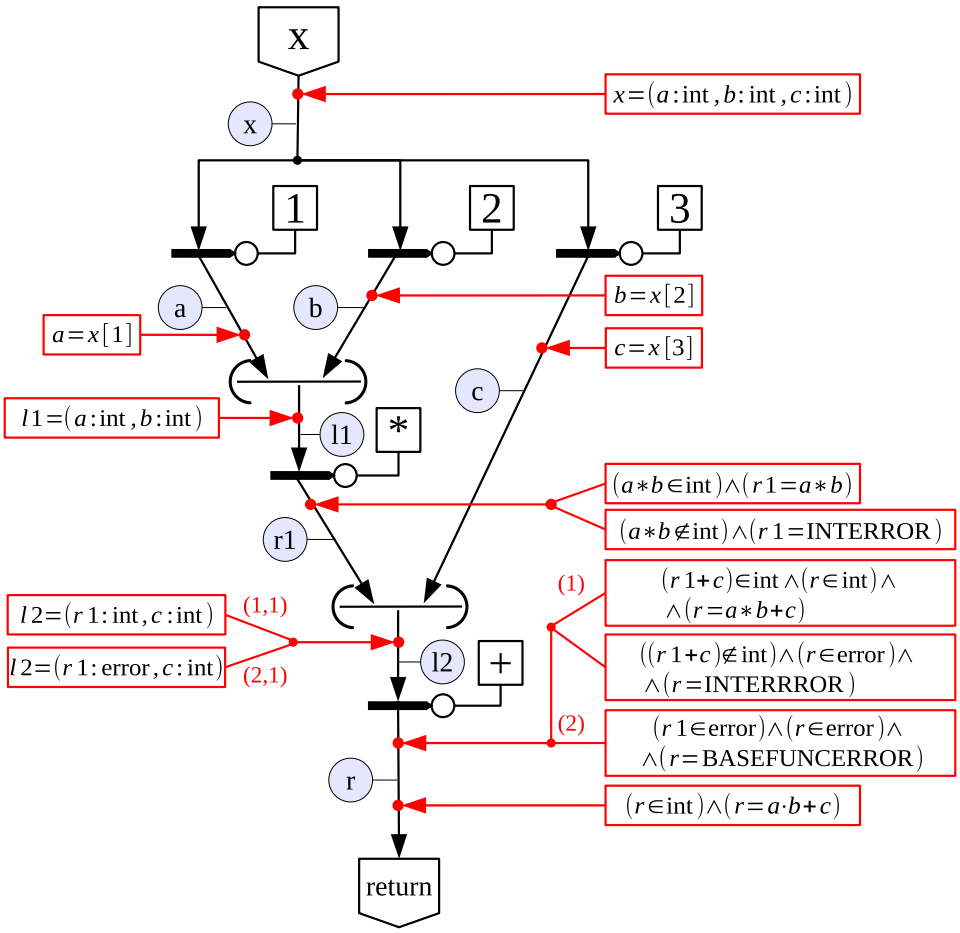


Рисунок О.1 — Информационный граф с разметкой функции `Func`; в скобках над некоторыми дугами указаны индексы формул

Ниже приведено текстовое представление данного ИГР в псевдо-xml формате. Текстовый формат ИГР содержит РИГ `<RIG>`, таблицу имён `<nameTable>`, предусловие `<precondition>` и постусловие `<postcondition>`. Если тройка не имеет разметки, то после тега `</postcondition>` идёт тег `</IGR>`. В рассматриваемом примере после постусловия идут формулы разметки `<marking>`. Каждый узел `<node>` имеет номер `<nodeNum>`, соответствующий номеру оператора в РИГ и список из количества формул у каждой его входной дуги `<parentXformulasNum>`. Далее идёт позиция индекса `<indexPos>` и список формул `<formulasList>`. Формулы могут быть многострочными, поэтому каждая заключена в теги `<formula>`, `</formula>`.

```
<IGR>
<RIG>
External
0 Func
Local
0 1
1 2
2 3
id delay operation links positions
0 0 arg pos 1 16 1 17
1 0 : 0 loc:0 pos 2 16 2 17
2 0 : 0 loc:1 pos 3 16 3 17
3 0 : 0 loc:2 pos 4 16 4 17
4 0 (---) 1 2 pos 5 10 5 16
5 0 : 4 * pos 5 16 5 17
6 0 (---) 5 3 pos 5 9 5 21
7 0 : 6 + pos 5 21 5 22
8 0 return 7 pos 5 29 5 35
</RIG>
<nameTable>
Extern Name
0 Func
Local Name
0 c0
1 c1
2 c2
Actor Name
0 x
1 a
2 b
3 c
4 l1
5 r1
6 l2
7 r
8 ret
</nameTable>
<precondition>
x=(_ a:int, b:int, c:int _)
</precondition>
<postcondition>
(r in int) /\ (r= a * b +c)
</postcondition>
<marking>
<node>
<nodeNum>
1
</nodeNum>
<parentXformulasNum>
1
</parentXformulasNum>
<indexPos>
```

```
0
</indexPos>
<formulasList>
<formula>
a=x[c0]
</formula>
</formulasList>
</node>
<node>
<nodeNum>
2
</nodeNum>
<parentXformulasNum>
1
</parentXformulasNum>
<indexPos>
0
</indexPos>
<formulasList>
<formula>
b=x[c1]
</formula>
</formulasList>
</node>
<node>
<nodeNum>
3
</nodeNum>
<parentXformulasNum>
1
</parentXformulasNum>
<indexPos>
0
</indexPos>
<formulasList>
<formula>
c=x[c2]
</formula>
</formulasList>
</node>
<node>
<nodeNum>
4
</nodeNum>
<parentXformulasNum>
1
1
</parentXformulasNum>
<indexPos>
0
</indexPos>
<formulasList>
```

```
<formula>
l1 = (_ a:int, b:int _)
</formula>
</formulasList>
</node>
<node>
<nodeNum>
5
</nodeNum>
<parentXformulasNum>
1
</parentXformulasNum>
<indexPos>
0
</indexPos>
<formulasList>
<formula>
((a * b) in int) /\ (r1 in int) /\ (r1 = a * b)
</formula>
<formula>
((a * b) notin int) /\ (r1 in error) /\ (r1 = INTERROR)
</formula>
</formulasList>
</node>
<node>
<nodeNum>
6
</nodeNum>
<parentXformulasNum>
2
1
</parentXformulasNum>
<indexPos>
0
</indexPos>
<formulasList>
<formula>
l2 = (_ r1:int, c:int _)
</formula>
</formulasList>
<indexPos>
1
</indexPos>
<formulasList>
<formula>
l2 = (_ r1:error, c:int _)
</formula>
</formulasList>
</node>
<node>
<nodeNum>
7
```

```
</nodeNum>
<parentXformulasNum>
2
</parentXformulasNum>
<indexPos>
0
</indexPos>
<formulasList>
<formula>
((r1 + c) in int) /\( r in int) /\ (r = r1 + c)
</formula>
<formula>
((r1 + c) notin int) /\ (r in error) /\ (r = INTERROR)
</formula>
</formulasList>
<indexPos>
1
</indexPos>
<formulasList>
<formula>
(r1 in error)  /\ (r in error) /\ (r = BASEFUNCERROR)
</formula>
</formulasList>
</node>
</marking>
</IGR>
```