ФГБОУ ВО «Московский государственный университет имени М.В. Ломоносова»
Механико-математический факультет

На правах рукописи

Кривчиков Максим Александрович

Формальные модели и верификация свойств программ с использованием промежуточного представления

Специальность 05.13.17 — «Теоретические основы информатики»

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
доктор физико-математических наук,
профессор В.А. Васенин

Москва — 2015

Содержание

Введение 5

1 К постановке основной задачи исследования 20
1.1 Библиографический обзор 1930–1989 гг. 21
1.1.1 Формальные модели вычислений 21
1.1.2 Аксиоматическая, операционная и денотационная семантика . . . 24
1.1.3 Темпоральная логика и полиморфное λ-исчисление 25
1.1.4 Исчисление конструкций 27
1.1.5 Выводы по результатам библиографического обзора 29
1.2 Современное состояние исследований 30
1.2.1 Индуктивные и коиндуктивные типы (1990-е гг.) 31
1.2.2 Программирование с зависимыми типами (с 2000 г. по настоящее время) 32
1.2.3 Российские исследования 34
1.2.4 Выводы по современному состоянию исследований 36
1.3 Задачи диссертационного исследования 37
1.3.1 Исследование и разработка математических моделей программ . 38
1.3.2 Разработка макета языка и программного средства построения формальных моделей и верификации свойств программ 40
1.3.3 Исследование возможности использования макета программного средства для описания формальной семантики языков программирования 41
1.3.4 Исследование динамического параллельного исполнения программ 42

2 Математическая модель базового языка 43
2.1 Расширенное Исчисление Конструкций 43
2.1.1 Основные понятия 44
2.1.2 Типы суждений 47
2.1.3 Контекст 48
2.1.4 Тип 48
2.1.5 Зависимые произведения 49
2.1.6 Зависимые суммы 51
2.1.7 Свойства исчисления 52
2.1.8 Правила редукции 53
2.2 Исчисление с типами эквивалентности 54
2.2.1 Правила идентичности 56
2.2.2 Редукция — базовые правила 59
2.2.3 Эквивалентность — Type 60
2.2.4 Эквивалентность на зависимых произведениях 63

2.2.5 Эквивалентность на зависимых суммах 64
2.2.6 Эквивалентность — $a =_A b$ 65
2.2.7 Редукция в термах удаления 68
2.3 Выводы по второй главе 71

3 Разработка и реализация базового языка 72
3.1 Исчисление с индуктивными типами 72
3.1.1 Конечные типы 73
3.1.2 Натуральные числа 74
3.1.3 Высшие индуктивные типы 75
3.1.4 Индуктивные типы 78
3.1.5 Коиндуктивные типы 81
3.2 Реализация макета программного средства 83
3.2.1 Синтаксис 84
3.2.2 Основные конструкции 86
3.2.3 Преобразование в термы 88
3.2.4 Типизация 91
3.2.5 Частичный вывод типов 92
3.2.6 Особенности реализации 95
3.3 Выводы по третьей главе 97

4 Статическая семантика языков программирования 99
4.1 Стандарт ECMA-335 100
4.2 Разработка статической формальной семантики 102
4.2.1 Элементарные типы 104
4.2.2 Сборки, модули и типы данных 104
4.2.3 Пользовательские типы данных 106
4.2.4 Поля и методы 107
4.2.5 Окружения статической семантики 108
4.2.6 Тело метода 109
4.2.7 Семантика типизации 111
4.3 Динамическая семантика подмножества CIL 112
4.3.1 Монады и преобразователи монад 113
4.3.2 Модель чисел с плавающей точкой 116
4.3.3 Стек преобразователей монад динамической семантики 117
4.3.4 Динамическая семантика инструкций 119
4.3.5 Формальная семантика фрагмента кода 122
4.4 Выводы по четвёртой главе 124

5 Модель динамического параллельного исполнения программ 125
5.1 Язык управления потоком исполнения 128
5.2 Монада возобновлений и реактивный параллелизм 131
5.3 Модель динамического параллельного исполнения 134
5.3.1 Динамическая семантика языка управления потоком исполнения 134
5.3.2 Модель управляющего ядра системы 137
5.4 Свойства модели 140
5.4.1 Последовательный режим исполнения 140
5.4.2 Режим исполнения по требованию 144

5.5 Примеры . . . 145
5.5.1 Завершимость в различных режимах исполнения . . . 146
5.5.2 Параллельное исполнение фрагмента кода CMS . . . 149
5.6 Выводы по пятой главе . . . 150

Заключение . . . 152

Список рисунков . . . 154

Список таблиц . . . 154

Список листингов . . . 155

Литература . . . 155
Работы автора по теме диссертации . . . 173

Приложение A Данные по состоянию программной инженерии . . . 175
A.1 Увеличение размера исходного кода программного обеспечения . . . 175
A.2 Плотность дефектов в программных продуктах с открытым исходным кодом . . . 176
A.3 Количество языков программирования, используемых при разработке программных продуктов с открытым исходным кодом . . . 177

Приложение B Исходный код для системы Coq . . . 180

Приложение C Формальная модель вычислений с плавающей точкой . . . 183
C.1 Введение . . . 183
C.2 Стандарт IEEE-754 . . . 184
C.3 Разработка модели чисел с плавающей точкой . . . 185
C.4 Тестирование модели . . . 187
C.5 Адаптация модели для других реализаций лямбда-исчисления с зависимыми типами . . . 189
C.6 Существующие результаты . . . 190
C.7 Заключение . . . 190

Приложение D Статическая формальная семантика стандарта ECMA-335 193

Приложение E Фрагмент кода CMS для демонстрации . . . 197

Приложение F Реализация модели динамического параллельного исполнения программ . . . 201
F.1 Фрагменты кода модели . . . 201
F.2 Исходный код на языке Haskell . . . 203

Введение

Актуальность работы. Программное обеспечение в настоящее время используется практически во всех сферах деятельности человека, в том числе — в организации быта и при автоматизации производственных процессов, в системах телекоммуникаций, на объектах транспорта и в медицинском оборудовании, в научных исследованиях и при сопровождении объектов критически важных инфраструктур [155, 156], включая оборонный комплекс. Согласно исследованию [18], проведённому на базе Кембриджского университета, ежегодные потери мировой экономики от просчётов, дефектов и ошибок в программах на 2013 год оцениваются в 312 млрд долларов США, а в [109] приводятся данные в 59.5 млрд долларов на 2002 год только для экономики США. Известны также случаи, когда ошибки в программном обеспечении становились причиной многочисленных травм и даже гибели людей [17, 79, 130]. Такие просчёты, различного рода дефекты и ошибки в исходном коде могут появиться на разных стадиях жизненного цикла программного продукта — от проектирования до модификации на этапе полномасштабной эксплуатации. Это особенно характерно для современных, сложно организованных (логически и архитектурно), больших по объёму программных комплексов [130, 174]. Чем раньше подобные недостатки появляются, тем труднее их обнаружить и найти способы исправления, как следствие — большие издержки. В этой связи, особенно актуальны в настоящее время поиски методов и средств, позволяющих обнаружить и устранить дефекты на ранней стадии жизненного цикла программы, а также исследования, предоставляющие возможность доказать корректность программного обеспечения по отношению к его спецификации, в первую очередь — функциональной. Отметим, что такие просчёты, дефекты и ошибки при написании кода, которые упоминаются выше, могут иметь как технический характер (например, нарушение типизации при работе с памятью или переполнение буфера), так и характер нарушения требований к программному продукту.

В рамках управления большими программными проектами обнаружение ошибок и дефектов, а также доказательство корректности продукта являются важнейшим элементом. При этом верификация и валидация продукта кода программ в основном проводятся на стадии их испытаний. Однако для повышения эффективности такие процессы должны распространяться и на другие стадии жизненного цикла продукта, от предпроектных исследований до его утилизации. Отметим, что процессы верификации и валидации определяются следующим образом [72]:

верификация — процесс, целью которого является показать соответствие продукта, сервиса или системы требованиям, спецификациям и другим условиям, которые накладываются на продукт;

валидация — процесс, целью которого является определить адекватность продукта, сервиса или системы потребностям заказчика.

С позиций минимизации числа дефектов и ошибок в программном обеспечении, особенно на начальных этапах его жизненного цикла, основным является процесс верификации, который и будет рассматриваться далее. В инженерии программ могут быть выделены следующие четыре основных подхода к верификации [160, Лекция 13].

Визуальный контроль (рецензирование) — верификация проводится путём просмотра и анализа исходного текста программы на соответствие предъявляемым к ней требованиям. В международной практике для обозначения этого подхода используется термин code review. Его применение рекомендуется во многих моделях разработки программного обеспечения. В том числе, такой подход используется в популярных на настоящее время моделях жизненного цикла Agile, в которых визуальный контроль встроен в модель как в качестве отдельного технологического процесса, так и в формате парного программирования. Тем не менее, визуальный контроль программ полагается на экспертное мнение и не может гарантировать должный уровень корректности, что, в особенности, является отрицательным фактором при разработке сложных программных комплексов.

Тестирование — верификация проводится путём выполнения программы на заранее заданных наборах входных данных и последующего сравнения полу-

ченных результатов с эталонными, полученными на основе требований или с использованием модели или прототипа программного комплекса. На практике за последние десятилетия тестирование получает все большее распространение и на примере проектов с открытым исходным кодом используется почти повсеместно. Отметим, что в [160] в понятие «тестирование программной системы» входит более широкий класс подходов, в том числе и рассмотренный далее статический анализ. Хотя в рамках тестирования возможно применение формальных методов (в том числе — генерация тестов по спецификациям и моделям [139], анализ полноты покрытия исходного кода тестами), проверка корректности работы программы на ограниченном наборе тестовых данных не может обеспечить корректность её работы на всех наборах данных.

Статический анализ — верификация проводится с помощью автоматического анализа исходного кода путём построения модели программы и сравнения ее с моделью, построенной на основе спецификаций. При этом могут быть заданы различные наборы правил для верификации различных видов требований. Чаще всего статический анализ используется для проверки программы на наличие технических ошибок, таких, например, как некорректная работа с памятью или переполнение буфера [107]. На настоящее время средства статического анализа активно развиваются и начинают широко использоваться. Примерами таких средств могут служить Coverity SAVE [13] и отечественные коммерческие разработки PVS Studio [126] и CppCat [127]. Средства статического анализа предоставляют широкие возможности с точки зрения верификации программного обеспечения. Верификация при этом зачастую производится с использованием формальных методов, что позволяет получить более высокие гарантии в рамках области применимости этих средств. Однако автоматические средства статического анализа имеют фундаментальное ограничение, а именно — автоматическое доказательство любых нетривиальных свойств для произвольных программ не представляется возможным, поскольку влечёт за собой разрешимость проблемы останова. По указанной причине результаты работы таких средств не могут не иметь ошибок пер-

вого рода (ложные позитивные срабатывания), либо ошибок второго рода (ложные негативные срабатывания). При этом на практике процент ложных позитивных срабатываний достаточно велик. Это обстоятельство затрудняет использование таких средств, а наличие ложных негативных срабатываний делает невозможным получение строгих гарантий корректности программы.

Формальная верификация — верификация заключается в построении доказательства выполнения требуемых свойств (формальной спецификации) для модели программы (формальной семантики). При этом средства формальной верификации на настоящее время обладают недостатком, аналогичным отмеченному выше для статического анализа. Как следствие, в связи с фундаментальными ограничениями, автоматическое построение доказательства возможно только для достаточно слабых с позиции выразительной мощности (выразительных возможностей) моделей. В этой связи общий подход формальной верификации в настоящее время подразделяется на два класса: верификация на моделях (model checking [25]) и дедуктивная или теоретико-доказательная верификация (deductive verification [46]). Первый класс предполагает автоматические средства верификации на моделях ограниченной сложности, второй — ручные или частично автоматизированные средства, однако на моделях произвольной сложности. Несмотря на то, что исследования в этой области ведутся достаточно давно, в общую современную практику программной инженерии формальная верификация не входит. Тем не менее, только формальная верификация может дать в качестве результата строгое математическое доказательство корректности программы.

Заметим, что в зарубежной терминологии рассматривается также разделение подходов к верификации на два больших класса: динамическая верификация, которая включает в себя запуск программы и анализ данных, которые получаются в результате её выполнения; статическая верификация, которая проводится путём анализа исходного или исполнимого кода программы.

Для того, чтобы процессы разработки программного обеспечения предоставляли максимум возможностей по снижению количества дефектов и ошибок, необходимо

систематическое использование каждого из подходов к верификации на протяжении всего жизненного цикла программного продукта. Однако, как было отмечено ранее, на настоящее время методы и средства формальной верификации не используются в общей практике программной инженерии. В этой связи, с учётом преимуществ этого подхода, актуальны исследования новых моделей, методов и средств формальной верификации, которые способствовали бы её широкому распространению.

На протяжении всего жизненного цикла программное обеспечение, как правило, подвергается модификациям. Необходимость таких модификаций может быть обусловлена, в том числе, следующими факторами: изменение функциональных требований в процессе эксплуатации; исправление обнаруженных дефектов в рамках сопровождения; внедрение новой версии программного обеспечения; переход на новую архитектуру аппаратных и системных программных средств. При длительной эксплуатации программ, общий объем исходного кода в результате таких модификаций существенно возрастает. В частности, для двух рассмотренных в приложении A.1 примеров объем исходного кода увеличился более чем в 5 раз за 18-20 лет. Таким образом, в общем случае однократного проведения формальной верификации исходного кода недостаточно. Этот факт влечёт за собой необходимость использования таких методов формальной верификации, которые предоставляют возможность эффективной модификации доказательств одновременно с внесением изменений в исходный код.

Недостатки существующих широко используемых подходов к верификации (в особенности тестирования) находят своё отражение в статистике плотности дефектов и ошибок в исходом коде программ, представленной в приложении A.2. Согласно этой статистике, в 2012 году для 118 проанализированных проектов (среднего и крупного размера) с открытым исходным кодом средняя плотность обнаруженных средством статического анализа дефектов составляла 0.69 на тысячу строк исходного кода. Авторами исследования это оценивается как хороший результат. Однако в [160, Лекция 10] приводятся сведения, согласно которым достижима на практике плотность ошибок менее 0.20, а уровень порядка 0.05 ошибок на тысячу строк исходного кода считается близким к минимальному достижимому существующими средствами. На недостатки

существующих подходов к верификации программ указывают многие специалисты в области программной инженерии. Некоторые из таких аргументов приведены далее.

В отчёте [66] Института Программной Инженерии Университета Карнеги-Мелона В. С. Хамфри (W. S. Humphrey) отмечает следующие факты:

- объем исходного кода программных продуктов на протяжении последних 40 лет (на момент проведения исследования в 2001 г.) рос экспоненциально;
- количество ошибок на одну тысячу строк исходного кода, согласно мнению автора, может считаться постоянным;
- тестирование, как наиболее распространённый в настоящее время способ контроля качества программного обеспечения, с ростом объёма системы становится неэффективным, т.к. необходимое для полного покрытия исходного кода количество тестов растёт экспоненциально относительно объёма программы;
- сложность используемых языков программирования общего назначения повышалась в рамках рассматриваемого периода времени, однако это не способствовало решению задач, связанных с повышением надёжности программного обеспечения.

Б. Боэм (B. Boehm) в статье [16] рассматривает историю западной программной инженерии, в том числе и историю применения формальных методов в программной инженерии. Согласно публикациям, на которые ссылается Боэм, использование формальных методов как средства для снижения количества дефектов в программном обеспечении началось в 1970-х годах. Применялись формальные методы в двух направлениях: доказательство корректности программ и построение заведомо корректных программ. Однако уже к середине 1990-х годов широкое использование формальных методов при разработке программного обеспечения общего назначения почти прекратилось в связи со значительным увеличением скорости изменения требований к программному обеспечению. Таким образом, с точки зрения истории западной программной инженерии, существенным недостатком формальных методов является снижение скорости разработки.

В.А. Васенин в статье [141] отмечает развитие формальных методов разработки программного обеспечения как один из существующих вызовов и перспективных под-

ходов в инженерии программ. Такие подходы, по его мнению, способны качественно улучшить состояние индустрии разработки программного обеспечения как с позиций повышения эффективности технологических процессов, так и в плане подготовки специалистов, обладающих высокой квалификацией и способных применять на практике существующие и перспективные методы. В части использования формальных методов отмечается следующее: «…на этапах разработки и внедрения в практику, сопровождения и модернизации программного обеспечения сложно организованных систем происходят перманентные коррекции технических требований к нему, изменение профилей стандартов и других атрибутов. Такие коррекции способны привести к изменению функциональных возможностей программной системы, качества исполнения ими функций, появлению у них тех или иных уязвимостей, других негативных последствий. В случае любой из перечисленных коррекций необходима оперативная проверка соответствия требований к модифицированной системе, её свойств тем требованиям и свойствам, которыми она обладала до модификации. Математические модели являются самым эффективным средством оперативной проверки такого соответствия, позволяя либо путём получения аналитических оценок, либо в ходе имитационного моделирования хотя бы в первом приближении дать ответы на возникающие вопросы. Таким образом, обеспечивается перманентный (постоянный и оперативный) контроль за корректностью (соответствием требованиям) программной системы на протяжении её жизненного цикла».

Проведение формальной верификации на предмет соответствия программного кода автоматизированной системы заданным спецификациям содержится в различных стандартах в области контроля качества и безопасности программного обеспечения. В частности, проведение формальной верификации требуется в рамках серии стандартов ГОСТ Р ИСО МЭК 15408 «Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий» [147–149] и необходимо для соответствия наивысшему оценочному уровню доверия (ОУД7).

Аргументы, приведённые в перечисленных публикациях и докладах, а также требования, содержащиеся в нормативных актах, определяют актуальность исследований моделей, методов и средств верификации программного обеспечения с использованием формальных методов. При этом современные тенденции развития аппаратного обеспечения показывают необходимость реинжиниринга, т.е. внесения в исходный код уже существующих и активно применяемых на практике программных комплексов изменений, значительно изменяющих процесс выполнения программы. На настоящее время целью реинжиниринга может являться, прежде всего, эффективное использование ресурсов новых платформ и вычислительных сред. К таким платформам и средам могут быть отнесены, в частности: SIMD-архитектуры[1], такие как графические ускорители (GPGPU) и вычислительные ускорители Intel Xeon Phi; специализированные аппаратные средства, такие как процессоры обработки сигналов (DSP), программируемые логические интегральные схемы (FPGA) и интегральные схемы специального назначения (ASIC); распределенные вычислительные среды (Grid Computing, Cloud Computing). В дорожной карте развития программного обеспечения для перспективных суперкомпьютеров производительностью порядка экзафлопс[2] [73] отмечается, что перспективные высокопроизводительные платформы, по всей видимости, будут основаны на гетерогенной архитектуре, что потребует реинжиниринга большого количества существующего и активно использующегося на практике программного кода для адаптации к таким платформам. Активно ведутся также исследования по разработке языков программирования и алгоритмов для квантовой информатики [55, 108, 111]. Опыт реинжиниринга крупных программных комплексов [112, 165, 175, 176] показывает, что проведение модификаций кода для его эффективного исполнения на другой платформе (в частности, так называемое распараллеливание) традиционными средствами требует значительных трудозатрат, которые обусловлены большим объёмом модифицируемого кода. В этой связи следует отметить новые подходы к реинжинирингу [145, 167], которые позволяют выделить и сохранить фраг-

[1] SIMD — Single Instruction Multiple Data, вычислительная архитектура, в рамках которой возможно параллельное исполнение элементарных операций на нескольких наборах операндов.

[2] 10^{18} операций с плавающей точкой в секунду

менты кода, связанные с предметной областью, и вносить модификации с меньшими трудозатратами. Тем не менее, в процессе реинжиниринга процесс исполнения исходного кода существенно изменяется и выполнение функциональных требований в общем случае может не сохраниться. Это замечание определяет актуальность исследований, позволяющих проводить формальную верификацию в процессе реинжиниринга.

Для проведения формальной верификации, как правило, используется формальная модель (семантика) языков программирования, применяемых при разработке программы. В этой связи следует отметить тот факт, что в настоящее время немалая часть программных продуктов разрабатывается с использованием нескольких языков программирования одновременно. Это подтверждается данными по проектам с открытым исходным кодом, представленными в приложении A.3. Таким образом, на практике для верификации может быть необходимо рассматривать формальную семантику одновременно нескольких языков программирования, следовательно, встаёт вопрос о формальной модели, допускающей такое представление.

Объектом настоящего диссертационного **исследования** являются математические модели и основанные на них средства разработки и анализа исходного кода программ с целью удовлетворения требований по их формальной верификации. Такие модели и средства должны допускать использование нескольких языков программирования.

В качестве **предмета исследования** рассматриваются процессы описания формальных моделей программ и языков программирования, используемых в исходных текстах программ.

Целью диссертационного исследования является разработка математических моделей и основанных на них программных средств, которые, в свою очередь, предназначены для построения формальных моделей программ и верификации их функциональных свойств с помощью промежуточного представления таких программ на основе λ-исчисления с зависимыми типами.

Для достижения этой цели сформулированы и решены следующие, перечисленные далее **задачи**.

1. Исследовать существующие и разработать новую разновидность λ-исчисления с зависимыми типами для его использования в качестве промежуточного представления при описании формальной семантики различных языков программирования.
2. На основе новой разновидности λ-исчисления с зависимыми типами разработать макеты языка и программного средства для построения формальных моделей и верификациц функциональных свойств программ.
3. Исследовать возможность использования разработанного макета для описания формальной семантики различных востребованных языков программирования.
4. Исследовать возможность использования разработанного макета для описания различных аспектов формальной семантики программ, в частности — механизмов описания параллельных программ.

Научная новизна работы заключается в том, что автором получены следующие далее результаты:

1. предложена новая разновидность лямбда-исчисления с зависимыми типами, обеспечивающая поддержку нетривиальных типов идентичности путём введения дополнительных, не рассматривавшихся ранее правил редукции для элементов типов идентичности;
2. на основе предложенной разновидности исчисления реализованы макеты языка программирования и программного средства, которые могут представлять спецификации, существенно использующие нетривиальные типы идентичности с помощью предложенных автором правил редукции;
3. построена новая модель статической формальной семантики промежуточного кода, соответствующего подмножеству стандарта ECMA-335, которая поддерживает обобщённые типы, соответствующие четвёртой редакции стандарта;

4. построена новая формальная модель динамического параллельного исполнения программ, основанная на модификации формальной семантики языка программирования.

Теоретическая и практическая значимость. Результаты исследований, представленные в настоящей диссертации призваны способствовать развитию моделей, методов и средств разработки и модификации программного обеспечения, позволяющие более эффективно реализовать предъявляемые к нему требования с помощью механизмов формальной верификации. Разработанный автором макет языка и средства верификации допускают более высокоуровневое представление программ, написанных на нескольких языках программирования. Такой язык и средства могут эффективно применяться при верификации существующих программ. Предложенная разновидность λ-исчисления с зависимыми типами и реализация базового языка на её основе может применяться для исследований в области компьютерной математики, в частности, при развитии гомотопической теории типов [121].

Область исследования. Диссертация соответствует паспорту специальности 05.13.17 «Теоретические основы информатики» по следующим областям исследований:

- п. 2. «Исследование информационных структур, разработка и анализ моделей информационных процессов и структур.»;
- п. 5. «Разработка и исследование моделей и алгоритмов анализа данных, обнаружения закономерностей в данных и их извлечениях … .»;
- п. 14. «Разработка теоретических основ создания программных систем для новых информационных технологий.»

Методика исследования. В работе применяются методы теории категорий, теории доменов, математической логики и теории доказательств, теории графов, а также методы программной инженерии и функционального программирования.

Апробация работы. Основные результаты диссертации докладывались на следующих конференциях и семинарах:

- международная конференция «Мальцевские чтения — 2014», секция алгебро-логических методов в информационных технологиях (Институт математики им. С.Л. Соболева СО РАН, г. Новосибирск, 10–13 ноября 2014 г.);
- научные конференции «Ломоносовские чтения — 2012, 2013, 2014», секция механики (Московский государственный университет имени М.В. Ломоносова, г. Москва, 16–25 апреля 2012 г., 15–23 апреля 2013 г., 14–23 апреля 2014 г.);
- Четвёртая научно-практическая конференция «Актуальные проблемы системной и программной инженерии (АПСПИ — 2015)» (МИЭМ НИУ ВШЭ, г. Москва, 20–21 мая 2015 г.);
- Третья конференция «Информационная безопасность АСУ ТП КВО» (РАНХ и ГС, г. Москва, 29–30 января 2015 г.);
- семинар «Проблемы современных информационно-вычислительных систем» под руководством д.ф.-м.н., проф. В.А. Васенина (2011, 2013, 2014 г., Механико-математический факультет Московского государственного университета имени М.В. Ломоносова, г. Москва);
- расширенный семинар «Методы суперкомпьютерного моделирования» (Институт космических исследований РАН, г. Таруса, 1–3 октября 2014 г.);
- семинар «Интеллектуальные системы» под руководством к.т.н. Ю.А. Загорулько (2014 г., Институт систем информатики им. А.П. Ершова СО РАН, г. Новосибирск);
- семинар «Теоретические проблемы программирования» под руководством д.ф.-м.н., проф. Р.И. Подловченко и д.ф.-м.н., проф. В.А. Захарова (2014 г., Факультет вычислительной математики и кибернетики Московского государственного университета имени М.В. Ломоносова, г. Москва);
- семинар отдела «Технологий программирования» под руководством д.ф.-м.н., проф. А.К. Петренко (2014 г., Институт системного программирования РАН, г. Москва).

Публикации. По теме диссертации опубликовано 11 научных работ [181–191], в том числе — пять статей (181–186) опубликованы в изданиях из Перечня, рекомендуемого ВАК. Две работы (181, 183) переведены на английский язык и изданы в журналах, индексируемых Web of Science.

Достоверность полученных результатов и выводов диссертации определяется строгими математическими доказательствами и практическими испытаниями, а также тем фактом, что основные результаты и выводы опубликованы в девяти печатных работах и прошли апробацию в форме докладов на ряде научных и научно-практических конференций и семинаров.

Структура и объем диссертации. Работа состоит из введения, четырёх глав, заключения, списка рисунков, таблиц и листингов, списка литературы. Объем диссертации — 153 страницы, приложений — 39 страниц. Список литературы включает 191 работу.

В *первой главе* представлен библиографический обзор и критический анализ публикаций, отражающих предысторию (раздел 1.1) и современное состояние исследований (раздел 1.2) в области разработки формальных моделей программ и языков программирования, а также в области моделей вычислений, основанных на λ-исчислении. По итогам обзора в разделе 1.3 формулируется постановка четырёх задач исследования. Решению этих задач посвящены последующие главы диссертации.

Во *второй главе* представлено решение первой задачи — исследование существующих и разработка новой разновидности λ-исчисления с зависимыми типами для его использования в качестве промежуточного представления при описании формальной семантики различных языков программирования. Раздел 2.1 имеет вводный характер и содержит определение одной из известных разновидностей λ-исчисления с зависимыми типами на основе Расширенного Исчисления Конструкций [83]. Предложенные автором новые правила редукции для расширенной интерпретации типов идентичности в исчислении указаны в разделе 2.2 (определения 2.11, 2.14–2.17). Для новых правил автором доказываются свойства, согласно которым такие правила сохраняют типизацию. Указанные свойства сформулированы и доказаны автором

в леммах 2.4–2.7, с использованием которых в качестве итогового результата сформулирована и доказана теорема 2.2. Кроме того, доказана теорема 2.1 о возможности редукции нейтральных термов.

Третья глава посвящена решению второй задачи — разработке макетов языка программирования и формальной спецификации, а также программного средства, которые, в свою очередь, предназначены для построения формальных моделей и верификации свойств программ. В первую очередь, в разделе 3.1 разработанная в предыдущей главе разновидность λ-исчисления с зависимыми типами дополняется термами, с использованием которых могут быть заданы значения. В число таких термов входят стандартные типы конечных множеств и натуральных чисел. Кроме того, в соответствие с основными положениями гомотопической теории типов [121], определяется понятие высших индуктивных типов, с использованием которых определяются индуктивные и коиндуктивные типы. Раздел 3.2 описывает основные особенности реализации макета программного средства на основе полученной модели исчисления. Отдельные аспекты реализации проверяются на примере типа списков (пример 3.1) и формальной записи известного парадокса (раздел 3.2.6), а также с использованием формального доказательства в среде Coq (приложение B).

В *четвёртой главе* с использованием разработанных автором макетов языка и программного средства строится модель статической формальной семантики управляемого кода стандарта ECMA-335 [41], что соответствует решению третьей задачи исследования. Раздел 4.1 содержит основные сведения о стандарте. Основные положения модели статической формальной семантики и процесс построения модели рассматриваются в разделе 4.2. В число описанных в этом разделе результатов входят, в первую очередь, окружения статической семантики, которые ставят ссылкам на различные объявления языка в соответствие объекты, соответствующие статической семантике таких объявлений. Раздел 4.3 включает описание динамической семантики подмножества стандарта ECMA-335. Такое подмножество позволяет решить задачу описания динамической семантики фрагмента кода, входящего в состав программного комплекса математического моделирования физических процессов. Описание динамической

семантики основано на известном подходе на основе монад [94, 97], который адаптируется автором для его применения в рамках макета языка и программного средства.

Пятая глава посвящена описанию разработанной автором модели динамического параллельного исполнения программ. Модель позволяет описать квазипараллельное (чередующее атомарные операции в произвольном порядке) исполнение программ в форме модификатора семантики языка программирования. Модель построена с использованием макета языка и программного средства, построенных в третьей главе, а также — подхода на основе монад, адаптированного к ним в четвёртой главе. В разделе 5.1 представлен язык управления потоком исполнения, который применяется для описания порядка запуска и использования результатов работы вычислительных потоков в программе. Раздел 5.2 описывает преобразователь монад возобновлений [98] и другие монады, которые используются для описания семантики. В разделе 5.3 описывается модель динамического параллельного исполнения программ, состоящая из двух частей – динамической семантики языка управления потоком исполнения и модели управляющего ядра системы. Раздел 5.4 содержит доказательства свойств модели, которые существенным образом опираются на расширенную интерпретацию типов идентичности согласно результатам, представленным во второй главе. В разделе 5.5 приведён пример расширения модели и описания семантики параллельного исполнения фрагмента кода, рассмотренного в четвёртой главе.

В *Заключении* приведены выводы по итогам диссертационного исследования. Отмечены основные направления дальнейших исследований в рассматриваемой области и даны рекомендации по практическому использованию полученных результатов.

Глава 1

К постановке основной задачи исследования

В настоящей главе представлено краткое описание предыстории и современного состояния исследований в области разработки верифицируемого программного обеспечения, реинжиниринга существующих программных комплексов, а также в области программирования (на основе языков разного уровня абстрагирования от платформы).

Первый раздел содержит обзор и критический анализ публикаций с 1932-го по 1989-й годы, тематика которых включает формальные модели вычислений, программ и языков программирования, в том числе — работы по формальной верификации. Обзор выполнен в контексте появления и развития языков программирования, с учётом эволюции вычислительной техники и методов программной инженерии. Областью особого интереса в ходе проведённого анализа являлось развитие подходов к описанию вычислений в терминах различных разновидностей λ-исчисления, которое выбрано в качестве базовой теоретической модели в настоящей работе.

Во втором разделе рассмотрено современное состояние исследований в области формальной верификации на основе зарубежных и российских публикаций, вышедших с 1990 г. по настоящее время. Основное внимание уделено подходу к формальной верификации на основе языков программирования с зависимыми типами, однако рассматриваются результаты и на других, смежных направлениях. Обозначены также потенциальные направления дальнейших исследований в этой области.

В последнем, третьем разделе настоящей главы сформулирована цель и даны постановки четырёх задач диссертации.

1.1. Библиографический обзор 1930–1989 гг.

Методы формальной верификации — получения математических строгих доказательств выполнения программой поставленных перед ней требований, находятся в области активного интереса исследователей начиная с 1960-х годов. В то же время, математические теории, которые легли в основу таких методов, были построены существенно раньше, в рамках исследований в области математической логики и оснований математики. Так, первые публикации, которые рассматриваются в настоящей статье относятся к началу 1930-х годов и связаны с Проблемой разрешения, сформулированной Д. Гильбертом. Дальнейшие исследования показали, что полнота некоторых моделей вычислений с позиций набора представимых алгоритмов в обязательном порядке влечёт за собой неразрешимость нетривиальных свойств для представлений алгоритмов в таких моделях, а также противоречивость формальных логик, основанных на такой модели. По этой причине дальнейшие исследования в области математической логики были сосредоточены на формальных системах, менее мощных с позиций представления вычислений. В настоящее время такие системы применяют, в частности, и для решения задачи формальной верификации. На истории развития одной разновидности таких формальных систем, а именно — типизированного λ-исчисления, и сделан акцент в настоящем разделе. Расширенная версия библиографического обзора, рассматривающая представленные на этом направлении публикации в контексте развития методов инженерии программ и появления языков программирования, содержится в статье [143].

1.1.1. Формальные модели вычислений

В 1930-е гг. в рамках фундаментальной математики активно велись исследования, связанные с Проблемой разрешения (Entscheidungsproblem), сформулированная Давидом Гильбертом в 1928 г. [61] и частично решённая в работах А. Чёрча [22] и А. Тьюринга [123]. Именно эти исследования и положили начало формальной теории вычислений. Истоки Проблемы разрешения заключались в кризисе оснований

математики и плане Гильберта. Основная формулировка, из которой была выделена проблема, состояла из следующих трёх вопросов: является ли математика полной, непротиворечивой и разрешимой. Первая и вторая теоремы Гёделя о неполноте [57] дали ответ на первые два вопроса. Для того, чтобы дать ответ на третий вопрос было необходимо, прежде всего, строго определить понятие разрешимости.

Первые с исторической точки зрения результаты, которые следует отметить в связи с задачами формальной верификации и построения формальных моделей программ — это формальные модели вычислений. На этих результатах в той или иной степени основаны все последующие исследования в указанных областях. К таким результатам следует отнести машины Тьюринга и Поста, а также λ-исчисление Чёрча.

В 1936 году в работе [123] Алан Тьюринг предложил концепцию абстрактного вычислителя, который в дальнейшем получил название «машина Тьюринга» (в 1938 году были опубликованы исправления к оригинальной статье [124]). Машину Тьюринга можно неформально описать как совокупность бесконечной в обе стороны ленты, разделённой на ячейки, в которых записаны символы определённого алфавита и управляющего устройства. Управляющее устройство имеет состояние, читающую и пишущую головку и набор правил перехода, задающих для входных состояния и символа, считанного с ленты, выходные состояние, символ, записываемый на ленту, и команду изменения позиции головки. Эмиль Пост в работе [105], вышедшей через пять месяцев после публикации первой работы Тьюринга, независимо предложил более простой вариант такой машины, основные отличия которой от модели Тьюринга можно описать следующим образом: алфавит ограничивается двумя символами («отмеченная» и «неотмеченная» ячейка), а правила перехода задаются с помощью фиксированного набора из шести инструкций.

Исходные результаты Алонзо Чёрча были впервые опубликованы в 1932 году как формальный язык, предлагаемый к использованию в качестве оснований математики [21]. Однако в 1935 году Клини и Россером был сформулирован парадокс [76], демонстрирующий противоречивость результатов Чёрча. В 1936 году Чёрч изолировал аспекты формальной системы, относящиеся к вычислимости в

публикации [23]. Парадокс Клини-Россера и его упрощение в виде парадокса Карри для λ-исчисления позволяют установить важный факт. Его суть в том, что если некоторая модель вычислений допускает незавершимость программ, представимых в ней, то такая модель является противоречивой как логическая система. В 1940 году Чёрч опубликовал работу [24], в которой рассматривалось λ-исчисление с простыми типами. В этой работе принципы λ-исчисления объединялись с теорией типов в стиле Principia mathematica Рассела и Уайтхеда.

Для ограничения выразительной мощности системы, в типизированном λ-исчислении к термам приписываются метки, называемые типами. Типы формируются согласно определённым правилам, а именно: задано множество базовых типов и комбинатор типов «$a \to b$», с использованием которого определяется тип функции между значениями типов a и b. Далее, вводится отношение типизации, обеспечивающее соответствие между типом аргумента функции и типом правой части в операции приложения. Такое исчисление обладает рядом важных свойств, в числе которых — сильная нормализация. Согласно свойству сильной нормализации любой корректно типизированный терм исчисления за конечную последовательность редукций приводится к нормальной форме, единственной вне зависимости от порядка редукций. Отметим следующие две формальные модели вычислений, которые появились в шестом десятилетии XX века, а именно — модель частично-рекурсивных функций Клини [75] и нормальные алгорифмы Маркова [163]. Последняя модель оказала большое влияние на развитие систем переписывания термов, в том числе, на ней был основан язык программирования Рефал. Основные идеи модели Маркова встречаются также в конструкциях сопоставления с образцом современных высокоуровневых языков программирования (таких как Wolfram Language или Haskell).

В 1952/53 учебном году на кафедре вычислительной математики МГУ А.А. Ляпунов прочитал курс лекций «Принципы программирования», который стал первым курсом по программированию в СССР [171] (сокращённые материалы курса изданы в 1958 году [161]). В рамках этого курса был предложен новый операторный метод описания программ, который далее был формализован Ю.И. Яновым [180] и получил

название «схемы программы». Этот курс положил основы отечественной теории программирования.

Важнейшим с точки зрения фундаментальных ограничений формальной верификации является результат Райса о неразрешимости проблемы эквивалентности вычислимых функций [110], опубликованный в 1953 году. Теорема Райса гласит, что для любого нетривиального свойства вычислимых функций алгоритмически неразрешима задача установления того факта, вычисляет ли произвольный алгоритм функцию с таким свойством. В соответствие с этой теоремой, не может существовать универсальных автоматических методов верификации для нетривиальных свойств. Отметим, что под нетривиальным в теореме Райса понимается такое свойство, множество удовлетворяющих которому вычислимых функций не является пустым и не равно всему множеству вычислимых функций.

1.1.2. Аксиоматическая, операционная и денотационная семантика

Распространение языков программирования в 1960-х привело к осознанию факта, который можно сформулировать следующим образом: «Непосредственной связи между программами, записанными в каких-либо языках программирования, и абстрактными моделями вычислений нет. Для того, чтобы получить возможность доказательства свойств программ, а не алгоритмов, необходимо некоторое связующее звено между исходными кодами и математическими моделями». Вероятно, именно с этим обстоятельством связано появление к концу 1960-х годов трёх основных подходов к описанию формальной семантики программ, а именно — операционной, аксиоматической и денотационной семантики.

Подход на основе операционной семантики описывает значение конструкций языка программирования с помощью непосредственного указания способа вычисления каждой конструкции в терминах абстрактного вычислителя. Впервые концепция операционной семантики была сформулирована и использована при разработке языка программирования Algol-68 [81].

Аксиоматический подход к описанию семантики программ был разработан Флойдом [45] и Хоаром [62] в 1967-69 гг. Аксиоматическая семантика не определяет непосредственно значение, как некоторое заданное число, промежуточный или конечный результата вычисления отдельных конструкций или программы в целом. Вместо этого задаётся определённый набор свойств отдельных её конструкций. Свойства конструкций описываются как аксиомы и правила вывода некоторой формальной системы, а свойства программы, в свою очередь, строятся как выводы из аксиом и правил вывода данной системы.

Денотационная семантика описывает значение программы с помощью специальных объектов, называемых денотациями. Основные положения этого подхода были разработаны Д. Скоттом и К. Стрейтчи в конце 1960-х (первые материалы были опубликованы в 1970 г. [114]). Существенный вклад в теорию доменов, заложившей основу денотационной семантики, внёс Ю.Л. Ершов в рамках серии работ по A-пространствам, первая из которых вышла в 1972 г. [150]. В роли денотаций могут выступать числа, функции, или некоторые конструкции, такие как домены Скотта-Ершова. Денотационная семантика не определяет последовательность шагов, которые необходимы для вычисления выражений языка. С одной стороны, это является преимуществом, так как позволяет исследовать семантику программы как некоторый цельный объект, а не как последовательность шагов, что происходит в случае операционной семантики. С другой стороны, для императивных языков, в которых порядок выполнения действий имеет значение, это обстоятельство усложняет модель. Введение в методы описания денотационной семантики языков программирования изложено в [113].

1.1.3. Темпоральная логика и полиморфное λ-исчисление

Развитие предложенных в конце 1960-х годов подходов к описанию семантики языков программирования происходило в нескольких направлениях, но наиболее значимые результаты были получены для описания семантики параллельных вычислений. В 1971 г. Г. Бекич предложил концепцию алгебры процессов, которая описывала семантику «квазипараллельного исполнения процессов» с использованием

операторов последовательной и квазипараллельной композиции [58]. В рамках этой концепции исполнение параллельных процессов представлялось как недетерминированное слияние элементарных шагов (т.е. исполнение рассматривалось как последовательное с произвольным порядком). К порождённому этой концепцией классу можно отнести, в том числе, такие широко распространённые модели конкурентного исполнения, как CSP (Communicating Sequential Processes) Хоара [63] и CCS (Calculus of Communication Systems) Милнера [92]. В представленной в настоящей работе модели динамического параллельного исполнения программ (глава 5) используется аналогичный подход. Его суть в том, что семантика параллельно исполняющихся процессов строится как недетерминированное слияние последовательностей атомарных шагов.

К концу 1970-х годов можно отнести также первые публикации, рассматривающие использование темпоральной логики для верификации программ, в том числе, линейную темпоральную логику (LTL) [103]. В последующие годы, как следует из материалов следующего подраздела, подход к верификации на основе темпоральной логики будет использован как основа при разработке методов верификации на моделях (model checking), одного из основных и широко используемых в настоящее время подходов к верификации программ.

Для описания семантики рекурсивных программ было предложено использование наименьших неподвижных точек [39, 99]. С позиций аксиоматической семантики важным результатом стало использование подхода слабейших предусловий (в рамках общего подхода преобразователей предикатов), предложенное Дейкстрой в 1975 г. [40]. Этот подход получил широкое применение в средствах статического анализа программ на предмет выполнения утверждений о состоянии в процессе их выполнения.

К 1970-м годам относится появление важного расширения типизированного λ-исчисления. Это полиморфное λ-исчисление, называемое также типизированным λ-исчислением второго порядка ($\lambda 2$) или, согласно исходному названию, System F (Система F). Основной особенностью этого расширения является возможность конструирования термов, зависящих от типов, то есть, абстрагирование от конкретных типов. Полиморфное λ-исчисление считается формализацией понятия парамет-

рического полиморфизма в языках программирования, впервые рассмотренного К. Стрейтчи в конце 1960-х (курс лекций повторно опубликован в 2000 г. [116]). Эта система, наравне с её расширением — полиморфным λ-исчислением с конструкторами типов (System F_{ω}), послужила основой для таких языков программирования, как Haskell и семейство языков ML.

1.1.4. Исчисление конструкций

С конца 1980-х годов верификация на модели [25] стала одним из наиболее распространённых подходов к формальной верификации программного обеспечения. Поскольку в настоящей работе рассматривается другой подход, а именно, дедуктивная верификация и программирование с зависимыми типами, в последующих разделах библиографии работы по верификации на модели не рассматриваются. В качестве дальнейших источников по этому подходу может быть рекомендована книга «Principles of Model Checking» [9], а также труды ежегодных международных конференций «Verification, Model Checking, and Abstract Interpretation» и симпозиумов «Model Checking Software».

К концу 1980-х гг. следует отнести основные теоретические результаты, связанные с полиморфным λ-исчислением второго порядка с зависимыми типами (Исчислением Конструкций, Calculus of Constructions). Прежде всего, в этой связи следует отметить работы Тьерри Кокана. В статье [28] рассматривается парадокс Жирара применительно к таким исчислениям. Как один из способов преодоления парадокса предложено использование иерархии универсумов. В работе [30] впервые представлено доказательство строгой нормализации для определённого в ней же Исчисления Конструкций. Отчёт INRIA [29] содержит предложения по дальнейшей работе над формальной системой, в том числе, по введению индуктивных типов. Из числа других исследователей следует отметить Чж. Луо, которым опубликована разновидность Исчисления Конструкций ECC [83], дополненную конструктором типов зависимых сумм.

В публикации [85] рассмотрено описание модулей (в реализации, используемой, в том числе, языками семейства ML) с позиций зависимых типов. Результаты аналогич-

ного подхода для языка SOL на базе λ-исчисления второго порядка представлены в работе [93]. В курсе лекций [87], прочитанном Пером Мартин-Лёфом в 1980 г., рассматривается интуиционистская теория типов, которая использует запись, аналогичную λ-исчислению с зависимыми типами для описания логического исчисления. Таким образом, интуиционистскую теорию типов можно рассматривать как логическую интерпретацию λ-исчисления с зависимыми типами. В качестве введения в теорию типов может быть рекомендовано [122].

Работа [100] посвящена извлечению программ исчисления F_ω из термов-доказательств Исчисления Конструкций. Внимание уделяется, в частности, вопросам удаления нерелевантной с позиций вычислений информации, такой как типы или доказательства информативных утверждений. Для последнего используется специальный тип Prop, который присваивается всем информативным утверждениям и позволяет в процессе извлечения отличать из от релевантных с позиций вычислений функций, имеющих тип Set.

Практический аспект использования λ-исчисления с зависимыми типами, а именно, реализация процедуры частичного разрешения частного случая формул (unification problem) в разновидности исчисления без полиморфизма и типов высших порядков, рассматривается в [42]. Такая процедура используется, например, для автоматизированного доказательства утверждений и, в более общем случае, для сопоставления с образцом в языках программирования.

Связь различных разновидностей λ-исчисления, в том числе, исчисления с зависимыми типами, математической логики, теории категорий и денотационной семантики является предметом анализа в статье [82]. Статья содержит обширную библиографию, включающую большое количество публикаций в предметной области за 1970-1988 гг.

Один из возможных алгоритмов проверки типов для Исчисления Конструкций представлен в статье [104]. Этот алгоритм был разработан для среды автоматизированных доказательств LEGO. Существенной особенностью этого алгоритма является

шаг проверки ограничений на уровни типов; аналогичный подход используется и в настоящей работе.

1.1.5. Выводы по результатам библиографического обзора

В настоящем разделе в исторической хронологии кратко представлен анализ публикаций, имеющих отношение к теме диссертации. Представлены результаты исследований — от первых формальных моделей вычислений 1930-х годов и заканчивая, возможно, одним из центральных для методов формальной верификации делением на теоретико-модельные и теоретико-доказательные методы формальной верификации. В рамках теоретико-модельных методов рассматриваются модели программ с конечно описываемым множеством состояний, чаще всего — автоматные модели. Для таких моделей возможна автоматическая верификация, однако приведение программ к таким моделям в общем случае является отдельной, весьма нетривиальной задачей. В рамках теоретико-доказательных (дедуктивных) методов доказательства свойств программ строятся вручную, с частичной автоматизацией, как выводы в некоторой логике, как правило — логике высшего порядка. На настоящее время известны результаты успешного применения к задачам формальной верификации методов каждого из классов. Тем не менее, наиболее сложным на этом направлении вопросом остаётся вопрос об эффективном представлении программ и их спецификаций в виде, допускающем эффективное использование тех или иных методов формальной верификации.

Хотя основные подходы к описанию формальной семантики языков программирования сформировались к началу 1970-х годов, применение таких подходов к существующим распространённым языкам программирования общего назначения на практике на настоящее время остаётся сложной задачей. Наличие формальной семантики не требуется, в частности, международными комитетами по стандартизации (такими как ISO/IEC JTC1 или ECMA) при утверждении стандарта языка, а исследования в области построения формальной семантики существующих языков носят, в первую очередь, инициативный характер.

1.2. Современное состояние исследований

Как было отмечено в предыдущем разделе, к 1980-м годам методы формальной верификации разделились на два условных класса — теоретико-модельные и теоретико-доказательные (дедуктивные). В случае теоретико-модельных методов программа описывается в терминах некоторой модели, допускающей исчерпывающую проверку состояний на предмет выполнения требуемых свойств (model checking). Для теоретико-доказательных, или дедуктивных методов доказательство выполнения свойств получают как вывод в некоторой формальной системе, в которой задана формальная семантика программы. В последние годы в рамках дедуктивной верификации сформировалось отдельное направление — программирование с зависимыми типами, основанное на наблюдении, получившем название «соответствие Карри-Говарда». Согласно этому наблюдению, доказательства, полученные методом естественной дедукции в его интуиционистской разновидности могут быть интерпретированы в терминах типизированного λ-исчисления. На базе типизированного λ-исчисления, таким образом, могут быть построены среды автоматизации математических доказательств, язык формальной спецификации которых в то же время является статически типизированным функциональным языком программирования. Формальная семантика языков программирования в рамках таких сред может быть представлена в виде интерпретатора. Это обстоятельство может существенно упростить задачу получения формальной модели программы и дальнейшей работы с ней.

В настоящем разделе рассматривается современное состояние исследований в области дедуктивных методов формальной верификации, в частности, программирования с зависимыми типами. При этом упоминаются также связанные с ними фундаментальные работы, результаты которых важны для области формальной верификации программ в целом. Следует отметить, что результаты, относящиеся к методам model checking выходят за рамки настоящей работы. В связи с высокой популярностью методов model checking относительно других подходов к формальной верификации, характеристика современного состояния исследований на этом

направлении потребовала бы отдельного исследования. На основе приведённых публикаций обосновано направление исследований настоящей работы. Полная версия анализа современного состояния исследований представлена в статье [144].

1.2.1. Индуктивные и коиндуктивные типы (1990-е гг.)

Первый из полученных в 1990-х гг. результатов в области формальной семантики языков программирования, который следует отметить особо — это модель денотационной формальной семантики подмножества языка C (стандарта ANSI C) [97]. В этой работе представлена модель формальной семантики существенного подмножества языка программирования C, используемого на практике. Основная структура модели опирается на предложенный в [94] подход к описанию вычислений с использованием монад — специальных конструкций из области теории категорий.

Вопросы применимости формальной верификации в условиях массового распространения программ рассматриваются в [95], где предлагается подход к распространению кода программы под названием proof-carrying code (код, несущий доказательство). В рамках такого подхода совместно с исходным или исполнимым кодом программы распространяется сертификат — доказательство выполнения кодом предъявляемых к нему требований. Сертификат представлен в машиночитаемом виде, что позволяет, в дополнение к предоставляемым криптографическими средствами гарантиям подлинности, получить и строгие математические гарантии выполнения требований. С целью снижения объёма доверенного кода, выполняющего проверку таких сертификатов, позднее в [6] был предложен модифицированный вариант подхода — foundational proof-carrying code (фундаментальный код, несущий доказательство). В такой модификации сертификат представляет собой вывод в некоторой формальной системе (в оригинальном исследовании используется эдинбургская логическая система LF). Выбранная формальная система должна быть достаточно выразительна, поэтому в качестве таких систем рассматриваются, в первую очередь, логики высших порядков и теория типов.

В рамках исследований по развитию исчисления конструкций, следует отметить следующие работы. Расширение исчисления конструкций индуктивными типами [101, 102] позволило рассматривать в рамках этой формальной системы сложные объекты, такие как списки или деревья, а также определять функции, оперирующие такими объектами, без расширения набора аксиом. Независимая формализация индуктивных типов рассмотрена в [3]; модификация, позволяющая интерпретировать ограничения на индуктивные типы непосредственно в рамках формальной системы логики второго порядка представлена в [90]. Дополнение индуктивных типов дуальными им коиндуктивными допускает описание в терминах исчисления конструкций моделей сервисного программного обеспечения, обрабатывающего потенциально бесконечный поток входных данных (запросов) [52].

На основе исчисления конструкций была разработана система автоматизации и поддержки доказательств Coq, одно из первых упоминаний о которой в литературе содержится в [65]. Эта система по настоящее время остаётся основной программной реализацией исчисления конструкций.

1.2.2. Программирование с зависимыми типами (с 2000 г. по настоящее время)

Вероятно, важнейшим на настоящее результатом практического применения средств формальной верификации является разработка верифицированного микроядра операционной системы SeL4 [133]. В [27] рассмотрена подробная ретроспектива проекта (в том числе — с обзором предшествовавших проектов по верификации ядер ОС), которая позволяет, в том числе, оценить трудозатраты на верификацию относительно объёма кода. Формальная спецификация системы безопасности ядра заняла 3 тыс. строк на специализированном предметно-ориентированном языке. Реализация микроядра занимает около 10 тыс. строк кода на языке C. Общий объём доказательств, гарантирующих выполнение микроядром спецификаций системы безопасности составил 100 тыс. строк. Следует отметить, что для верификации ядра SeL4 использовалось средство Isabelle/HOL, основанное на логике высшего

порядка, что показывает возможность использования для аналогичных задач средств, основанных на λ-исчислении с зависимыми типами.

В числе важных результатов по приложению λ-исчисления с зависимыми типами к задаче формальной верификации программного обеспечения в первую очередь следует отметить проект CompCert [78] — сертифицирующий компилятор языка C, разработанный с использованием средства автоматизации и поддержки математических доказательств Coq. В рамках этого проекта выпущено большое количество публикаций по верификации различных аспектов процесса трансляции языка программирования (см., например, [15,74,86]; полный список публикаций на официальном сайте [26]). Заметим, что при реализации CompCert использовался, в том числе, адаптированный к исчислению конструкций подход к описанию семантики на основе монад, который упоминался ранее. В числе близких результатов других исследований можно выделить компилятор языка C, позволяющий верифицировать требования к вычислительной сложности используемых алгоритмов (CerCo [19]), а также исследовательский язык F^* [115], на котором была построена верифицированная реализация распределенного протокола авторизации. С учётом широкого использования методов формальной верификации при разработке аппаратного обеспечения, результатов исследований по построению формальной модели семантики набора команд и схемы работы с памятью распространённых процессорных архитектур [134, 135], а также подходов к верификации методов оптимизации программ [10, 118], можно утверждать, что принципиально возможна разработка верифицирующего компилятора языков программирования в машинный код, при использовании которого гарантируется сохранение доказанных свойств программы при исполнении на целевой аппаратной платформе [7].

Область приложения средств формальной верификации на основе λ-исчисления с зависимыми типами не ограничивается исключительно разработкой программного обеспечения с высокими требованиями к безопасности. Так, в [54] представлено доказательство Теоремы о четырёх красках в системе Coq, которое в то же время является программой, реализующей требуемую раскраску удовлетворяющих условиям теоремы графов. Этот пример является практической иллюстрацией соответствия

Карри-Говарда, упомянутого во введении. На основе средства Coq были реализованы также элементы теории доменов [12].

Из числа направлений теоретических исследований по расширению исчисления конструкций следует выделить следующие. Расширения моделей индуктивных типов вложенными индуктивными [88], индексированными индуктивными [11], индуктивно-рекурсивными [51] и индуктивно-индуктивными [47] типами расширяет выразительные возможности исчисления по описанию корректных по построению сложных по своей структуре программ. Кроме того, результаты последних исследований по семантике таких расширений [43, 48] позволяют упростить реализацию индуктивных и коиндуктивных типов. Одним из современных направлений исследований является гомотопическая теория типов [121] и связанная с ней программа унивалентных оснований математики [129], в рамках которой предполагается использовать расширенную теорию типов как базис для описания математических теорий. В частности, в рамках этого направления, была получена модель теории типов в терминах алгебраической топологии.

Реализация средства поддержки доказательств Matita [131] на основе исчисления конструкций показывает, что возможна компактная реализация (до 5 тыс. строк кода) ядра исчисления конструкций. Такое ядро, в первую очередь, содержит реализацию алгоритма проверки типов, от корректности которой непосредственно зависит корректность проверки доказательств системой.

1.2.3. Российские исследования

Исследования в области формальной верификации программного обеспечения в настоящее время активно ведутся, в том числе, российскими научными организациями. За исключением исследований в области методов model checking, которые, как было отмечено ранее, выходят за рамки настоящей работы, в число основных подходов к описанию формальных моделей программ и формальной верификации, рассматриваемых в настоящее время российскими учёными, входят аксиоматический, операционный и денотационный подходы, а также подходы на основе схем программ.

Общие вопросы построения формальных моделей программ и подходов к решению задачи их формальной верификации рассматриваются в учебном пособии [179].

Методы описания моделей программ на основе схем программ, предложенные А.П. Ершовым и Ю.И. Яновым в 1950-х годах, на настоящее время остаются в области интереса преимущественно российских учёных. Тем не менее, такие методы предоставляют широкий класс возможностей по исследованию преобразований программ, в особенности — при исследовании вопросов корректности различных оптимизаций при их трансляции. В числе работ на этом направлении следует отметить [152] и недавние работы по разрешимости задачи эквивалентности некоторых классов схем программ [172]. Вопросы корректности преобразований программ рассматриваются также в [169], однако вместо подхода на основе схем программ используются логические предикаты на базе абстрактных синтаксических деревьев.

Вопросы описания статической формальной семантики языков программирования на примере стандарта ECMA-335 исследованы в [142]. Подходы к построению формальных моделей программ на основе денотационной семантики в приложении, в частности, к квазипараллельным и параллельным вычислениям рассматриваются в работах [140, 146]. Модели и методы, предложенные в первой работе, были использованы на практике при реализации системы динамического распараллеливания NewTS. В [158] для описания формальной семантики регистровых языков программирования, таких как языки ассемблера, используется операционная семантика. Модификация операционного подхода, получившая название «операционно-онтологический подход» рассматривается в [136].

Известны исследования по описанию аксиоматической семантики подмножеств языков C [106, 137, 154, 168] и C# [166]. Кроме того, в [153] рассматривается пример решения задачи формальной верификации программной реализации алгоритма сортировки с использованием аксиоматической семантики в системе Isabelle/HOL. Средство автоматизированной верификации, использующее методы аксиоматической семантики и алгоритмы решения задачи выполнимости булевых формул, представлено в [178]. Алгебраический подход применяется в [151] для описания семантики

императивного языка программирования на базе Pascal с нетривиальной семантикой операций присваивания. Вопросы описания аксиоматической семантики функционального потокового (data-flow) языка программирования рассматриваются в [157].

В числе методов, развиваемых российскими научными организациями следует упомянуть подход к автоматизированной генерации тестов на основе формальной спецификации программ UniTesK [173]. Как отмечается в [159], подход применялся на практике в таких областях как тестирование операционных систем и реализаций телекоммуникационных протоколов, а также при верификации аппаратного обеспечения. Известны также результаты применения методов автоматизированной генерации тестов с использованием статической семантики языка программирования C к верификации компилятора такого языка [138]. Теория конформности, определяющая теоретический базис для методов тестирования на основе формальных моделей, изложена в [139].

С учётом основной тематики настоящей работы, следует отметить публикации в русскоязычных изданиях, связанные с применением языков программирования с зависимыми типами. В частности, в [164] на языке Agda реализованы конструктивные версии утверждений и методов коммутативной алгебры. В [162] представлена реализация метода решения задачи выполнимости булевых формул на языка Agda в приложении к верификации на основе темпоральной логики. Такой результат показывает возможность применения методов model checking в рамках верификации программ с использованием языков программирования с зависимыми типами.

1.2.4. Выводы по современному состоянию исследований

Методы формальной верификации с использованием языков программирования с зависимыми типами в последние годы являются объектом интереса широкого круга исследователей. Перечисленные примеры результатов недавних исследований, полученных с использованием таких методов, в особенности, компилятор CompCert и микроядро SeL4, верифицированное с использованием близкого к языкам программирования с зависимыми типами средства Isabelle/HOL, показывают возможность их прак-

тического применения для решения задач формальной верификации. Кроме того, перспективы компактной реализации алгоритмов проверки доказательств, построенных на основе языков программирования с зависимыми типами, свидетельствуют о возможности реализации фундаментального кода, несущего доказательство в терминах таких языков. С использованием такого подхода в перспективе может стать возможным, в частности, решение такой актуальной задачи как выполнение автоматической проверки программного обеспечения на предмет отсутствия недекларированных возможностей, непосредственно при развёртывании программных комплексов на целевой платформе. По этой причине основанное на λ-исчислении с зависимыми типами промежуточное представление можно рассматривать как потенциальную основу для предлагаемых средств разработки формальных моделей и верификации программ.

1.3. Задачи диссертационного исследования

Цель и задачи диссертации во Введении сформулированы следующим образом.

Цель: разработать математические модели и основанных на них программные средства, которые, в свою очередь, предназначены для построения формальных моделей программ и верификации их функциональных свойств с помощью промежуточного представления таких программ на основе λ-исчисления с зависимыми типами.

Задачи:

1. Исследовать существующие и разработать новую разновидность λ-исчисления с зависимыми типами для его использования в качестве промежуточного представления при описании формальной семантики различных языков программирования.
2. Разработать на основе выбранных моделей макеты языка и программного средства построения формальных моделей и верификации свойств программ.
3. Исследовать возможность использования разработанного макета для описания формальной семантики различных востребованных языков программирования.

4. Исследовать возможность использования разработанного макета для описания различных механизмов представления формальной семантики программ, в частности — механизмов описания параллельных программ.

Дальнейшая часть настоящего раздела посвящена детализации и описанию постановок перечисленных выше задач диссертационного исследования.

1.3.1. Исследование и разработка математических моделей программ

Как было отмечено в первом разделе настоящей главы, методы дедуктивной верификации подразумевают представление формальной семантики программы в виде формулы некоторого используемого логического исчисления (формальной системы). Одной из традиционных моделей, описывающих вычисления в рамках формальной системы, является λ-исчисление. Использование нетипизированной разновидности λ-исчисления в задачах верификации осложняется тем фактом, что, как логическая система, нетипизированное λ-исчисление противоречиво, так как допускает парадокс Клини-Россера. Лябмда-исчисление с простыми типами непротиворечиво, однако в рамках соответствия Карри-Говарда оно эквивалентно исчислению высказываний — логике, в рамках которой формулировать сложные утверждения о программах не представляется возможным. На настоящее время наиболее активно развиваются разновидности λ-исчисления с зависимыми типами, в том числе, имеются результаты, показывающие его применимость к верификации программ. Для одной из таких разновидностей, а именно для предикативного исчисления индуктивных конструкций, известен результат, показывающий её эквивалентность конструктивной теории множеств (IZF с дополнительной аксиомой о существовании бесконечного количества недостижимых кардиналов). Этот факт позволяет сделать вывод о возможности описания в рамках этой модели математических теорий, на которых основаны методы предметной области. Как следствие, обоснован выбор модели исчисления конструкций и его разновидностей как отправных в исследовании математических моделей в рамках первой задачи диссертационного исследования.

С учётом цели исследования, в постановку первой задачи следует внести следующие дополнения. Во-первых, основным предназначением для исследуемых математических моделей программ является формальная верификация свойств таких программ. В этой связи при рассмотрении разновидностей исчисления конструкций, необходимо учитывать их свойства применительно к процедуре формальной верификации. Поскольку существенной частью процедуры формальной верификации является получение доказательств эквивалентностей (например, эквивалентности программы спецификации), целесообразно в первую очередь рассматривать такие разновидности, в которых получение доказательств эквивалентности упрощено. В частности, подобную расширенную трактовку эквивалентности допускает гомотопическая теория типов. Во-вторых, разрабатываемые математические модели программ должны предназначаться для использования в качестве единого промежуточного представления при описании программ, написанных на нескольких языках программирования. Таким образом, для упрощения модели её представление может быть основано на индексах де Брёйна, когда вместо имён переменных на уровне формальной системы рассматриваются их индексы. Такое представление является более компактным, так как в случае представления с именами переменных необходимо вводить понятие α-эквивалентности формул — эквивалентности при условии переименования переменных.

Поскольку целью работы является разработка моделей и основанных на них программных средств, в состав рассматриваемой задачи не входят вопросы исследования свойств исчисления, таких, в частности, как свойство сильной нормализации.

***Задача* 1.** *Исследовать существующие и разработать новые математические модели промежуточного представления программ, записанных на различных языках программирования, с целью проведения их дедуктивной верификации. Такие модели должны быть основаны на исчислении конструкций (Calculus of Constructions) и допускать расширенную трактовку понятия эквивалентности. Представление модели должно быть основано на индексах де Брёйна. Исследование свойства сильной нормализации не является предметом настоящей задачи.*

1.3.2. Разработка макета языка и программного средства построения формальных моделей и верификации свойств программ

В первом разделе настоящей главы были рассмотрены основные существующие средства разработки и формальной верификации программного обеспечения, использующие в качестве теоретической основы разновидности исчисления конструкций. Такие средства могут быть разделены на две категории: средства автоматизации математических доказательств (Agda, Coq) и языки программирования общего назначения с поддержкой зависимых типов (ATS, Epigram, Idris). При этом ни одно из существующих средств не реализует особенности интерпретации типов эквивалентности в терминах гомотопической теории типов. Следовательно, необходима разработка макета языка программирования, формальной спецификации и программного средства для построения формальных моделей и верификации свойств программ. Такое средство должно считывать наборы формул λ-исчисления, содержащих формальные модели и доказательства, выполнять проверку корректности вывода в таких формулах в терминах заданной формальной системы и выводить результат проверки.

Следует отметить, что разработка полноценного программного средства построения формальных моделей и верификации свойств программ является достаточно трудоёмким процессом и выходит за пределы цели исследования. В этой связи, вторая задача диссертации рассматривается со следующими далее ограничениями, не влияющими на достижение поставленной цели.

1. Для простоты синтаксис разрабатываемого макета языка должен быть задан как Lisp-подобный. Выражения (формы) языка состоят из атомов (целые числа, десятичные дроби, строки и ключевые слова) и списков (конечных последовательностей атомов, сгруппированных круглыми, квадратными или фигурными скобками).
2. Для простоты макет программного средства не должен поддерживать интерактивное доказательство утверждений. Принципиальная возможность режима

интерактивного доказательства подтверждается исследованиями по перечисленным выше существующим программным средствам (Coq, Idris, Agda).

3. Вопросы корректности реализации макета программного средства в рамках настоящего исследования не рассматриваются. Построение версии программного средства, для которой имеется формальное доказательство частичной корректности является одним из направлений дальнейших исследований.

***Задача* 2.** *На основе выбранных моделей разработать макеты языка и программного средства построения формальных моделей и верификации свойств программ. Макеты должны удовлетворять требованиям и ограничениям, описанным в подразделе 1.3.2.*

1.3.3. Исследование возможности использования макета программного средства для описания формальной семантики языков программирования

Поскольку основная часть исследования связана с вопросами описания формальных моделей программ и языков программирования, в рамках третьей задачи требуется сформировать общие методические рекомендации формальной спецификации языков программирования с использованием макетов базового языка и программного средства, которые разработаны при решении второй задачи. В числе известных исследований, перечисленных в первых двух разделах настоящей главы, рассматривался один из подходов к описанию формальной семантики языков программирования, допускающий композицию — подход на основе монад и преобразователей монад (понятия вводятся в рамках теории категорий; в рассматриваемой работе вводится в главе 4). Использование такого подхода в базовом языке представляется целесообразным, с учётом известных результатов исследований, использующих этот подход. Для иллюстрации применимости разработанного средства к задачам описания семантики языков предполагается описать статическую семантику существующего языка программирования Common Intermediate Language (CIL), который определяется стандартом ECMA-335, а также описать с использованием подхода на основе монад и преобразователей монад динамическую семантику подмножества такого языка.

С использованием построенных моделей может быть получено описание формальной семантики фрагмента кода, входящего в состав программного комплекса математического моделирования физических процессов в АЭС. Такие исследования проводились автором в рамках научно-исследовательской работы НИИ механики МГУ №1/404–11 по договору с ОАО «ВНИИАЭС».

***Задача* 3.** *Исследовать возможность использования базового языка для описания формальной семантики языков программирования. К базовому языку должен быть адаптирован подход к описанию семантики языков программирования на основе монад и преобразователей монад. Для иллюстрации применимости базового языка к задачам описания формальной семантики языков программирования общего назначения должен быть рассмотрен пример описания статической семантики языка программирования общего назначения CIL и пример описания динамической семантики подмножества такого языка. С использованием построенных моделей получить описание формальной семантики фрагмента кода, входящего в состав программного комплекса математического моделирования физических процессов в АЭС.*

1.3.4. Исследование динамического параллельного исполнения программ

Как отмечено во введении, одной из современных тенденций развития аппаратного обеспечения является переход к вычислительным средам параллельной и распределённой архитектуры. При реинжиниринге программы для выполнения в вычислительных средах параллельной архитектуры естественным образом возникает необходимость максимально локализовать изменения состояния программы с целью эффективного распределения вычислений по устройствам.

***Задача* 4.** *С использованием макетов языка и программного средства разработать модель динамического параллельного исполнения программ, которая путём реализации статических проверок гарантирует корректность параллельного расчёта с позиций получаемого результата.*

Глава 2

Математическая модель базового языка

Настоящая глава посвящена решению первой задачи диссертационного исследования.

В первом разделе содержится набор правил вывода, с использованием которых макет базового языка задаётся как формальная система. Кроме того, в разделе представлены пояснения, с учётом которых он может служить введением в лямбда-исчисление с зависимыми типами на примере модифицированного Исчисления Конструкций. Рассмотрена связь основных элементов исчисления с математическими утверждениями и доказательствами, а также с конструкциями функционального программирования на основе соответствия Карри-Говарда. В качестве альтернативного введения может быть рекомендована также книга [122].

Во втором разделе представлены новые правила редукции для типов идентичности и аксиомы унивалентности, предложенные автором. Для таких вновь вводимых правил доказаны утверждения, гарантирующие корректность типизации. Новые правила расширяют формальную систему, описанную в первом разделе.

2.1. Расширенное Исчисление Конструкций

В настоящем разделе представлена разновидность Исчисления Конструкций стандартные правила вывода, которые в целом аналогичны общепринятым для исчислений с иерархией универсумов и зависимыми суммами [83].

Основным отличием, которое связано с модификацией модели автором, является введение нелинейной иерархии универсумов. В [83] универсумы индексируются некоторыми целыми числами, а в представленной модели (подмножестве) исчисления терм индексирован различными переменными, которые затем выступают в качестве

вершин в направленном графе уровней универсумов с метками $\{<,\leqslant\}$ на дугах. После завершения проверки типов к полученному графу применяется алгоритм Тарьяна [117] для выделения сильно связанных компонент. Если все сильно связанные компоненты содержат только дуги с меткой $\leqslant$, то терм, подвергаемый проверке, не является противоречивым. В противном случае считается, что терм противоречив и не допускается алгоритмом проверки типов. Поскольку после отождествления элементов сильно связанных компонент, переменным уровней универсумов в непротиворечивом терме могут быть присвоены числовые значения с использованием алгоритмов топологической сортировки, представленное в данном подразделе подмножество исчисления обладает свойством сильной нормализации аналогично [83].

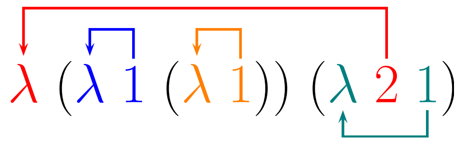
В первом подразделе вводятся основные понятия и определения, а также описывается используемая система записи. В последующих подразделах последовательно вводятся элементы исчисления и приводятся пояснения по использованию таких элементов.

2.1.1. Основные понятия

Исчисление представлено в виде набора правил вывода, определяющих правила формирования корректных термов, и правил редукции, определяющих допустимые схемы преобразования термов (вычислений). В отличие от распространённых способов описания правил вывода для различных разновидностей λ-исчисления, вместо имён переменных в настоящей работе используется позиционная запись в нотации де Брёйна [132]. Это связано с тем обстоятельством, что исчисление используется автором в качестве промежуточного представления. Таким образом, можно избежать вопросов описания α-эквивалентности (эквивалентности термов с точностью до замены имён переменных). Однако, для простоты представления, в правилах вывода и правилах редукции буквами латинского алфавита обозначаются какие-либо произвольные термы, удовлетворяющие условиям правила.

В записи де Брёйна вместо имён переменных используются натуральные или неотрицательные целые числа. Число обозначает, на сколько уровней вверх в

представлении терма в качестве дерева была определена переменная, которую необходимо подставить. На рисунке 2.1 представлен пример использования индексов де Брёйна для нетипизированного λ-исчисления. Стрелками указано, к какому терму абстракции относится ссылка на каждую из переменных. Отметим, что в рамках настоящей работы, в отличие от приведённого на рисунке 2.1 примера, используется нумерация переменных с нуля.



Источник изображения: http://en.wikipedia.org/wiki/File:De_Bruijn_index_illustration_1.svg

Рисунок 2.1: Использование индексов де Брёйна в нетипизированном λ-исчислении

В настоящем подразделе приведены кратко основные понятия, используемые для презентации исчисления. Более подробна информация об индексах де Брёйна приведена, например, в [170, Глава 6], откуда взяты некоторые определения настоящего раздела.

Переменная, номер которой больше, чем глубина терма, называется **свободной переменной**. Например, в терме $\lambda 2$ свободной является переменная 2. **Минимальным уровнем** терма назовём число, на 1 большее, чем максимальная разность между номером свободной переменной и её глубиной. Если в терме нет свободных переменных, будем считать его минимальный уровень равным нулю. Терм $\lambda 2$ при нумерации переменных от нуля имеет минимальный уровень 2, тогда как терм $\lambda\lambda 2$ — минимальный уровень 1. Под **уровнем** терма будем понимать подходящее по контексту число, не меньшее минимального уровня такого терма.

В более сложном случае терм может вводить новую переменную не для всех своих подтермов. Например, определяемый далее терм $\mathrm{split}(C, \mathbf{a}, r)$ вводит новую переменную для подтермов C и r, но не вводит переменную для терма $\mathbf{a}$. Таким образом, уровень термов C и r на 1 выше, чем уровень терма $\mathbf{a}$. Поскольку все термы определяются правилами вывода, в посылках которых указаны правила типизации, для каких подтермов вводятся новые переменные будет понятно из таких правил.

Для лучшего понимания в таких правилах подтермы, для которых не вводятся новые переменные будут выделяться полужирным шрифтом.

Для термов, записанных в нотации де Брёйна определяются операции сдвига и подстановки.

Определение* 2.1.** *Операция* ***сдвига *терма t на d уровней с пропуском c переменных (обозначается как $t\uparrow^d_c$) определяется следующим образом:*

- *для переменных* $i\uparrow^d_c = \begin{cases} i, & \text{если } i < c; \\ i+d & \text{если } i \geqslant c. \end{cases}$
- *для термов вида $F(t_1,\ldots,t_k,s_1,\ldots,s_l)$, где для подтермов t_i вводится новая переменная, а для подтермов $s_1,\ldots,s_l$ не вводится новая переменная:*

$$F(t_1\uparrow^d_{c+1},\ldots,t_k\uparrow^d_{c+1},s_1\uparrow^d_c,\ldots,s_l\uparrow^d_c) \qquad .$$

В результате операции сдвига минимальный уровень терма t увеличивается на d.

В сокращённой записи, $t\uparrow^d$ обозначает сдвиг без пропуска переменных ($t\uparrow^d_0$); $t\uparrow$ обозначает сдвиг на один уровень без пропуска переменных ($t\uparrow^1_0$)

Определение* 2.2.** *Операция* ***подстановки *терма s уровня $(L-j-1)$ вместо переменной с номером j в терме t уровня L (обозначается как $t\downarrow_j s$) определяется следующим образом:*

- *для переменных* $i\downarrow_j s = \begin{cases} s, & \text{если } i = j; \\ i, & \text{если } i \neq j. \end{cases}$
- *для термов вида $F(t_1,\ldots,t_k,s_1,\ldots,s_l)$, где для подтермов t_i вводится новая переменная, а для подтермов $s_1,\ldots,s_l$ не вводится новая переменная:*

$$F(t_1\downarrow_{j+1}(s\uparrow),\ldots,t_k\downarrow_{j+1}(s\uparrow),s_1\downarrow_j s,\ldots,s_l\downarrow_j s) \qquad .$$

Для специального случая терма t уровня не меньше 1, в котором отсутствует переменная 0 введём операцию пустой подстановки ($t\downarrow$). Эта операция уменьшает номера переменных и уровень терма на 1.

Правила вывода сгруппированы по определяемым ими типам и отмечены метками. Для ссылок на правила вывода далее будут использоваться метки. Символом $\mathbb{N}$ обозначим натуральные числа с нулём.

2.1.2. Типы суждений

Определим типы суждений (фактически — отношений на термах и контекстах), которые могут находиться в выводах правил, приведённых в настоящей главе.

***Определение* 2.3.** *В рамках общих правил вывода рассматриваются следующие типы суждений:*

Корректно сформированный контекст Γ *может быть использован в правой части суждения типизации:*

$\Gamma\,\mathrm{wf}.$ *(WF-CONTEXT)* ;

Типизация *означает, что терм* x *в контексте* Γ *удовлетворяет правилам типизации и имеет тип* A*:*

$\Gamma \vdash x : A.$ *(TYPING)* ;

Совместимость *позволяет подставить вместо терма* y *терм* x*:*

$x \leqslant y.$ *(COMPAT)* ;

Редукция *вводит правила преобразования из терма* x *в терм* y*:*

$x \longrightarrow y.$ *(REDUCTION)*.

Заметим, что отношение совместимости несимметрично исключительно с целью того, чтобы в результате проверки типов был сформирован корректный граф уровней универсумов. Если не рассматривать уровни универсумов, то это отношение можно считать симметричным. Учитывая наличие отношения совместимости, использование суждения типизации $\Gamma \vdash t : T$ в посылках правил вывода следует интерпретировать как сокращение для пары суждений $\Gamma \vdash t : T'; T' \leqslant T$, где тип T' – некоторый новый терм.

Далее будет использоваться также общепринятое сокращение $a \longrightarrow^* b$, которое означает, что терм a приводится к виду b путём применения нуля или более правил редукции, из числа определённых на момент использования такой записи.

2.1.3. Контекст

Контекст Γ уровня k представляет собой конечную последовательность термов уровней $0, 1, \ldots, k-1$, соответственно. В нём хранится информация о типах свободных переменных терма уровня k, который рассматривается в таком контексте.

***Определение* 2.4.** *Контекст формируется с использованием следующих правил вывода:*

$$\frac{}{[\,]\mathrm{wf}}\ (\textit{CTX-EMPTY}) \quad ; \quad \frac{\Gamma \mathrm{wf} \quad \Gamma \vdash A : \mathrm{Type}_i}{\Gamma, A \mathrm{wf}}\ (\textit{CTX-ADD}) \quad ; \quad \frac{\Gamma, A \mathrm{wf}}{\Gamma, A \vdash 0 : A}\ (\textit{CTX-DEREF}) \quad .$$

Правило CTX-EMPTY вводит пустой контекст. Правило CTX-ADD позволяет расширить контекст типом A, если A корректно типизирован в этом контексте. Правило CTX-DEREF позволяет использовать информацию контекста при типизации свободной переменной. С помощью операции пустой подстановки с использованием этого правила можно типизировать и другие переменные.

***Определение* 2.5.** *Общие свойства отношения совместимости задаются с использованием следующих правил вывода:*

$$\frac{[i \in \mathbb{N}]}{i \leqslant i}\ (\textit{VAR-COMPAT}) \quad ; \quad \frac{}{x \leqslant x}\ (\textit{IDENTITY-COMPAT}) \quad ; \quad \frac{\Gamma \vdash x : A' \quad A' \leqslant A}{\Gamma \vdash x : A}\ (\textit{TYPING-COMPAT}) \quad .$$

Правила VAR-COMPAT, IDENTITY-COMPAT, TYPING-COMPAT задают основные положения для использования отношения совместимости: переменные с совпадающими номерами совместимы; равные по построению термы совместимы; если типы совместимы, то в правую часть отношения типизаций вместо меньшего типа может быть подставлен больший.

2.1.4. Тип

Одна из предпосылок к введению понятия «тип всех типов», а именно — параметрический полиморфизм типов, тесно связана с интерпретацией λ-исчисления с зависимыми типами как модели функционального языка программирования. Идея параметрического полиморфизма была предложена C. Strachey [116] (переиздание оригинального конспекта лекций от 1967 г.) и впервые реализована в рамках полиморфного

λ-исчисления System F [53] Jean-Yves Girard. Широко известным примером полиморфного типа является тип «список элементов некоторого типа a». С помощью полиморфного определения стандартные операции над списками могут быть заданы только один раз, независимо от того, какой именно тип может быть подставлен вместо a.

Наиболее простым способом введения параметрического полиморфизма в λ-исчисление является добавление импредикативного (ссылающегося на само себя) правила вывода вида Type : Type. Однако, как и в случае наивной теории множеств, подобное импредикативное правило приводит к тому, что в формальной системе, которая использует такое правило, могут быть определены парадоксы, аналогичные парадоксу Рассела. Для λ-исчисления с зависимыми типами, в частности, известен парадокс Жирара [53], следствием которого является неразрешимость проверки типов в системах с таким правилом вывода.

Один из корректных способов определения Type был предложен Т. Коканом в [28]. Этот способ заключается в следующем: вместо единственного «типа всех типов» Type (в англоязычных работах подобные универсальные конструкции называются «universe», что на русский язык можно перевести как «универсум») рассматривается счётная иерархия универсумов $\mathrm{Type}_i, i \in \mathbb{N}$.

Как упоминалось в начале раздела, в рамках настоящей работы для организации иерархии универсумов используются уникальные идентификаторы.

***Определение* 2.6.** *Тип формируется с использованием следующих правил вывода:*

$$\frac{\Gamma \mathrm{wf}}{\Gamma \vdash \mathrm{Type}_i : \mathrm{Type}_j \quad [i < j]}\ \textit{(TYPE-TYPING)} \quad ; \quad \frac{}{\mathrm{Type}_i \leqslant \mathrm{Type}_j \quad [i \leqslant j]}\ \textit{(TYPE-COMPAT)} \quad .$$

Согласно правилу TYPE-TYPING, при использовании правил вывода вида «тип всех типов», в граф уровней добавляется ребро с меткой $<$. Если же используется правило совместимости TYPE-COMPAT, в граф уровней добавляется ребро с меткой $\leqslant$.

2.1.5. Зависимые произведения

Заметим, что в настоящем и последующих подразделах при введении новых типов используется схема правил вывода, аналогичная [122]. Правило вывода, имя которого

имеет суффикс -FORM (правило формирования, formation) задаёт непосредственно конструктор типа. В посылках таких правил находятся параметры, от которых зависит тип, а в выводе — суждение вида T : Type, где T — вновь определяемый тип. Одно или несколько правил вывода с суффиксом или инфиксами -INTRO- (правило введения, introduction) задаёт способы определения элементов такого типа. В выводе таких правил содержится суждение вида $t : T$. Одно или несколько правил вывода с суффиксом или инфиксами -ELIM- (правило удаления, elimination) задаёт способы использования элементов вновь определённого типа для получения элементов других типов. В числе посылок такого правила содержится единственное суждение вида $t : T$, где t — подтерм определяемого терма. Такой подтерм t далее будем называть **удаляемым**. Наконец, на вновь введённых термах с использованием правил с суффиксом -COMPAT- определяется отношение совместимости. За исключением отдельных случаев, такие правила задают способ переноса отношения совместимости на подтермы и не нуждаются в дополнительном пояснении.

***Определение* 2.7.** *Зависимые произведения формируются с использованием следующих правил вывода:*

$$\frac{\Gamma \vdash A : \mathrm{Type}_i \quad \Gamma, A \vdash B : \mathrm{Type}_j}{\Gamma \vdash \Pi A.B : \mathrm{Type}_k \ [i \leqslant k, j \leqslant k]} \text{(PI-FORM)} \quad ; \qquad \frac{\Gamma \vdash \Pi A.B : \mathrm{Type}_k \quad \Gamma, A \vdash b : B}{\Gamma \vdash \lambda A.b : \Pi A.B} \text{(PI-INTRO)} \quad ;$$

$$\frac{\Gamma \vdash x : \Pi A.B \quad \Gamma \vdash a : A}{\Gamma \vdash x \cdot a : (B \downarrow a)} \text{(PI-ELIM)} \quad ; \qquad \frac{A_1 \geqslant A_2 \quad B_1 \leqslant B_2}{\Pi A_1.B_1 \leqslant \Pi A_2.B_2} \text{(PI-COMPAT)} \quad ;$$

$$\frac{A_1 \geqslant A_2 \quad b_1 \leqslant b_2}{\lambda A_1.b_1 \leqslant \lambda A_2.b_2} \text{(LAM-COMPAT)} \quad ; \qquad \frac{x_1 \leqslant x_2 \quad a_1 \leqslant a_2}{x_1 \cdot a_1 \leqslant x_2 \cdot a_2} \text{(APP-COMPAT)} \quad .$$

Зависимое произведение (П-типы, dependent product) представляет собой обобщение понятия типа функций между некоторыми типами A и B. Если стандартный для типизированного λ-исчисления функциональный тип $A \to B$ можно интерпретировать как функцию, преобразующую элемент типа A в элемент типа B, то в случае зависимого произведения, которое обозначается как $\Pi\{x : A\}.B(x)$ и вводится с использованием правила PI-FORM, тип B может зависеть от значения аргумента x. В

качестве простого примера использования зависимого произведения можно рассмотреть следующий тип функций: $\Pi\{n:\mathbb{N}\}.\mathrm{Vector}(\mathbb{N},n)$. Функции такого типа принимают в качестве аргумента натуральное число n и возвращают вектор натуральных чисел длины в точности n. В случае со стандартными функциональными типами (например, в рамках λ-исчисления с простыми типами) выразить такую зависимость между аргументом и типом результата нельзя.

Элементы типов зависимых произведений вводятся с помощью λ-абстракции согласно правилу PI-INTRO. Удаление λ-абстракции производится, как и в случае нетипизированного λ-исчисления, с помощью приложения (применения функции к аргументу), которая вводится правилом PI-ELIM.

2.1.6. Зависимые суммы

***Определение* 2.8.** *Зависимые суммы формируются с использованием следующих правил вывода:*

$$\frac{\Gamma\vdash A:\mathrm{Type}_i \quad \Gamma,A\vdash B:\mathrm{Type}_j}{\Gamma\vdash \Sigma A.B:\mathrm{Type}_k \quad [i\leqslant k, j\leqslant k]}\textit{(SIGMA-FORM)} \quad ; \quad \frac{\Gamma\vdash\Sigma A.B:\mathrm{Type}_k \quad \Gamma\vdash a:A \quad \Gamma\vdash b:(B\downarrow a)}{\Gamma\vdash \mathrm{pair}_{\Sigma A.B}(a,b):\Sigma A.B}\textit{(SIGMA-INTRO)} \quad ;$$

$$\frac{\Gamma\vdash a:\Sigma A.B \quad \Gamma,\Sigma A.B\vdash C:\mathrm{Type}_i \quad \Gamma,A,(B\downarrow 0)\vdash r:(C\downarrow \mathrm{pair}_{\Sigma A.B}(1,0))}{\Gamma\vdash \mathrm{split}(C,\mathbf{a},r):C\downarrow a}\textit{(SIGMA-ELIM)} \quad ;$$

$$\frac{A_1\leqslant A_2 \quad B_1\leqslant B_2}{\Sigma A_1.B_1\leqslant \Sigma A_2.B_2}\textit{(SIGMA-COMPAT)} \quad ; \quad \frac{\Sigma A_1.B_1\leqslant\Sigma A_2.B_2 \quad a_1\leqslant a_2 \quad b_1\leqslant b_2}{\mathrm{pair}_{\Sigma A_1.B_1}(a_1,b_1)\leqslant \mathrm{pair}_{\Sigma A_2.B_2}(a_2,b_2)}\textit{(PAIR-COMPAT)} \quad ;$$

$$\frac{a_1\leqslant a_2 \quad C_1\leqslant C_2 \quad r_1\leqslant r_2}{\mathrm{split}(C_1,a_1,r_1)\leqslant \mathrm{split}(C_2,a_2,r_2)}\textit{(SPLIT-COMPAT)} \quad .$$

По аналогии с зависимыми произведениями, зависимые суммы (Σ-типы, dependent sum) могут рассматриваться как обобщение упорядоченной пары типов A и B (упорядоченная пара, как правило, обозначается как $A\times B$) в случае, когда тип B зависит от значения типа A. Примером зависимой суммы может послужить пара из натурального числа n и вектора натуральных чисел длины n: $\Sigma\{n:\mathbb{N}\}.\mathrm{Vector}(\mathbb{N},n)$. Заметим, что разница между этим примером и примером зависимого произведения из предыдущего подраздела состоит в том, что элементом зависимой суммы явля-

ется пара значений $[n,v]$, тогда как элементом зависимого произведения является λ-абстракция, результатом применения которой к аргументу n является некоторый элемент v. Элементы зависимой суммы вводятся с помощью терма зависимой пары (SIGMA-INTRO) и удаляются с использованием терма разделения (SIGMA-ELIM).

***Замечание* 1.** *Перечислим введенные на настоящее время термы:* $0,\ldots,n,\mathrm{Type}_i,\Pi A.B,\lambda A.b,x\cdot a,\Sigma A.B,\mathrm{pair}_{\Sigma A.B}(a,b),\mathrm{split}(C,a,r)$.

Отметим также, что для всех приведённых правил вывода, каждый терм встречается в левой части отношения типизации в точности один раз.

2.1.7. Свойства исчисления

***Лемма* 2.1.** *В правой части отношения типизации всегда находится тип, т.е., в формальной записи:*

$$\frac{\Gamma\vdash x : A}{\Gamma\vdash A : \mathrm{Type}_i} \quad .$$

Доказательство. Доказательство проходит с использованием индукции по выводу посылки $\Gamma\vdash x : A$ и по структуре терма x.

Если x — это переменная, то посылка утверждения леммы может быть получена только с использованием правила CTX-DEREF на некотором подконтексте Γ, т.к. только в этом правиле переменная находится в левой стороне отношения типизации. Таким образом, для некоторого Δ, являющегося подконтекстом Γ, справедливо $\Delta, A\ \ \mathrm{wf}$. Это заключение может быть получено только с использованием правила CTX-ADD, следовательно, по посылкам этого правила, $\Delta\vdash A : \mathrm{Type}_i$. Поскольку Δ — подконтекст Γ, $\Gamma\vdash A\uparrow^x : \mathrm{Type}_i$.

Пусть x — не переменная. Рассмотрим структуру терма x и последнее применённое правило в выводе $\Gamma\vdash x : A$ с учётом предыдущего замечания, и покажем, что из посылок этого правила выводимо утверждение леммы:

1. Type_i — по TYPE-TYPING;
2. $\Pi A.B$ — по PI-FORM, TYPE-TYPING;

3. $\lambda A.b$ — по посылке в PI-INTRO;
4. $x \cdot a$ — согласно посылке в PI-ELIM, $\Gamma \vdash x : \Pi A.B$, следовательно, по предыдущему, $\Pi A.B$: Type и, согласно PI-FORM, $\Gamma, A \vdash B : \text{Type}_j$, следовательно, по свойству подстановки, утверждение леммы выполняется;
5. $\Sigma A.B$ — по SIGMA-FORM, TYPE-TYPING;
6. $\text{pair}_{\Sigma A.B}(a,b)$ — по SIGMA-INTRO;
7. $\text{split}(C,a,r)$ — по SIGMA-ELIM.

Были рассмотрены все термы, определённые на настоящее время, поэтому утверждение леммы справедливо. □

Далее, при введении новых правил вывода, в заключении которых содержится отношение типизации, будет необходимо сохранить единственность типизации для термов и продемонстрировать сохранение утверждения леммы 2.1.

2.1.8. Правила редукции

Правила редукции предназначены для преобразования термов. С помощью последовательного применения таких правил в λ-исчислении представляется понятие процесса вычисления. Основным свойством, требуемым от правил редукции, является сохранение типизации, т.е. при применении правил термы должны переходить в термы того же типа. О корректности с этих позиций правил, приведённых в настоящем разделе упоминается, например, в [83].

***Определение* 2.9.** *В рамках базового исчисления рассматриваются следующие правила редукции:*

$$(\lambda A.t)\cdot x \longrightarrow t\downarrow x \;\; \textit{(BETA-REDUCTION)} \quad ; \quad \text{split}(C,\text{pair}_{\Sigma A.B}(a,b),r) \longrightarrow r\downarrow a\downarrow b \;\; \textit{(SIGMA-REDUCTION)} \quad .$$

Правило BETA-REDUCTION позволяет вычислять результат операции приложения путём подстановки значения аргумента (правой части) в функцию (левую часть). Правило SIGMA-REDUCTION позволяет вычислять значение терма разделения на зависимых парах с помощью явного разделения таких пар.

Введём следующие сокращения, которые могут потребоваться далее:

1. $\mathrm{fst}x \equiv \mathrm{split}(A,x,1)$;
2. $\mathrm{snd}x \equiv \mathrm{split}(B \downarrow \mathrm{fst}0,x,0)$.

***Лемма* 2.2.** *Для* fst*,* snd *верны следующие соотношения:*

1. $\Gamma \vdash x : \Sigma A.B \Rightarrow \Gamma \vdash \mathrm{fst}(x) : A$;

2. $\Gamma \vdash x : \Sigma A.B \Rightarrow \Gamma \vdash \mathrm{snd}(x) : B \downarrow \mathrm{fst}(x)$;

3. $\mathrm{fst}(\mathrm{pair}(a,b)) \longrightarrow^* a$;

4. $\mathrm{snd}(\mathrm{pair}(a,b)) \longrightarrow^* b$.

Доказательство. 1. Первое суждение получается путём применения правила SIGMA-ELIM с посылкой $C \equiv A\uparrow$.

2. Второе суждение получается путём применения того же правила со следующими посылками:

   $C \equiv (B \downarrow \mathrm{fst}0)\uparrow$

   $r \equiv 0$ и, по предыдущему, $\Gamma, A, (B \downarrow 0) \vdash r : B \downarrow 1$.

3. Справедливость третьего и четвёртого суждений может быть показана путём применения правила редукции SIGMA-REDUCTION один и два раза, соответственно.

□

В завершение описания основных положений модели λ-исчисления с зависимыми типами для удобства читателя представим в виде таблицы основные положения соответствия Карри-Говарда в терминах представленной разновидности исчисления (Таблица 2.1). Напомним, что Исчисление Конструкций и основанные на нем разновидности λ-исчисления интерпретируются в рамках интуиционистской логики.

2.2. Исчисление с типами эквивалентности

Различные разновидности формальной теории типов Мартин-Лёфа и исчисления конструкций широко используются как для формализации математических утверждений и доказательств, так и для описания формальной семантики программ и языков

Таблица 2.1: Соответствие Карри-Говарда

Формальная логика	λ-исчисление с зависимыми типами
Высказывание A	Тип (A : Type)
Доказательство высказывания A	Элемент типа A ($a : A$)
Импликация $A \to B$ (из A следует B)	Зависимое произведение $\Pi A.B$
Конъюнкция $A \wedge B$ (A и B)	Зависимая сумма $\Sigma A.B$
Дизъюнкция $A \vee B$ (A или B)	Тип дизъюнкции (см. 3.1.1) $\Sigma\{w : \#2\}.\mathbf{case}(2, w, \mathrm{Type}, \mathsf{c}_0 \equiv A, \mathsf{c}_1 \equiv B).$
Квантор всеобщности $\forall a, B(a)$ (для любого a верно $B(a)$)	Зависимое произведение $\Pi\{a : A\}.B(a)$
Квантор существования $\exists a, B(a)$ (для некоторого a верно $B(a)$	Зависимая сумма $\Sigma\{a : A\}.B(a)$

программирования. В последнее время различными группами исследователей были получены новые результаты исследований, объединение которых в составе единой формальной системы может предоставить новые возможности для эффективного описания формальных моделей программ. Из числа таких результатов можно выделить, в первую очередь, гомотопическую теорию типов, краткая информация о которой приведена далее.

Интерпретация теории типов с точки зрения алгебраической топологии и высшей теории категорий позволила создать новую разновидность теории типов, получившей название гомотопической теории типов [121]. Новые элементы включают в себя, прежде всего, аксиому унивалентности, из которой следует свойство функциональной экстензиональности. Согласно этому свойству, две функции идентичны в том и только в том случае, когда они дают идентичные результаты на одинаковых аргументах. Аксиома унивалентности позволяет рассматривать идентичность на типах как отношение конвертируемости. В соответствие с этим отношением:

- если можно задать функцию конвертации между двумя типами, которая удовлетворяет определённым свойствам, то это означает, что типы идентичны;

- можно определить понятие высших индуктивных типов, которые позволяют задавать типы не только набором генераторов элементов, но и набором генераторов путей (эквивалентностей) между такими элементами (например, [64]).

В настоящем разделе представлена разновидность λ-исчисления с зависимыми типами, которая поддерживает нетривиальную редукцию типов идентичности. Отметим, что на настоящее время автором не получено результатов по метатеоретическим свойствам предложенной разновидности исчисления. Получение таких свойств затруднено в связи с большим количеством базовых термов и правил редукции. Это обстоятельство, с одной стороны, несомненно, является недостатком представленных результатов, а с другой — определяет одно из направлений исследований в перспективе. Следует, однако, заметить, что из известных на время написания настоящей работы результатов [14, 80] ни один не предоставляет аналогичных возможностей с точки зрения редукции нетривиальных типов идентичности.

Основным теоретическим результатом, представленным в настоящем разделе, является новая разновидность λ-исчисления с зависимыми типами, поддерживающая аксиому унивалентности и нетривиальные редукции для типов идентичности (в рамках работы не исследовался вопрос нормализации исчисления с вновь введёнными правилами редукции).

2.2.1. Правила идентичности

В числе составляющих частей определения формальной системы, как правило, рассматривается отношение равенства термов (в настоящей разновидности исчисления такое понятие заменяется несимметричным отношением совместимости термов). Однако такое отношение является внешним для формальной системы, оно используется при проверке корректности типов, но не может использоваться при построении утверждений. Понятие типов идентичности было введено в теорию типов Мартин-Лёфом в [87] как средство описания утверждений о равенстве объектов в терминах теории типов.

***Определение* 2.10.** *Типы идентичности формируются с использованием следующих правил вывода:*

$$\frac{\Gamma \vdash A : \mathrm{Type}_i \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \mathrm{Type}_j \quad [j \leqslant i]} \textit{(EQUIV-FORM)} \quad ; \quad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{refl}_A a : a =_A a} \textit{(EQUIV-INTRO)} \quad ;$$

$$\frac{\Gamma \vdash a,b : A \quad \Gamma \vdash \gamma : a =_A b \quad \Gamma, A, a =_A 0 \vdash C : \mathrm{Type}_i \quad \Gamma \vdash x : C \downarrow_1 a \downarrow \mathrm{refl}_A a}{\Gamma \vdash J(A,a,b,\mathbf{C},\gamma,x) : C \downarrow_1 b \downarrow \gamma} \textit{(EQUIV-ELIM)} \quad .$$

Для любых двух элементов a,b типа A определён (возможно, пустой) тип идентичности $a =_A b$, соответствующий утверждению о равенстве a и b. В классической формулировке типов идентичности, в явном виде может быть построен только элемент рефлексивности, $\mathrm{refl}_A a$ доказывающий равенство элемента некоторого типа самому себе. Для удаления типов идентичности используется терм J, который в представленном в правиле EQUIV-ELIM виде был предложен в [101]. Применение этого терма может быть интерпретировано следующим образом: «если верно, что a и b равны, то некоторые вхождения a в x, указанные в C, могут быть заменены на b». Отметим, при этом, что с позиций настоящей работы в качестве удаляемого правилом EQUIV-ELIM терма следует рассматривать именно подтерм C, поскольку именно структура C определяет возможность применения тех или иных правил редукции, которые будут введены далее.

Для лучшего понимания специфики типов идентичности, которой обусловлено, в том числе, и расширение понятия о них в рамках гомотопической теории типов, следует упомянуть интензиональную и экстензиональную разновидность теории типов. В рамках интензиональной теории типов на структуру типов идентичности не налагается никаких ограничений. Влияние элементов типов идентичности на другие термы ограничивается применением терма удаления J. Однако в оригинальных лекциях [87] предлагалось ввести суждение, позволяющее получить отношение равенства термов из элемента типа идентичности, т.е. структура типов идентичности фиксировалась как тривиальная — либо истинное, либо ложное утверждение. Подобная интерпретация получила название экстензиональной теории типов. Такая разновидность теории типов, фактически, позволяет вводить новые способы редукции термов с помощью формул исчисления путём доказательства идентичности. Это свойство достаточно

удобно при работе с разновидностями λ-исчисления, построенными на основе такой теории типов. Однако как показано, например, в [96, 14.1, с. 90], экстензиональная интерпретация приводит к неразрешимости задачи проверки типов в исчислении.

Тем не менее, исследования, позволяющие частично использовать элементы экстензиональной теории типов, не утрачивая при этом свойство разрешимости задачи проверки типов, продолжались. В числе таких исследований следует отметить, в частности, неопубликованную работу [4]. Подход, изложенный в этой работе, заключается в интерпретации утверждения о равенстве элементов некоторого типа с позиций равенства всех термов удаления для таких элементов. В [5] в рамках этой интерпретации предлагаются правила редукции, аналогичные слабой форме представленных в настоящей работе PI-EQ-REDUCTION и SIGMA-EQ-REDUCTION. В то же время, предложенные автором правила допускают правило удаления равенства в представленной выше форме EQUIV-ELIM, тогда как правила в [5] не допускают дополнительное использование равенства в типе C — шаблоне подстановки.

Одним из последних результатов, предлагающим расширенную интерпретацию типов идентичности, является гомотопическая теория типов [8, 128]. Интерпретация гомотопической теории типов основана на абстрактной теории гомотопий. В числе основных положений этой теории с позиций настоящей работы следует отметить аксиому унивалентности, согласно которой тип идентичности между любыми двумя типами эквивалентен типу эквивалентности (преобразования, сохраняющего свойства) между такими типами. Руководствуясь этим положением, а также свойствами типов эквивалентности, представленными в [121, Chapter 2], добавим в определение разновидности λ-исчисления как формальной системы новое суждение «структуры эквивалентности» и использующие его термы введения и удаления элементов типов идентичности.

***Определение* 2.11.** *Для описания структуры эквивалентностей высших размерностей, дополнительно к рассматриваемым в определении 2.3 используется следующий новый вид суждений:*

Структура эквивалентности $f =_A g \Leftrightarrow X$.

При этом требуется выполнение следующего свойства:

$$\frac{\Gamma \vdash A : \mathrm{Type}_i \quad \Gamma \vdash f,g : A}{\Gamma \vdash X \downarrow f \downarrow g : \mathrm{Type}_j} \quad .$$

***Определение* 2.12.** *Отношение структуры эквивалентности используется при типизации с использованием следующих правил вывода:*

$$\frac{f =_A g \Leftrightarrow X \quad \Gamma \vdash x : X \quad \Gamma \vdash A : \mathrm{Type} \quad \Gamma \vdash f : A \quad \Gamma \vdash g : A}{\Gamma \vdash \mathrm{ua}_A(f,g,x) : f =_A g} \text{(UA-INTRO)} \quad ;$$

$$\frac{f =_A g \Leftrightarrow X \quad \Gamma \vdash \gamma : f =_A g}{\Gamma \vdash \mathrm{itoe}(\gamma) : X} \text{(UA-ELIM)} \quad .$$

Отношение совместимости на вновь определяемых термах задаётся поэлементно с сохранением порядка, аналогично случаю зависимых сумм.

***Лемма* 2.3.** *Утверждение леммы 2.1 выполняется для вновь вводимых в определениях 2.10, 2.12 термов.*

Доказательство. Рассмотрим структуру терма x и последнее применённое правило (см. доказательство леммы 2.1) для вновь вводимых термов:

1. $a =_A b$ в правиле EQUIV-FORM — по правилу TYPE-TYPING;
2. $\mathrm{refl}_A a$ в правиле EQUIV-INTRO — имеем $\Gamma \vdash a : A$, следовательно, по предположению индукции, $\Gamma \vdash A : \mathrm{Type}_i$, поэтому утверждение леммы верно по правилу EQUIV-FORM;
3. $J(A,a,b,C,\gamma,x)$ в правиле EQUIV-T — имеем $\Gamma \vdash b : A$ и $\Gamma, A \vdash C : \mathrm{Type}_i$, следовательно, утверждение леммы выполняется по свойству операции подстановки $\downarrow$;
4. $f =_A g$ в правиле UA-INTRO — посылок правила достаточно для применения EQUIV-FORM;
5. X в правиле UA-ELIM — по определению отношения структуры эквивалентности.

□

2.2.2. Редукция — базовые правила

***Определение* 2.13.** *Для типов эквивалентности вводятся следующие базовые правила редукции:*

$$J(A,a,a,X,\mathrm{refl}_A a,x) \longrightarrow x \quad \textit{(J-REFL-REDUCTION)} \qquad ;$$

$$\mathrm{itoe}(\mathrm{ua}(f,g,x)) \longrightarrow x \quad \textit{(ITOE-UA-REDUCTION)} \qquad ;$$

$$\mathrm{ua}(f,g,\mathrm{itoe}(\gamma)) \longrightarrow \gamma \quad \textit{(UA-ITOE-REDUCTION)} \qquad ;$$

$$(X\downarrow_1 a\uparrow^2) \leqslant X\downarrow_1 1 \leqslant (X\downarrow_1 a\uparrow^2) \Rightarrow J(A,a,b,X,\gamma,x) \longrightarrow x \quad \textit{(J-INDEP-REDUCTION)} \qquad .$$

В базовых правилах редукции рассматриваются тривиальные случаи: редукция для рефлексивных элементов типов идентичности и взаимно-обратная связь между ua и itoe. Правило J-INDEP-REDUCTION показывает, что, если тип X, задающий схему конвертации по эквивалентности, не зависит от переменных $0,1$, т.е. не содержит мест их подстановки, то результат операции J тривиален.

Введём следующие сокращения.

1. Для элемента типа идентичности $\gamma : a =_A b$ обратный к нему элемент равен $\gamma^{-1} \equiv J(A,a,b,a =_A 1,\gamma,\mathrm{refl}_A a)$.
2. Для двух элементов типов идентичности $\gamma : a =_A b, \delta : b =_A c$ их композиция равна $\gamma \circ \delta \equiv J(A,b,c,a =_A 1,\delta,\gamma)$.

Корректность сокращений по отношению к типизации очевидна из правила EQUIV-T.

В последующих подразделах рассматриваются вопросы описания типов идентичности в терминах структуры эквивалентности и соответствующих им правил редукции на определённых ранее типах.

2.2.3. Эквивалентность — Type

Как было отмечено ранее, эквивалентность на типах определяется исходя из аксиомы унивалентности. Согласно [121, Chapter 4], допускается несколько равносильных определений эквивалентности на типах. В настоящей работе используется определение, основанное на двусторонней обратимости некоторого отображения между типами.

Поскольку предложенные автором правила имеют достаточно сложную структуру, для более понятного представления система записи термов была дополнена тремя перечисленными далее обозначениями.

$\mathbf{let}\, c_i : T_i, c_i \equiv t_i \,\mathbf{in}\, t$

— согласно такой записи, вместо c_i, имеющего тип T_i, в терм t подставляется терм t_i. Запись допускает введение несколько последовательных промежуточных переменных.

$(x : T)$

— используется в левой части зависимых сумм, λ-абстракций и произведений; означает, что именем x в правой части обозначается переменная 0.

$(a,b) \mapsto t$

— означает, что терм t рассматривается в контексте, расширенном одной или большим числом переменных. Именами a, b и т.п. обозначаются такие переменные (в примере — b означает 0, a — 1).

Кроме того, в дальнейшем будут опущены операции с уровнем терма. Интерпретация таких расширений записи приведена далее, в подразделе 3.2.3.

Введем следующие сокращения [121, 4.2 – 4.3]:

1. $\mathrm{linv}(f : \Pi A.B) \equiv \Sigma(\Pi A.B).\Pi A.1 \cdot (f \cdot 0) =_A 0$;
2. $\mathrm{rinv}(f : \Pi A.B) \equiv \Sigma(\Pi A.B).\Pi B.f \cdot (1 \cdot 0) =_B 0$;
3. $\mathrm{biinv}(f : \Pi A.B) \equiv \Sigma \mathrm{linv}(f).\mathrm{rinv}(f)$.

Положим также $id_A : \Pi A.A \equiv \lambda A.0$.

***Определение* 2.14.** *Для типов* Type_i *эквивалентности высших размерностей рассматриваются следующим образом:*

$$A =_{\mathrm{Type}} B \Leftrightarrow \Sigma(\Pi A.B).\mathrm{biinv}(0) \quad \textit{(TYPE-EQ)} \qquad ;$$

$$\mathrm{refl}_{\mathrm{Type}} A \longrightarrow \mathrm{ua}_{\mathrm{Type}}(A, A, \mathrm{pair}(id_A, \mathrm{pair}_{\mathrm{biinv}(0)}[\mathrm{pair}(id_A, \lambda A.\mathrm{refl} 0), \mathrm{pair}(id_A, \lambda A.\mathrm{refl} 0)]))$$

$$\textit{(TYPE-REFL-REDUCTION)} \qquad ;$$

$$J(\text{Type}, a, b, 1, \gamma, x) \longrightarrow \text{fst}(\text{itoe}(\gamma)) \cdot x \quad \textit{(TYPE-EQ-REDUCTION)} \qquad .$$

***Лемма* 2.4.** *Определение 2.14 сохраняет типизацию, а именно:*

1. *выполняется условие отношения структуры эквивалентности*

$$\Gamma \vdash A, B : \text{Type} \Rightarrow \Gamma \vdash \Sigma(\Pi A.B).\text{biinv}(0) : \text{Type} \qquad ;$$

2. *правило* TYPE-EQ-REDUCTION *не нарушает типизацию*

$$\Gamma \vdash J(\text{Type}, a, b, 0, \gamma, x) : b \Rightarrow \Gamma \vdash \text{fst}(\text{itoe}(\gamma)) \cdot x : b \qquad ;$$

3. *сохраняется правило редукции* T-REFL-REDUCTION, *т.е. при исключении этого правила справедливо*

$$J(\text{Type}, A, A, 1, \text{refl}_{\text{Type}} A, x) \longrightarrow^* x \qquad .$$

Доказательство. Для первого утверждения леммы приведём дерево вывода типа:

$$\text{Sigma-Form}\dfrac{\text{Pi-Form}\dfrac{\Delta \vdash A : \text{Type} \qquad \Delta, A : \text{Type} \vdash B : \text{Type}}{\Delta \vdash \Pi A.B : \text{Type}} \qquad \text{Sigma-Form}\dfrac{\Delta, f : \Pi A.B \vdash \text{linv}(f) : \text{Type}(1) \qquad \Delta, f : \Pi A.B, \text{Type} \vdash \text{rinv}(f) : \text{Type}(1)}{\Delta, f : \Pi A.B \vdash \text{biinv}(f) \equiv \Sigma \text{linv}(f).\text{rinv}(f) : \text{Type}}}{\Delta \equiv \Gamma, A : \text{Type}, B : \text{Type} \vdash \Sigma(f : \Pi A.B).\text{biinv(f)} : \text{Type}}$$

$$\text{(1)Sigma-Form}\dfrac{\dfrac{\vdots}{\Delta, f : \Pi A.B, g : \Pi A.B \vdash \Pi(x : A).g \cdot (f \cdot x) =_A x : \text{Type}} \qquad \Delta, f : \Pi A.B, g : \Pi A.B \vdash \Pi(y : B).f \cdot (g \cdot y) =_B y : \text{Type} \qquad \Delta, \Pi A.B \vdash \Pi A.B : \text{Type}}{\Delta, f : \Pi A.B \vdash \text{linv(f)}, \text{rinv(f)} : \text{Type}} \qquad .$$

Второе утверждение может быть получено применением правил редукции SIGMA-REDUCTION, BETA-REDUCTION и ITOE-UA-REDUCTION с использованием вывода из первого утверждения.

Третье утверждение справедливо по TYPE-REFL-REDUCTION, TYPE-EQ-REDUCTION и правил редукции согласно второму утверждению. □

В дальнейшем, в случае, если в определении используется конструкция **let** с достаточным количеством дополнительных аннотаций типов, доказательства аналогичных лемм сохранения типизации будут приведены в сокращённом виде.

2.2.4. Эквивалентность на зависимых произведениях

***Определение* 2.15.** *Для типов* $\Pi A.B$ *эквивалентности высших размерностей задаются с использованием следующих правил:*

$$f =_{\Pi A.B} g \Leftrightarrow \Pi(a : A).(f \cdot a) =_B (g \cdot a) \quad \textit{(PI-EQ)} \qquad ;$$

$$\mathrm{refl}_{\Pi A.B} f \longrightarrow \mathrm{ua}(f, f, \lambda(a : A).\mathrm{refl}_{B\downarrow a}(f \cdot a)) \quad \textit{(PI-REFL-REDUCTION)} \qquad ;$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{J}\big(\Pi \mathrm{A.B, f, g}, \big(\mathrm{h}, \eta\big) \mapsto \mathrm{h} \cdot \mathrm{D}, \gamma, \mathrm{x}\big) \longrightarrow \mathbf{let} \qquad \text{(APP-EQ-REDUCTION)}} \\
\mathrm{D_1} & \equiv & \mathrm{D} \downarrow_1 \mathrm{g} \downarrow \gamma \\
\mathrm{y} & : & \mathrm{f} \cdot \mathrm{D_1} \\
\mathrm{y} & \equiv & \mathrm{J}\big(\Pi \mathrm{A.B, f, g, f} \cdot \mathrm{D}, \gamma, \mathrm{x}\big) \\
 & \mathbf{in} & \mathrm{J}\big(\mathrm{B} \downarrow \mathrm{D_{1,}}\ \mathrm{f} \cdot \mathrm{D_{1,}}\ \mathrm{g} \cdot \mathrm{D_{1,}}\ \big(\mathrm{h}, \eta\big) \mapsto \mathrm{h}, \gamma, \mathrm{y}\big) \qquad ;
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{J}\big(\mathrm{A, a, b}, \Pi \mathrm{E.F}, \gamma, \mathrm{x}\big) \longrightarrow \mathbf{let} \qquad \text{(PI-EQ-REDUCTION)}} \\
\mathrm{E_0, E_1} & : & \mathbf{Type} \\
\mathrm{E_0} & \equiv & \mathrm{E} \downarrow_1 \mathrm{a} \downarrow \mathbf{refl}\ \mathrm{a} \\
\mathrm{E_1} & \equiv & \mathrm{E} \downarrow_1 \mathrm{b} \downarrow \gamma \\
\varepsilon & : & \mathrm{E_1} =_{\mathbf{Type}} \mathrm{E0} \\
\varepsilon & \equiv & \mathrm{J}\big(\mathrm{A, a, b, E} =_{\mathbf{Type}} \mathrm{E_0}, \gamma, \mathbf{refl}\ \mathrm{E_0}\big) \\
\mathrm{e} & : & \mathrm{E0} \\
\mathrm{e} & \equiv & \lambda \mathrm{E_1}\ .\ \mathrm{J}\big(\mathbf{Type}, \mathrm{E_1}, \mathrm{E_0}, 1, \varepsilon, 0\big)
\end{array}$$

$$
\begin{aligned}
e' &\equiv \lambda(e : E_0).(h : A).(\eta : a =_A h).\, J(A, a, h, E, \eta, e) \\
F' &\equiv (h, \eta) \mapsto F \downarrow_2 h \downarrow_1 \eta \downarrow (e' \cdot (e \cdot 2) \cdot 1 \cdot 0) \\
&\ \mathbf{in}\ \ \lambda E_1 .\, J(A, a, b, F', \gamma, x \cdot (e \cdot 0)) \quad .
\end{aligned}
$$

***Лемма* 2.5.** *Определение 2.15 сохраняет типизацию, а именно:*

APP-EQ-REDUCTION $J(\Pi A.B, f, g, (h,\eta) \mapsto h \cdot D, \gamma, x) : Y \longrightarrow y : Y \quad ;$

PI-EQ-REDUCTION $J(A, a, b, \Pi E.F, \gamma, x) : Y \longrightarrow y : Y \quad .$

Доказательство.

APP-EQ-REDUCTION Исходный и результирующий термы имеют тип $g \cdot D_1$. Имеющиеся в определении аннотации типов дополним следующими наблюдениями. Т.к. исходный терм корректно типизирован, следовательно, для любого $t : \Pi A.B$ и $\tau : f = t$, $t \cdot (D \downarrow_1 t \downarrow \tau) :$ Type. Для любых таких t, τ справедливо $(D \downarrow_1 t \downarrow \tau) : A$. Остальные выводы в рамках доказательства могут быть тривиально получены из вывода $J(\Pi A.B, f, g, (h,\eta) \mapsto h \cdot D, \gamma, x) : Y$.

PI-EQ-REDUCTION Укажем вывод типов для последнего терма. Отметим общую схему преобразования, задаваемого этим правилом. Аргумент типа E_1 преобразуется в тип E_0. Затем производится вычисление значения исходной функции с таким аргументом $(x \cdot (e \cdot 0))$. Тип такого приложения — $F \downarrow_2 a \downarrow_1 \mathrm{refl}\, a \downarrow (e \cdot 0)$. При этом F корректно типизируем в контексте $\Gamma, A, a =_A 0, E \downarrow_1 1 \downarrow 0$, поэтому для сохранения типизации в этот терм подставляется терм e'. □

2.2.5. Эквивалентность на зависимых суммах

***Определение* 2.16.** *Для типов $\Sigma A.B$ эквивалентности высших размерностей задаются с использованием следующих правил:*

$x =_{\Sigma A.B} y \quad \Leftrightarrow$ (**SIGMA**-EQ)

$$\Sigma\ (\mathbf{fst}\ x =_A \mathbf{fst}\ y)\ .\ (J(A, \mathbf{fst}\ x, \mathbf{fst}\ y, B \downarrow 1, 0, \mathbf{snd}\ x) =B \downarrow \mathbf{fst}\ y\ \mathbf{snd}\ y) \quad ;$$

$\mathbf{refl}_{\Sigma A.B}\ x \longrightarrow$ (**SIGMA**-REFL-REDUCTION)

$$\mathbf{ua}(x, x, \mathbf{pair}(\mathbf{refl}\ (\mathbf{fst}\ x), \mathbf{refl}\ (\mathbf{snd}\ x))) \quad ;$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{J}\big(\Sigma \mathrm{A.B,\ a,\ b,\ } \mathbf{split}\big(\mathbf{Type},\ 1,\ \mathrm{r}\big),\ \gamma,\ \mathrm{x}\big) \longrightarrow \mathbf{let} \qquad \text{(SPLIT-EQ-REDUCTION)}} \\
\mathrm{r}' & \equiv & \big(\mathrm{h},\ \eta\big) \mapsto \mathrm{r} \downarrow_1 \mathrm{h} \downarrow \mathrm{J}\big(\mathrm{A},\ \mathbf{fst}\ \mathrm{a},\ \mathrm{h},\ \big(\mathrm{m},\ \mu\big) \mapsto \mathrm{B} \downarrow \mathrm{m},\ \eta,\ \mathbf{snd}\ \mathrm{a}\big) \\
\mathrm{x}' & : & \mathrm{r} \downarrow_1 \mathbf{fst}\ \mathrm{b} \downarrow \mathrm{J}\big(\mathrm{A},\ \mathbf{fst}\ \mathrm{a},\ \mathbf{fst}\ \mathrm{b},\ \big(\mathrm{m},\ \mu\big) \mapsto \mathrm{B} \downarrow \mathrm{m},\ \gamma,\ \mathbf{snd}\ \mathrm{a}\big) \\
\mathrm{x}' & \equiv & \mathrm{J}\big(\mathrm{A},\ \mathbf{fst}\ \mathrm{a},\ \mathbf{fst}\ \mathrm{b},\ \mathrm{r}',\ \mathbf{fst}\ \big(\mathbf{itoe}\ \gamma\big),\ \mathrm{x}\big) \\
\mathrm{a_2}' & \equiv & \mathrm{J}\big(\mathrm{A},\ \mathbf{fst}\ \mathrm{a},\ \mathbf{fst}\ \mathrm{b},\ \mathrm{B} \downarrow 1,\ \mathbf{fst}\ \big(\mathbf{itoe}\ \gamma\big),\ \mathbf{snd}\ \mathrm{a}\big) \\
 & \mathbf{in} & \mathrm{J}\big(\mathrm{B} \downarrow \mathbf{fst}\ \mathrm{b},\ \mathrm{a_2}',\ \mathbf{snd}\ \mathrm{b},\ \mathrm{r} \downarrow \mathbf{fst}\ \mathrm{b} \downarrow 1,\ \mathbf{snd}\ \big(\mathbf{itoe}\ \gamma\big),\ \mathrm{x}'\big) \qquad ;
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{J}\big(\mathrm{A,\ a,\ b,\ } \Sigma \mathrm{E.F},\ \gamma,\ \mathrm{x}\big) \longrightarrow \mathbf{let} \qquad \text{(SIGMA-EQ-REDUCTION)}} \\
\mathrm{E_1} & \equiv & \mathrm{E} \downarrow_1 \mathrm{a} \downarrow \mathbf{refl}\ \mathrm{a} \\
\mathrm{y_1} & \equiv & \lambda\ \mathrm{A}\ .\ \mathrm{a} =_{\mathrm{A}} 0\ .\ \mathrm{E_1}\ .\ \mathrm{J}\big(\mathrm{A},\ \mathrm{a},\ 2,\ \mathrm{E},\ 1,\ 0\big) \\
\mathrm{y_2} & \equiv & \lambda \mathrm{E_1}.\ \big(\mathrm{F} \downarrow_2 \mathrm{a} \downarrow_1 \mathbf{refl}\ \mathrm{a} \downarrow 0\big)\ .\ \mathrm{J}\big(\mathrm{A},\ \mathrm{a},\ \mathrm{b},\ \mathrm{F} \downarrow \big(\mathrm{y_1} \cdot 1 \cdot 0 \cdot 3\big),\ \gamma,\ 0\big) \\
 & \mathbf{in} & \mathbf{split}\ \big(\Sigma \mathrm{E.F} \downarrow_1 \mathrm{b} \downarrow \gamma,\ \mathrm{x},\ \mathbf{pair}\big(\mathrm{y_1} \cdot 1,\ \mathrm{y_2} \cdot 1 \cdot 0\big)\big) \qquad .
\end{array}
$$

***Лемма* 2.6.** *Определение 2.16 сохраняет типизацию, то есть:*

SPLIT-EQ-REDUCTION $J(\Sigma A.B,a,b,\mathrm{split}(\mathrm{Type},1,r),\gamma,x) : Y \longrightarrow y : Y \qquad ;$

SIGMA-EQ-REDUCTION $J(A,a,b,\Sigma E.F,\gamma,x) : Y \longrightarrow y : Y \qquad .$

Доказательство. В случае, рассматриваемом в правиле SPLIT-EQ-REDUCTION, перенос вдоль пути γ с шаблоном split(Type, h, r) разделяется на две части: сначала вдоль этого пути переносится первая компонента split(Type,a,r) с коррекцией типа второй компоненты, после чего перенос осуществляется для второй компоненты.

Для правила SIGMA-EQ-REDUCTION компоненты x разделяются и пересобираются в пару за два шага, содержание которых аналогично предыдущему случаю. □

2.2.6. Эквивалентность — $a =_A b$

Эквивалентности высших размерностей — то есть, эквивалентности над типами идентичности, описываются в терминах структуры типа эквивалентности типа-носителя A. Для того, чтобы избежать кольцевой зависимости, правило для удаления типа идентичности в определении 2.17 разделено на два. Первое правило EQ-EQ-CONST-REDUCTION определяется для случая, когда тип-носитель A не зависит от

переменных 0 и 1, т.е. не меняется при удалении. Такое правило адекватно описывает преобразования типов идентичности в рамках одного носителя — например, получение обратного типа идентичности и композицию. Затем с использованием этого правила вводится общее правило EQ-EQ-REDUCTION. Определение последнего правила достаточно сложно, поскольку описать соответствие I_1 и I_2 для разных носителей в общем случае в рамках рассматриваемой теории не представляется возможным без введения дополнительных определений сравнимой сложности.

***Определение* 2.17.** *Для типов $a =_A b$ при условии $a =_A b \Leftrightarrow I$ эквивалентности высших размерностей задаются с использованием следующих правил вывода:*

$$\alpha =_{a=_A b} \beta \Leftrightarrow \textbf{itoe}\ \alpha =_{\mathrm{I}} \textbf{itoe}\ \beta \qquad \text{(ID-EQ)} \qquad ;$$

$$\textbf{refl}_{a=_A b}\alpha \longrightarrow \textbf{ua}\big(\alpha, \alpha, \textbf{refl}\ (\textbf{itoe}\ \alpha)\big) \qquad \text{(ID-REFL)} \qquad ;$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{J}\big(\mathrm{B}, \mathrm{u}, \mathrm{v}, \mathrm{a} =_{\mathrm{A}} \mathrm{b}, \xi, \mathrm{x}\big) \longrightarrow \textbf{let} \qquad \text{(EQ-EQ-CONST-REDUCTION)}} \\
\mathrm{A}_1 & \equiv & \mathrm{A} \downarrow_1 \mathrm{u} \downarrow \textbf{refl}\ \mathrm{u} \\
\mathrm{A}_2 & \equiv & \mathrm{A} \downarrow_1 \mathrm{v} \downarrow \xi \\
\text{при} & \multicolumn{2}{l}{\mathrm{A}_1 \leqslant \mathrm{A}_2 \leqslant \mathrm{A}_1} \\
\mathrm{A} & \equiv & \mathrm{A}_1 \\
1 =_{\mathrm{A}} 0 & \Leftrightarrow & \mathrm{I} \\
\mathrm{x}' & : & \mathrm{I} \downarrow_1 \big(\mathrm{a} \downarrow_1 \mathrm{u} \downarrow \textbf{refl}\ \mathrm{u}\big) \downarrow \big(\mathrm{b} \downarrow_1 \mathrm{u} \downarrow \textbf{refl}\ \mathrm{u}\big) \\
\mathrm{x}' & \equiv & \textbf{itoe}\ \mathrm{x} \\
\multicolumn{3}{l}{\quad \textbf{in ua}\Big(\mathrm{J}\big(\mathrm{B}, \mathrm{u}, \mathrm{v}, (\mathrm{h}, \eta) \mapsto \mathrm{I} \downarrow_1 (\mathrm{a} \downarrow_1 \mathrm{h} \downarrow \eta) \downarrow \mathrm{b} \downarrow_1 \mathrm{h} \downarrow \eta, \xi, \mathrm{x}\big)\Big) \qquad ;}
\end{array}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathrm{J}\big(\mathrm{B}, \mathrm{u}, \mathrm{v}, \mathrm{a} =_{\mathrm{A}} \mathrm{b}, \xi, \mathrm{x}\big) \longrightarrow \textbf{let} \qquad \text{(EQ-EQ-REDUCTION)}} \\
\mathrm{A}_1 & \equiv & \mathrm{A} \downarrow_1 \mathrm{u} \downarrow \textbf{refl}\ \mathrm{u} \\
\mathrm{A}_2 & \equiv & \mathrm{A} \downarrow_1 \mathrm{v} \downarrow \xi \\
1 =_{\mathrm{A}_2} 0 & \Leftrightarrow & \mathrm{I}_2 \\
\mathrm{a}_1 & \equiv & \mathrm{a} \downarrow_1 \mathrm{u} \downarrow \textbf{refl}\ \mathrm{u} : \mathrm{A}_1 \\
\mathrm{a}_2 & \equiv & \mathrm{a} \downarrow_1 \mathrm{v} \downarrow \xi : \mathrm{A}_2 \\
\mathrm{a}'_1 & \equiv & \mathrm{J}\big(\mathrm{B}, \mathrm{u}, \mathrm{v}, \mathrm{A}, \xi, \mathrm{a}_1\big) : \mathrm{A}_2 \\
\mathrm{b}_1 & \equiv & \mathrm{b} \downarrow_1 \mathrm{u} \downarrow \textbf{refl}\ \mathrm{u} : \mathrm{A}_1 \\
\mathrm{b}_2 & \equiv & \mathrm{b} \downarrow_1 \mathrm{v} \downarrow \xi : \mathrm{A}_2
\end{array}$$

$$
\begin{array}{lll}
b'_1 & \equiv & J\big(B,\ u,\ v,\ A,\ \xi,\ b_1\big)\ :\ A_2 \\
x_1 & \equiv & \mathbf{itoe}\big(\mathbf{refl}\ a'1\big)\ :\ I_2 \downarrow_1 a'_1 \downarrow a'_1 \\
x_2 & : & I_2 \downarrow_1 a'_1 \downarrow b'_1 \\
x_2 & \equiv & J\big(A_1,\ a_1,\ b_1,\ I_2 \downarrow_1 a'_1 \downarrow J\big(B,\ u,\ v,\ A,\ \xi,\ 1\big),\ x,\ x_1\big) \\
z & \equiv & \lambda\ A\ .\ B\ .\ u{=_B}\ 0.\ J\big(B,\ u,\ v,\ A,\ \xi \circ 0^{-1},\ 2 \downarrow_1 1 \downarrow 0\big) \\
x_3 & \equiv & J\big(B,\ u,\ v,\ I_2 \downarrow_1 z \cdot a \downarrow b'_{1,}\ \xi,\ x_2\big) \\
 & \mathbf{in} & J\big(B,\ u,\ v,\ I_2 \downarrow_1 a_2 \downarrow z \cdot b,\ \xi,\ x_3\big) \quad .
\end{array}
$$

***Замечание* 2.** *Дополнительное правило редукции термов вида*

$J\big(a =_A b,\ \alpha,\ \beta,\ J\big(A,\ a,\ b,\ C,\ 1,\ x\big),\ \xi,\ y\big)$

не вводится, т.к. редукция J не требует раскрытия элемента типа идентичности.

***Лемма* 2.7.** *Определение 2.17 сохраняет типизацию, то есть:*

EQ-EQ-CONST-REDUCTION $J(B,u,v,a =_A b,\xi,x) : Y \longrightarrow y : Y$ *при A не зависящем от переменных* 0,1*;*

EQ-EQ-REDUCTION $J(B,u,v,a =_A b,\xi,x) : Y \longrightarrow y : Y$

Доказательство. В случае EQ-EQ-CONST-REDUCTION, как было отмечено ранее, как для исходного $x : a_1 =_A b_1$, так и для конечного типа $y : a_2 =_A b_2 \equiv Y$ можно рассматривать одну и ту же структуру эквивалентности. Таким образом, преобразование типов идентичности сводится к преобразованию типов эквивалентности. Опущенные подтермы в ua — a_2 и b_2, как можно вывести из преобразования.

В правиле EQ-EQ-REDUCTION для преобразования используется следующий технический приём: в качестве исходного терма для преобразования используется не данный терм x, как в других правилах, а терм refla'_1. Терм a'_1 здесь — образ исходного a_1 в целевом типе A_2. Затем, используя удаление в тои числе и данного x, такой терм отображается в целевой тип $a_2 =_{A_2} b_2$. Указанных в определении аннотаций типов в целом достаточно для восстановления вывода при типизации. □

2.2.7. Редукция в термах удаления

В предыдущих подразделах были введены правила редукции для всех простых случаев удаления типов идентичности. Однако на практике структура типа-шаблона подстановки в терме удаления, на которую опираются введённые правила, нередко зависит от переменных, вводимых термом удаления.

Приведём несколько дополнительных определений из [84]. Форма определений отличается от оригинальной для того, чтобы учесть дополнительные термы, вводимые в настоящей работе.

Определение* 2.18.** *Терм A находится в* ***головной нормальной форме*, если он имеет вид* $A \equiv A_0(A_1, A_2, \ldots, A_k)$*, где* A_0 *— один из конструкторов, вводимых правилами -INTRO и -FORM.*

Определение* 2.19.** *Терм A находится в* ***нейтральной форме*, если он не находится в головной нормальной форме и к нему неприменимы правила редукции.*

Определение* 2.20.** *Терм* A_k *называется* ***ключевым редексом*, если он имеет ввид, вводимый правилами -ELIM, причём в удаляемой позиции* A_k *не стоит ключевой редекс.*

Следующее определение предложено автором.

Определение* 2.21.** *Конечная последовательность термов, вводимых правилами -ELIM* $A_1, \ldots, A_k$ *называется* ***главной последовательностью редексов *для терма A, если A имеет вид* $A = A_1 * A_2 * \ldots * A_k$*, где под операцией* $a * b$ *подразумевается операция подстановки b в удаляемую позицию a.*

Следующая лемма используется далее при доказательстве теоремы 2.1.

***Лемма* 2.8.** *Пусть* $A = \Sigma B.C$*,* $x, y : A$*,* $\gamma : x =_A y$*. Тогда для любого* $D : \Pi A.\mathit{Type}$ *и* $r : (\Pi B.C.D \cdot \mathrm{pair}_A(1,0))$*, можно построить:*

1. $\delta : D \cdot x =_{\mathrm{Type}} D \cdot y$*;*

2. $\mathrm{itoe}(\delta) \cdot \mathrm{split}(D \cdot 0, x, r \cdot 1 \cdot 0) =_{D \cdot y} \mathrm{split}(D \cdot 0, y, r \cdot 1 \cdot 0)$.

Доказательство. Первый терм строится тривиально:

$$\delta \equiv J(A,x,y,D\cdot x =_{\text{Type}} D\cdot 1,\gamma,\text{refl}(D\cdot x)) \qquad .$$

Для получения второго терма:

$$J(A,x,y,\text{itoe}(0)\cdot\text{split}(D\cdot 0,x,r\cdot 1\cdot\ 0) =_{D\cdot 1} \text{split}(D\cdot 0,1,r\cdot 1\cdot 0),\gamma,\text{refl}(\text{split}(D\cdot 0,x,r\cdot 1\cdot 0))) \qquad .$$

Таким образом, получены оба требуемых терма. □

Далее следует сформулированная автором теорема о возможности редукции термов удаления.

***Теорема* 2.1.** *К термам удаления типов идентичности в пустом контексте с типами-шаблонами подстановки, которые находятся в нейтральной форме, могут быть применены правила редукции.*

Доказательство. Рассмотрим главную последовательность редексов для типа-шаблона подстановки с позиций структуры ключевого редекса.

1. Ключевой редекс — приложение: $C \equiv x\cdot y$, $x : \Pi Y.\text{Type}$, $y : Y$. В этом случае редукция выполняется аналогично формулам в определении 2.15:

$$
\begin{aligned}
x &: c^1{}_1 * c^1{}_2 * c^1{}_3 \ldots c^1{}_k \\
c_k &\equiv 1 \cdot y: \\
m &\equiv \mathbf{itoe}(\gamma) \cdot y_0 : a \cdot y_0 = b \cdot y0 \\
n &\equiv J(B \downarrow y_0, a \cdot y_0, b \cdot y_0, c^1 1 * c^1 2 * \ldots * 1, m, x) \\
o &\equiv J(A, a, b, c_1 * c_2 * \ldots * (b \cdot 1), \gamma, x) \\
o &: c^2{}_1 * c^2{}_2 * c^2{}_3 * \ldots * c^2{}_k \qquad .
\end{aligned}
$$

2. Ключевой редекс — разделение зависимой суммы: $\text{split}(C,a,r)$, $C : \Pi(\Sigma A.B).\text{Type}$, $a : \Sigma A.B$, $r : \cdot\text{pair}(1,0)$. Редукция выполняется аналогично формулам в определении 2.16:

$$
\begin{aligned}
c_k &\equiv \mathbf{split}(C, 1, r) \\
m &\equiv J(A, a, b, c_1 * c_2 * c_3 \ldots J(A, a, 1, \mathbf{split}(\mathbf{Type}, 1, C), 0, \mathbf{split}(C^1, a, r^1)), \gamma, x)
\end{aligned}
$$

$$y \equiv J(A, a, b, \mathbf{split}(\mathbf{Type}, 1, C), \gamma, \mathbf{split}(C^1, a, r^1))$$

$$m : c^2{}_1 * c^2{}_2 * c^2{}_3 * \ldots * y$$

$$\delta : y =_{C^2} \mathbf{split}(C^2, a, r^2)$$

$$n \equiv J(C^2, y, \mathbf{split}(C^2, a, r^2), c^2{}_1 * c^2{}_2 * c^2{}_3 * \ldots * 1, \delta, m) .$$

Терм δ в предыдущих формулах получен по лемме 2.8.

3. Ключевой редекс — удаление типа идентичности по его структуре для первой переменной шаблона подстановки: $c_{k-1} * c_k \equiv c_{k-1} * itoe(1)$.

   Переменная 1 здесь имеет заданный тип $t = u$, поэтому при $t = u \Leftrightarrow X$, $\mathrm{itoe}(\gamma) : itoe(a) =_X itoe(b)$ и, соответственно, $\mathrm{itoe}(1) : X$, где X не зависит от переменных 0 и 1 шаблона подстановки. Пусть $c_{k-1} * c_k : E$. Тогда $c_{k-1} * c_k$ можно заменить на следующую формулу:

$$J(t =_X u, a, 1, E, 0, c_{k-1}{}^1 * \mathbf{itoe}(\mathbf{refl}\ a)) .$$

4. Ключевой редекс — удаление типа идентичности по его структуре для второй переменной шаблона подстановки: $c_{k-1} * c_k \equiv c_{k-1} * itoe(0)$

   В этом случае при $t = u \Leftrightarrow X$ терм $X \downarrow_2 a \downarrow_1 1 \downarrow 0$ имеет один и тот же внешний конструктор для любых значений переменных 1,0, удовлетворяющих правилам типизации исходного терма. Тип результата применения $c_{k-1} * c_k$ обозначим E. Тогда вместо $c_{k-1} * c_k$ может быть подставлена следующая формула:

$$J(A, a, 1, E, 0, c^1{}_{k-1} * \mathbf{itoe}(\mathbf{refl}\ a)) .$$

Случаи $C \equiv itoe(0)$ или $C \equiv itoe(1)$ для корректно типизированных термов рассматриваемого исчисления невозможны, т.к. введённые в настоящем разделе отношения структуры типа эквивалентности не допускают $a =_X b \Leftrightarrow \mathrm{Type}$ ни для каких X. Все необходимые случаи были рассмотрены, доказательство теоремы завершено. □

Следующая далее теорема призвана резюмировать результаты, представленные в настоящем разделе.

***Теорема* 2.2.** *В представленной разновидности исчисления возможна редукция любого корректно типизируемого терма удаления типа идентичности в другой терм, имеющий тип, совместимый с исходным.*

Доказательство. Все возможные случаи были рассмотрены в теореме 2.1, в леммах 2.4, 2.5, 2.6 и 2.7 и в определении 2.13. □

2.3. Выводы по второй главе

В первом разделе настоящей главы представлено расширенное исчисление конструкций с использованием индексов де Брёйна. Такая форма позволяет компактно задавать правила вывода исчисления без необходимости введения отношения эквивалентности на термах с точностью до переименования переменных. Однако, как продемонстрировано на предложенных автором дополнительных правилах редукции, такая форма осложняет определение нетривиальных правил. Поэтому для представления таких правил был использован синтаксис с именованными переменными. Правила в таком синтаксисе транслируются в индексы де Брёйна по схеме, приведённой в подразделе 3.2.3 настоящей главы.

Упомянутые выше предложенные автором новые правила редукции элементов типов идентичности показывают, что можно синтаксически описать вычислительное содержание аксиомы унивалентности. С другой стороны, нетривиальная структура новых правил свидетельствует о том, что доказательство свойств нормализации и разрешимости проверки типов для них является сложной задачей. Тем не менее, на настоящее время исследователями, разрабатывающими альтернативные подходы к описанию гомотопической теории типов, такие свойства также не были получены. Ряд доказанных автором лемм обеспечивает сохранение типизации при использовании новых правил редукции. Сформулированная и доказанная автором теорема 2.1 может рассматриваться как первый шаг к доказательству гипотезы Воеводского о вычислительном содержании аксиомы унивалентности.

Таким образом, в настоящей главе представлены результаты исследования существующих разновидностей λ-исчисления с зависимыми типами, а также разработки новой такой разновидности для использования в качестве промежуточного представления при описании формальной семантики различных языков программирования.

Глава 3

Разработка и реализация базового языка

В настоящей главе представлено решение второй задачи диссертационного исследования — реализации макетов языка программирования и формальной спецификации, а также программного средства, обеспечивающего проверку типов выражений такого языка на основе предложенной автором разновидности λ-исчисления с зависимыми типами.

Первый раздел содержит расширения используемой разновидности исчисления, которые предназначены для описания используемых в программах типов данных. В число таких расширений входят: конечные типы, свойства которых с точки зрения функционального программирования аналогичны свойствам конечных множеств; типы натуральных чисел; индуктивные и коиндуктивные типы.

Вопросы реализации подлежащего разработке макета программного средства описания формальных моделей программ и дедуктивной верификации их свойств в объёме, необходимом для дальнейшего изложения рассматриваются во втором разделе. Представлено описание синтаксиса макета языка, схема трансляции выражений макета языка в объекты, соответствующие термам исчисления, схема проверки и вывода типов выражений для макета языка. Термы в макете языка являются промежуточным представлением для формальной верификации программ, написанных с использованием нескольких языков программирования.

3.1. Исчисление с индуктивными типами

В предыдущем разделе в состав математической модели базового языка не были введены типы, с использованием которых могут быть представлены те или значения. К таким типам могут быть отнесены: тривиальный тип, состоящий из одного

элемента; тип булевских значений; тип натуральных чисел; индуктивные типы данных. В настоящем разделе вводятся стандартные определения конечных типов и натуральных чисел, используемые, в частности, в программных средствах Coq, Agda и Idris. После этого вводится новый, предложенный автором механизм описания высших индуктивных типов гомотопической теории типа в терминах предложенной разновидности исчисления с дополнительными правилами вывода. Как будет показано в последующих подразделах, этих трёх аксиоматически вводимых типов достаточно для описания нетривиальных индуктивных и коиндуктивных типов.

3.1.1. Конечные типы

***Определение* 3.1.** *Конечные типы формируются с использованием следующих правил вывода:*

$$\frac{\Gamma \mathrm{wf}}{\#i : \mathrm{Type}}\ \textit{(FIN-FORM)} \quad ; \quad \frac{\Gamma \mathrm{wf} \quad i<j}{j\#i : \#i}\ \textit{(FIN-INTRO)} \quad ;$$

$$\frac{\Gamma \vdash f : \#k \quad \Gamma, \#k \vdash C : \mathrm{Type} \quad \Gamma \vdash c_i : C \downarrow i\#k (i=1..k)}{\Gamma \vdash \mathrm{case}_k(f, C, c_{0\ldots(k-1)}) : C \downarrow f}\ \textit{(FIN-ELIM)} \quad ; \quad \mathrm{case}_k(i\#k, C, c_{0\ldots(k-1)}) \longrightarrow c_i$$

$$\textit{(CASE-REDUCTION)} \quad ; \quad \frac{f_1 \leqslant f_2 \quad C_1 \leqslant C_2 \quad c_{i,1} \leqslant c_{i,2}}{\mathrm{case}_k(f_1, C_1, c_{0\ldots(k-1),1}) \leqslant \mathrm{case}_k(f_2, C_2, c_{0\ldots(k-1),2})}\ \textit{(CASE-COMPAT)} \quad .$$

***Определение* 3.2.** *Равенство на конечных типах задаётся с использованием следующих правил:*

$$a =_{\#k} b \Leftrightarrow \mathbf{let} \quad \text{(FIN-EQ)}$$
$$\mathrm{is}_i \equiv \mathbf{case}_k\left(b, \mathbf{Type}, c_{0,\ldots,i-1,i+1,\ldots,k-1} \equiv \#0, c_i \equiv \#1\right)$$
$$\mathbf{in} \quad \mathbf{case}_k\left(a, \mathbf{Type}, \mathrm{is}_0, \mathrm{is}_1, \ldots, \mathrm{is}_{k-1}\right)$$

$$\mathbf{refl}_{\#k}\ a \longrightarrow \mathbf{ua}\left(a, a, 0\#1\right) \quad \text{(FIN-REFL-REDUCTION)}$$

$$J\left(\#k, a, b, \mathbf{case}_k\left(1, C, c_0, \ldots, c_{k-1}\right), \gamma, x\right) \longrightarrow \mathbf{let} \quad \text{(CASE-EQ-REDUCTION)}$$
$$D \downarrow a' \equiv \Pi\left(b' : \#k\right) . \left(\gamma' : a' =_{\#k} b'\right) . \left(x' : C \downarrow_1 a' \downarrow \mathbf{refl}\ a'\right) . C \downarrow_1 b' \downarrow \gamma'$$
$$c_{i,j} \downarrow \gamma' \equiv$$

$$
\begin{aligned}
& \lambda\ C \downarrow i\#k \downarrow \mathbf{refl}\ i\#k\ .\ 0 \text{ при } i = j \\
& \lambda\ C \downarrow i\#k \downarrow \mathbf{refl}\ i\#k\ .\ \mathbf{case}_0 \left(C \downarrow j\#k \downarrow \gamma', \gamma'\right) \text{ при } i \neq j \\
c_i \quad \equiv \quad & \lambda \left(b' : \#k\right) . \left(\gamma' : i\#k =_{\#k} b'\right) . \left(x' : C \downarrow_1 i\#k \downarrow \mathbf{refl}\ i\#k\right) . \\
& \mathbf{case}_k \left(b', \left(D \downarrow i\#k\right) \cdot 0, c_{i,0}, \dots, c_{i,k-1}\right) \\
\mathbf{in}\ \mathbf{case}_k & \left(a, D \downarrow 0, c_0, \dots, c_{k-1}\right) \cdot b \cdot \gamma \cdot x
\end{aligned}
$$

3.1.2. Натуральные числа

***Определение* 3.3.** *Натуральные числа формируются с использованием следующих правил вывода:*

$$
\frac{\Gamma \mathrm{wf}}{\mathbb{N} : \mathrm{Type}_i}\ \textit{(NAT-FORM)} \quad ; \quad \frac{\Gamma \mathrm{wf}}{\mathbb{0} : \mathbb{N}}\ \textit{(NAT-INTRO-0)} \quad ; \quad \frac{\Gamma \vdash n : \mathbb{N}}{\mathbb{S}n : \mathbb{N}}\ \textit{(NAT-INTRO-S)} \quad ;
$$

$$
\frac{\Gamma \vdash n : \mathbb{N} \quad \Gamma,\mathbb{N} \vdash A : \mathrm{Type}_i \quad \Gamma \vdash c_0 : A \downarrow \mathbb{0} \quad \Gamma,\mathbb{N},A \downarrow 0 \vdash c_s : A \downarrow (\mathbb{S}1)}{\Gamma \vdash \mathrm{iter}_{\mathbb{N}}(n,A,c_0,c_s) : A \downarrow n}\ \textit{(NAT-ELIM)} \quad ;
$$

$$
\mathrm{iter}_{\mathbb{N}}(\mathbb{0},A,c_0,c_s) \longrightarrow c_0\ \textit{(ITER-0-REDUCTION)} \quad ; \quad \mathrm{iter}_{\mathbb{N}}(\mathbb{S}k,A,c_0,c_s) \longrightarrow c_s \downarrow k \downarrow iter_{\mathbb{N}}(k,A,c_0,c_s)
$$

$$
\textit{(ITER-S-REDUCTION)} \quad ; \quad \frac{n_1 \leqslant n_2 \quad A_1 \leqslant A_2 \quad c_{0,1} \leqslant c_{0,2} \quad c_{s,1} \leqslant c_{s,2}}{\mathrm{iter}_{\mathbb{N}}(n_1,A_1,c_{0,1},c_{s,1}) \leqslant \mathrm{iter}_{\mathbb{N}}(n_2,A_2,c_{0,2},c_{s,2})}\ \textit{(ITER-COMPAT)} \quad .
$$

***Определение* 3.4.** *Типы эквивалентности на натуральных числах задаются с использованием следующих правил вывода:*

$$
\begin{aligned}
& a =_{\mathbb{N}} b \Leftrightarrow \mathbf{let} \qquad \text{(NAT-EQ)} \\
& \mathrm{isZero} \quad \equiv \quad \lambda \left(b' : \mathbb{N}\right) . \mathbf{iter}_{\mathbb{N}}\left(b', \mathbf{Type}, \#1, \#0\right) \\
& \mathrm{isSucc} \downarrow p \downarrow \mathrm{next} \quad \equiv \quad \lambda \left(b' : \mathbb{N}\right) . \mathbf{iter}_{\mathbb{N}}\left(b', \mathbf{Type}, \#0, \mathrm{next} \cdot 1\right) \\
& \qquad \mathbf{in} \quad \mathbf{iter}_{\mathbb{N}} \left(a, \Pi\mathbb{N}.\mathbf{Type}, \mathrm{isZero}, \mathrm{isSucc}\right)
\end{aligned}
$$

$$
\mathbf{refl}\ x \quad \Leftrightarrow \quad \mathbf{ua}\left(x, x, 0\#1\right) \qquad \text{(NAT-REFL-REDUCTION)}
$$

$$
\begin{aligned}
& J\left(\mathbb{N}, a, b, \mathbf{iter}_{\mathbb{N}} \left(1, C, c_0, c_s\right), \gamma, x\right) \longrightarrow \mathbf{let} \qquad \text{(}\mathbf{ITER}\text{-EQ-REDUCTION)} \\
& D \downarrow a' \quad \equiv \quad \Pi \left(b' : \mathbb{N}\right) . \left(\gamma' : a' =_{\mathbb{N}} b'\right) . \left(x' : C \downarrow_1 a' \downarrow \mathbf{refl}\ a'\right) . C \downarrow_1 b' \downarrow \gamma' \\
& c_0 \quad \equiv \quad \lambda \left(b' : \mathbb{N}\right) . \left(\gamma' : \mathbb{0} =_{\mathbb{N}} b'\right) . \left(x' : C \downarrow_1 \mathbb{0} \downarrow \mathbf{refl}\ \mathbb{0}\right) .
\end{aligned}
$$

$$
\begin{aligned}
& \quad \mathbf{iter}_{\mathbb{N}} \left(b', D \downarrow \mathbb{0} \cdot 0, \lambda _ . x' . x', \right. \\
& \quad \lambda\, k . IH . \left(\gamma' : \mathbb{0} = \mathbb{S}k\right) . x' . \\
& \quad \left. \mathbf{case}_0 \left(\mathbf{itoe}(\gamma'), D \downarrow \mathbb{0} \cdot \mathbb{S}k \cdot \gamma' \cdot x'\right)\right) \\
c_s \quad \equiv & \quad \lambda \left(k : \mathbb{N}\right) . \left(IH : D \downarrow k\right) . \left(b' : \mathbb{N}\right) . \left(\gamma' : \mathbb{S}k =_{\mathbb{N}} b'\right) . \\
& \quad \left(x' : C \downarrow_1 \left(\mathbb{S}k\right) \downarrow \mathbf{refl} \left(\mathbb{S}k'\right)\right) \\
& \quad \mathbf{iter}_{\mathbb{N}} \left(b' , D \downarrow \mathbb{S}k \cdot 0, \right. \\
& \quad \lambda \left(\gamma' : \mathbb{S}k = \mathbb{0}\right) . x' . \mathbf{case}_0 \left(\mathbf{itoe}(\gamma'), D \downarrow \mathbb{S}k \cdot \mathbb{0} \cdot \gamma' \cdot x'\right), \\
& \quad \lambda \left(m : \mathbb{N}\right) . \left(IH' : D \downarrow k \downarrow m\right) . \left(\gamma' : \mathbb{S}k = \mathbb{S}m\right) . \left(x' : _ \right) . \\
& \quad \left. IH \cdot m \cdot \mathbf{ua} \right) \\
\mathbf{in} & \quad \mathbf{iter}_{\mathbb{N}} \left(a, D \downarrow 0, c_0, c_s \right) \cdot b \cdot \gamma \cdot x
\end{aligned}
$$

3.1.3. Высшие индуктивные типы

Прежде всего, необходимо определить сокращение для типа, выполняющего роль структуры типа эквивалентности на высших индуктивных типов. Это сокращение будет использовано в последующих определениях:

$$
\begin{aligned}
& Eq \left(A : \mathbf{Type}\right) \left(I : \Pi A.A.\mathbf{Type}\right) \left(a\ b : A\right) : \mathbf{Type} \equiv \Sigma\ \#3.\ \mathbf{case}_3 \left(0, \mathbf{Type}, \right. \\
& \quad a{=}b \\
& \quad I\ a\ b \\
& \quad \left. I\ b\ a\right)
\end{aligned}
$$

***Определение* 3.5.** *Высшие индуктивные типы формируются с использованием следующих правил вывода:*

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, A, A \vdash I : \text{Type}}{\Gamma \vdash \mathbf{HIT}(A,I) : \text{Type}} \textit{(HIT-FORM)} \quad ;$$

$$\frac{\Gamma \vdash \mathbf{HIT}(A,I) : \text{Type} \quad \Gamma \vdash a : A}{\Gamma \vdash \text{elN}(a,A,I) : \mathbf{HIT}(A,I)} \textit{(HIT-ELEM-INTRO-N)} \quad ;$$

$$\frac{\Gamma \vdash \mathbf{HIT}(A,I) : \text{Type} \quad \Gamma \vdash k : \mathbf{N} \quad \Gamma \vdash a \equiv c_0, c_1, \ldots, c_{k+1} \equiv b : A \quad \Gamma \vdash \gamma_i : \text{Eq}(A,I,c_i,c_{i+1})(i=0\ldots k)}{\Gamma \vdash \text{eqI}(k,[c_i],[\gamma_i],A,I) : \text{elN}(a,A,I) =_{\mathbf{HIT}(A,I)} \text{elN}(b,A,I)} \textit{(HIT-EQ-INTRO-I)} \quad ;$$

$$\frac{\Gamma\vdash \mathbf{HIT}(A,I):\mathrm{Type}\quad \Gamma,\mathbf{HIT}(A,I)\vdash C:\mathrm{Type}\quad \Gamma,A\vdash f:C\downarrow \mathrm{elN}(0,A,I)\quad \Gamma\vdash e:\mathbf{HIT}(A,I)}{\Gamma\vdash \mathrm{Hel}(A,I,C,f,e):C\downarrow e}\ \textit{(HIT-ELEM-ELIM)}\quad ;$$

$$\frac{\begin{array}{ll}\Gamma\vdash \mathbf{HIT}(A,I):\mathrm{Type} & \Gamma,A,A,I\downarrow_1 1\downarrow 0\vdash \alpha:C\downarrow \mathrm{elN}(2,A,I)=_{\mathrm{Type}}C\downarrow \mathrm{elN}(1,A,I)\\ \Gamma\vdash a,b:\mathbf{HIT}(A,I) & \Gamma,A,A,I\downarrow_1 1\downarrow 0,C\downarrow \mathrm{elN}(2,A,I),C\downarrow \mathrm{elN}(2,A,I)\vdash \beta:T\\ \Gamma\vdash \gamma:a=_{\mathbf{HIT}(A,I)}b & T\equiv \mathrm{fst}(\mathrm{itoe}(\alpha\uparrow^2))\cdot 1=_{C\downarrow \mathrm{elN}(3,A,I)}0\\ & \Gamma,\mathbf{HIT}(A,I)\vdash H\equiv \mathrm{Hel}(A,I,C,f,0):C\downarrow 0\end{array}}{\Gamma\vdash \mathrm{Heq}(A,I,C,f,\alpha,\beta,a,b,\gamma):J(\mathbf{HIT}(A,I),a,b,C\uparrow,\gamma,H\downarrow a)=_{C\downarrow b}H\downarrow b}\ \textit{(HIT-EQ-ELIM)}\quad .$$

Структура типа эквивалентности на высших индуктивных типах не определена.

***Определение* 3.6.** *Для термов* eqI *вводятся следующие правила нормализации:*

$$\frac{\mathrm{fst}(\gamma_i)\leqslant \mathrm{fst}(\gamma_{i+1})\leqslant 0\#3}{\mathrm{eqI}(k,c_i,\gamma_i,A,I)\longrightarrow \mathrm{eqI}(k-1,[c_0,\ldots,c_i,c_{i+2},\ldots,c_{k+1}],[\gamma_0,\ldots,\mathrm{pair}(0\#3,\gamma_i\circ\gamma_{i+1},\ldots,\gamma_k),A,I])}\ \textit{(HIT-EQ-COMP)}\quad ;$$

$$\frac{c_i\leqslant c_{i+2}\leqslant c_i,\quad \mathrm{fst}(\gamma_i)\leqslant 1\#3,\quad \mathrm{fst}(\gamma_{i+1})\leqslant 2\#3,\quad \mathrm{snd}(\gamma_i)\leqslant \mathrm{snd}(\gamma_{i+1})\leqslant \mathrm{snd}(\gamma_i)}{\mathrm{eqI}(k,c_i,\gamma_i,A,I)\longrightarrow \mathrm{eqI}(k-2,[c_0,\ldots,c_i,c_{i+3},\ldots,c_{k+1}],[\gamma_0,\ldots,\gamma_{i-1},\gamma_{i+2},\ldots,\gamma_k])}\ \textit{(HIT-EQ-UNIT-R)}\quad ;$$

$$\frac{c_i\leqslant c_{i+2}\leqslant c_i,\quad \mathrm{fst}(\gamma_i)\leqslant 2\#3,\quad \mathrm{fst}(\gamma_{i+1})\leqslant 1\#3,\quad \mathrm{snd}(\gamma_i)\leqslant \mathrm{snd}(\gamma_{i+1})\leqslant \mathrm{snd}(\gamma_i)}{\mathrm{eqI}(k,c_i,\gamma_i,A,I)\longrightarrow \mathrm{eqI}(k-2,[c_0,\ldots,c_i,c_{i+3},\ldots,c_{k+1}],[\gamma_0,\ldots,\gamma_{i-1},\gamma_{i+2},\ldots,\gamma_k])}\ \textit{(HIT-EQ-UNIT-L)}\quad .$$

***Определение* 3.7.** *Операция обращения на элементах типа* $\mathrm{Eq}(A,I,a,b)$ *вводится с использованием следующих правил редукции:*

$$\mathrm{pair}(0\#3,\delta)^{-1}\longrightarrow \mathrm{pair}(0\#3,\delta^{-1})\ \textit{(HIT-EQ-INV-0)}\quad ;$$

$$\mathrm{pair}(1\#3,\delta)^{-1}\longrightarrow \mathrm{pair}(2\#3,\delta)\ \textit{(HIT-EQ-INV-1)}\quad ;$$

$$\mathrm{pair}(2\#3,\delta)^{-1}\longrightarrow \mathrm{pair}(1\#3,\delta)\ \textit{(HIT-EQ-INV-2)}\quad .$$

Первое правило выполняет композицию элементов типов идентичности на A, стоящих последовательно в цепи. Второе и третье правило позволяет взаимно уничтожать генераторы типов идентичности и взаимно обратные к ним.

Далее считается, что термы eqI всегда представлены в нормальной форме. После любого изменения подтермов eqI проводится процедура нормализации.

***Определение* 3.8.** *Для высших индуктивных типов задаются следующие правила редукции:*

$$\text{refl}(\text{elN}(a,A,I)) \longrightarrow \text{eqI}(0,[a,a],[\text{pair}(0\#3,\text{refl}_A a)],A,I) \ \textit{(HIT-REFL-REDUCTION)} \quad ;$$

$$\text{eqI}(k,[c_0,\ldots,c_{k+1}],[\gamma_0,\ldots,\gamma_k],A,I)^{-1} \longrightarrow \text{eqI}(k,[c_{k+1},\ldots,c_0],[\gamma_k^{-1},\ldots,\gamma_0^{-1}],A,I) \ \textit{(HIT-INV-REDUCTION)} \quad ;$$

$$\text{eqI}(k,[c_i],[\gamma_i],A,I)\circ\text{eqI}(l,[d_j],[\delta_j],A,I) \longrightarrow \text{eqI}(k+l,[c_i,d_j],[\gamma_i,\delta_j],A,I) \ \textit{(HIT-COMP-REDUCTION)} \quad ;$$

***Определение* 3.9.** *Для термов удаления высших индуктивных типов вводятся следующие правила редукции:*

$$\text{Hel}(A,I,C,f,\text{elN}(a,A,I)) \longrightarrow f\downarrow a \ \textit{(HIT-ELEM-REDUCTION)} \quad ;$$

$$\text{Heq}(A,I,C,f,\alpha,\beta,d,d,\text{eqI}(\mathbb{0},[a],[],A,I)) \longrightarrow \text{refl}(f\downarrow a) \ \textit{(HIT-EQ-0-REDUCTION)} \quad ;$$

$$\mathbf{Heq}\big(\text{A, I, C, f, }\alpha,\beta,\text{d, e, }\mathbf{eqI}\big(\text{S k, }[\text{c}_\text{i}],[\gamma_\text{i}],\text{A, I}\big)\big) \longrightarrow \mathbf{let} \quad \text{(HIT-EQ-S-REDUCTION)}$$

$$\begin{aligned}
&\text{prev} && \equiv && \mathbf{Heq}\big(\text{A, I, C, f, }\alpha\text{, d, }\mathbf{elN}(\text{c}_\text{k,}\text{ A, I}), \mathbf{eqI}\big(\text{k, }[\text{c}_{0\ldots\text{k}}], [\gamma_{0\ldots\text{k-1}}], \text{A, I}\big)\big)\\
&\delta && \equiv && \text{eqN}\big(1, [\text{c}_\text{k}, \text{c}_{\text{k+1}}], [\mathbf{pair}(0\#3, \mathbf{snd}\ \gamma_\text{k})], \text{A, I}\big)\\
&\text{e}_\text{k} && \equiv && \mathbf{elN}(\text{c}_\text{k}, \text{A, I})\\
&\text{e}_{\text{k+1}} && \equiv && \mathbf{elN}(\text{c}_{\text{k+1}}, \text{A, I})\\
&\varepsilon && \equiv && \lambda\mathbf{HIT}(\text{A, I}).(\text{e}_\text{k} = 0).\ \text{J}\big(\mathbf{HIT}(\text{A, I}), \text{e}_\text{k}, 1, \text{C}\uparrow, 0, \text{f}\downarrow\text{e}_\text{k}\big)\\
&\eta && \equiv && \lambda\mathbf{HIT}(\text{A, I}).(\text{e}_\text{k} = 0).\ \text{J}\big(\mathbf{HIT}(\text{A, I}), \text{e}_{\text{k,}}\ 1, \text{C}\downarrow\text{e}_\text{k} =_{\mathbf{Type}} \text{C}\uparrow, \varepsilon\cdot 1\cdot 0, \mathbf{refl}\ \text{C}\downarrow\text{e}_\text{k}\big)\\
&\mathbf{case}_0 && \equiv && \text{J}\big(\mathbf{HIT}(\text{A, I}), \text{e}_\text{k}, \text{e}_{\text{k+1}}, \mathbf{fst}\ (\mathbf{itoe}\ \eta\cdot 1\cdot 0)\cdot\text{e}_\text{k} =_{\text{C}\uparrow} 1, \delta, \mathbf{refl}\ \text{e}_\text{k}\big)\\
&\mathbf{case}_1 && \equiv && \beta\downarrow_4\text{c}_\text{k}\downarrow_3\text{c}_{\text{k+1}}\downarrow_2\mathbf{snd}\ \gamma_\text{k}\downarrow_1\mathbf{elN}(\text{c}_\text{k}, \text{A, I})\downarrow\mathbf{elN}(\text{c}_{\text{k+1}}, \text{A, I})
\end{aligned}$$

$$\begin{array}{lll}
\mathbf{case}_2 & \equiv & \big(\beta \downarrow_4 c_{k+1} \downarrow_3 c_k \downarrow_2 \mathbf{snd}\ \gamma_k \downarrow_1 \mathbf{elN}(c_{k+1}, A, I) \downarrow \mathbf{elN}(c_k, A, I)\big)^{-1} \\
\mathrm{curr} & \equiv & \mathbf{case}_3\ \big(\mathbf{fst}\ \gamma_k, f \downarrow \mathbf{elN}(c_k, A, I) = f \downarrow e, \mathbf{case}_0, \mathbf{case}_1, \mathbf{case}_2\big) \\
B & \equiv & J\big(\mathbf{HIT}(A, I), d, 1, C\uparrow, 0, \mathbf{Hel}(A, I, C, f, d)\big) \\
& \mathbf{in} & J\big(\mathbf{HIT}(A, I), e_k, e_{k+1}, B =_{C\uparrow} 1, \delta, \mathrm{prev}\big) \circ \mathrm{curr} \quad .
\end{array}$$

3.1.4. Индуктивные типы

Схема, представленная в следующем определении, позволяет задать индуктивный тип вида (категории) $K = \Pi I.\,\mathrm{Type}$ с позиций, аналогичных денотационной семантике рекурсивных программ. Такая схема основана на методе финитной итерации построения инициальных алгебр и терминальных коалгебр в рамках теории категорий, достаточно полное описание которого есть в [2]. Тип I является типом индексов для определения простых индексированных индуктивных типов. Для обычных индуктивных типов можно принять $I \equiv \#1$. Функтор F задаёт структуру индуктивного типа. В качестве индуктивного типа, соответствующего минимальной неподвижной точке μF рассматривается последовательный набор приближений к этой неподвижной точке, начиная от инициального объекта O категории K. Для перехода на следующий уровень приближения используется функция f, от которой требуется инъективность f-inj. Структура высшего индуктивного типа гарантирует независимость функций из μF в произвольный тип C от уровня приближения.

***Определение* 3.10.** *Схема определения индуктивных типов поддерживается следующими формулами, где параметрами являются I, F, f, $f\text{-}inj$:*

$$\begin{array}{lll}
I & : & \mathbf{Type} \\
K & \equiv & \Pi I.\mathbf{Type} \\
O & \equiv & \lambda I.\#0 \\
F & : & \Pi K.K \\
F^n & \equiv & \lambda K.\ \mathbf{iter}_{\mathbb{N}}\ (n, K, 0, F \cdot 0) \\
T & \equiv & \Sigma(i : I).(n : \mathbb{N}).(F^n \cdot O \cdot i) \\
f & : & \Pi(i : I).(n : \mathbb{N}).(F^n \cdot O \cdot i).F^{\mathbb{S}\,n} \cdot O \cdot i \\
\text{f-inj} & : & \Pi(i : I).(n : \mathbb{N}).(f_1\ f_2 : F^n \cdot O \cdot i).
\end{array}$$

$$
\begin{array}{lll}
 & & \left(x : f \cdot i \cdot n \cdot f_1 = f \cdot i \cdot n \cdot f_2\right).\ f_1 = f2 \\
E\ a\ b & \equiv & \mathbf{split}\left(\mathbf{Type}, a, \left(a_0, a_r\right) \mapsto \mathbf{split}\left(\mathbf{Type}, b, \left(b_0, b_r\right) \mapsto\right.\right. \\
 & & \quad \mathbf{split}\left(\mathbf{Type}, a_r, \left(a_1, a_2\right) \mapsto \mathbf{split}\left(\mathbf{Type}, b_r, \left(b_1, b_2\right) \mapsto\right.\right. \\
 & & \quad\ \Sigma\left(\alpha : a_0 = b_0\right).\left(\beta : \mathbb{S}a_1 = b_1\right).\ \mathbf{let} \\
 & & \quad\quad a'2 \equiv J\left(I, a_0, b_0, \left(h, \eta\right) \mapsto F^{a_1} \cdot O \cdot h, \alpha, a_2\right) \\
 & & \quad\quad a''2 \equiv J\left(\mathbb{N}, \mathbb{S}a_1, b_1, \left(h, \eta\right) \mapsto F^{h} \cdot O \cdot b_0, f \cdot b_0 \cdot a_1 \cdot a_2', \beta\right) \\
 & & \quad\ \mathbf{in}\ \left(a''2 =_{F^{b1 \cdot O \cdot b_0}} b_2\right)\left.\left.\left.\left.\right)\right)\right)\right) \\
\mathrm{Ind}\left(I, F, f\right) & \equiv & \mathbf{HIT}\left(T, E\ 1\ 0\right) \qquad .
\end{array}
$$

Аналогичная схема, требующая расширения исчисления явными аннотациями размера рассматривается в [1]. Следующий пример призван продемонстрировать возможность описания известных индуктивных типов по предложенной схеме.

***Пример* 3.1.** *Тип списков, составленных из элементов типа A $\mathrm{List}(A)$ задаётся следующим рекурсивным функтором:*

$$
\mathrm{List}\,A \equiv \mathrm{nil} + \mathrm{cons}\,A\,\mathrm{List}\,A \qquad .
$$

Требуется:

1. *построить аналогичное определение* List *в терминах предложенного подхода, совместно с конструкторами* $\mathrm{nil} : \mathrm{List}\,A$ *и* $\mathrm{cons} : A \to \mathrm{List}\,A \to \mathrm{List}\,A$*;*
2. *определить схему индукции, аналогичную предоставляемой системой Coq для типа списков:*

$$
\begin{array}{lll}
\mathrm{ind}_{\mathrm{List}} & : & \Pi\left(A : \mathbf{Type}\right).\left(C : \Pi\left(\mathrm{List}\ A\right).\mathbf{Type}\right).\left(c\text{-}nil : C \cdot \left(nil \cdot A\right)\right). \\
 & & \quad \left(c\text{-}cons : \Pi\left(a : A\right).\left(l : \mathrm{List}\ A\right).C \cdot \left(cons \cdot A \cdot a \cdot l\right)\right). \\
 & & \quad \Pi\left(l : \mathrm{List}\ A\right).\ C \cdot l \qquad .
\end{array}
$$

Решение. 1. Функтор List в предложенной схеме может быть представлен как L. Функция перехода на следующий уровень приближения fFL реализована путём пересборки зависимых пар в требуемый тип:

$$
L\ A \quad \equiv \quad \lambda\left(L' : \Pi\#1.\mathbf{Type}\right).\left(i : \#1\right).\Sigma\left(c : \#2\right).
$$

$$
\begin{aligned}
&\quad\; \mathbf{case}_2\ (\mathbf{Type}, c, \#1, \Sigma A.(L' \cdot i))\\
\mathrm{caseS} \;&\equiv\; \lambda(i : I)\ (k : \mathbb{N})\ (P : \Pi(L^k \cdot O \cdot i).L^{\mathbb{S}\,k} \cdot O \cdot i).\ \lambda(l : L^{\mathbb{S}\,k} \cdot O \cdot i).\\
&\quad\; \mathbf{split}\ (_\,, l, (c, v) \mapsto \mathbf{split}\ (_\,, v, (a, l') \mapsto\\
&\quad\; \mathbf{case}_2(c, _\,, \mathbf{pair}(0\#2, 0\#1), \mathbf{pair}(1\#2, a, P \cdot l'))))\\
\mathrm{fL}\ A \;&\equiv\; \lambda\#1.\mathbb{N}.\mathbf{iter}_{\mathbb{N}}\ (0, \Pi(L^0 \cdot O \cdot 1).L^{S_1} \cdot O \cdot 2,\\
&\quad\; \lambda O.\mathbf{case}_0\ (L^1 \cdot O \cdot 3, 0 \cdot 3),\\
&\quad\; \mathrm{caseS} \cdot 3 \cdot 1 \cdot 0)\\
\mathrm{List} \;&\equiv \lambda\mathbf{Type}.\mathrm{Ind}(\#1, L(0), \mathrm{fL}(0), \mathrm{fL\text{-}inj}(0))
\end{aligned}
$$

Для краткости, формула доказательства инъективности fL-inj опущена. Вместо этого, рассмотрим процесс её построения в форме доказательства. В доказательстве используются обозначения из типа f-inj в определении 3.10. Поскольку тип индексов #1 тривиален, считаем индекс i всегда равным 0#1.

Доказательство проведём по индукции по исходному размеру $n : \mathbb{N}$. База индукции: при размере $\mathbb{0}$ оба аргумента f1, f2 представляют собой элементы типа ложных высказываний #0, поэтому утверждение леммы получается с помощью удаления такого типа.

Шаг индукции. По индуктивной гипотезе, для $n \equiv k$ утверждение леммы верно и необходимо показать его для $n = \mathbb{S}\,k$. Рассмотрим структуру термов f1, f2. Если первый компонент у каждого из них — 0#2, то их вторые компоненты — 0#1, следовательно, тип их равенства тривиален. В случае, когда они имеют различные первые компоненты (0#2 и 1#2), такие же компоненты будет иметь и результат применения к ним fL. Следовательно, тип fst(itoe(x)) нормализуется в #0, исходя из чего можно получить требуемый результат.

В последнем случае, когда первые компоненты f1, f2 равны 1#2, рассмотрим их вторые компоненты как $\mathbf{pair}(a_1 : A, b_1 : L^k\ A\ 0\#1)$ и $\mathbf{pair}(a_2 : A, b_2 : L^k\ A\ 0\#1)$, соответственно. По правилу редукции ITER-S-REDUCION, аргумент x имеет тип $\mathbf{pair}(a_1, f \cdot i \cdot k \cdot b_1) = \mathbf{pair}(a_2, f \cdot i \cdot k \cdot b_2)$. Тогда по гипотезе индукции имеем $y : b_1 = b_2$ и результат может быть построен как $\mathbf{pair}(\mathbf{refl}\ 1\#2, \mathbf{pair}(\mathbf{fst}(\mathbf{itoe}\ x), y))$.

Функции nil и cons задаются следующим образом:

$$
\begin{aligned}
\text{nil A} &\equiv \mathbf{elN}\big(\mathbf{pair}\big(0\#1,\ \mathbf{pair}\big(\mathbb{S}0,\ \mathbf{pair}\big(0\#2,\ 0\#1\big)\big)\big),\ _\ ,\ _\ \big) \\
\text{cons A a l} &\equiv \mathbf{Hel}\big(_\ ,\ _\ ,\ \text{List A},\ (ll) \mapsto \\
&\quad \mathbf{elN}\big(\mathbf{pair}\big(0\#1,\ \mathbf{pair}\big(\mathbb{S}\big(\mathbf{fst}\ (\mathbf{snd}\ ll)\big), \\
&\quad \mathbf{pair}\big(1\#2,\ \mathbf{pair}\big(a,\ \mathbf{snd}\ (\mathbf{snd}\ ll)\big)\big)\big)\big),\ _\ ,\ _\ \big)\big) \quad .
\end{aligned}
$$

2. Приведём процесс построения требуемого терма в форме доказательства. Прежде всего, построим преобразование элементов $t : T \equiv \Sigma(i : \#1).(n : \mathbb{N}).(l : L^n \cdot O \cdot i)$ в $C(\mathbf{elN}(t,_,_))$. Как и ранее, считаем i равным $0\#1$. Разберём $L^n \cdot O \cdot i$ индукцией по n. При $n \equiv 0$ последний компонент t имеет тип $\#\ 0$, для него требуемый результат можно получить с помощью его удаления. В рамках шага индукции, если первый компонент l — $0\#\ 2$, то требуемый результат — c-nil. Если же первый компонент — $1\#2$, то требуемый результат задан $\text{c-cons} \cdot \mathbf{fst}(\mathbf{snd}(l)) \cdot \mathbf{snd}(\mathbf{snd}(l))$. Корректный перенос эквивалентности при повышении уровня приближения задаётся по определению функций c-nil и c-cons, которые оперируют объектами типа List A, а не внутренним представлением.

□

3.1.5. Коиндуктивные типы

Аналогичным образом, путём замены определения функции f и f-inj могут быть определены коиндуктивные типы. Для краткости изложения, в настоящем подразделе представлена подобная схема для неиндексированных индуктивных типов. Схема иллюстрируется примером определения коиндуктивного типа потоков.

***Определение* 3.11.** *Схема определения коиндуктивных типов определяется следующими формулами:*

$$
\begin{aligned}
F &: \Pi\ \mathbf{Type}\ .\ \mathbf{Type} \\
F_{\mathbf{iter}} &: \lambda\ (S : \mathbf{Type})\ .\ (n : \mathbb{N})\ .\ (\mathbf{iter}_{\mathbb{N}}\ (n,\ \mathbf{Type},\ S,\ F \cdot 0)) \\
F_{red} &: \lambda\ (S : \mathbf{Type})\ .\ (n : \mathbb{N})\ .\ (F_{\mathbf{iter}} \cdot S \cdot n)\ .\ (F_{\mathbf{iter}} \cdot \#1 \cdot n) \\
F_{type} &: \Sigma\ (S : \mathbf{Type})\ .\ (s : S)\ .\ \Pi(s : S)\ .\ (n : \mathbb{N})\ .\ F_{\mathbf{iter}}\ S\ n \\
F_{\mathbf{eq}}\ (i\ j : F_{type}) &: \mathbf{split}(i,\ \mathbf{Type},\ (S_1,\ si) \mapsto \mathbf{split}(si,\ \mathbf{Type},\ (si_1,\ si_2) \mapsto \\
&\quad \mathbf{split}(j,\ \mathbf{Type},\ (S_2,\ sj) \mapsto \mathbf{split}(sj,\ \mathbf{Type},\ (sj_1,\ sj_2) \mapsto \Pi\ (n : \mathbb{N})\ . \\
&\quad F_{red} \cdot S_1 \cdot (si_2 \cdot si_1 \cdot n) = F_{red} \cdot S_2 \cdot (sj_2 \cdot sj_1 \cdot n)))))
\end{aligned}
$$

$$\mathrm{Coind}(F, F_{red}) \equiv \mathbf{HIT}(F_{type}, F_{\mathbf{eq}}) \quad .$$

***Пример* 3.2.** *Тип бесконечных потоков, составленных из элементов типа A — $Stream(A)$ задаётся следующим рекурсивным функтором:*

$$\mathrm{FStream}(A) \equiv A \times \mathrm{FStream}(A) \quad .$$

Требуется:

1. *определить тип бесконечных потоков с использованием предлагаемого подхода совместно с термом удаления* $out_{Stream} : \Pi Stream.FStream \cdot Stream$ *;*
2. *показать возможность определения элементов такого типа с помощью анаморфизма:*

$$ana_{Stream} : \Pi\,(S : \mathbf{Type})\,.\,(\Pi\,S.\,\mathrm{FStream}\cdot S)\,.\,S\,.\,\mathrm{Stream} \quad .$$

Решение. Функтор потоков и требуемый для определения терм F_{red} в предложенной схеме могут быть заданы следующим образом:

$$
\begin{aligned}
\mathrm{FStream} &\equiv \lambda\,(S : \mathbf{Type})\,.\,\Sigma\,A\,.\,S \\
F_{red} &\equiv \lambda\,(S : \mathbf{Type})\,.\,(n : \mathbb{N})\,.\,\mathbf{iter}_{\mathbb{N}}\,(n, \\
&\quad (m) \mapsto \Pi(F_{\mathbf{iter}} \cdot S \cdot m).F_{\mathbf{iter}} \cdot \#1 \cdot m, \\
&\quad \lambda\,S.\,0\#1, \\
&\quad (k, prev_{red}) \mapsto \lambda\,(t : F_{\mathbf{iter}} \cdot S \cdot (\mathbb{S}k))\,. \\
&\quad\ \mathbf{split}\,(t, F_{\mathbf{iter}} \cdot \#1 \cdot (\mathbb{S}k), (a, s) \mapsto \\
&\quad\ \mathbf{pair}(a, prev_{red} \cdot s))) \\
\mathrm{Stream} &\equiv \mathrm{Coind}(\mathrm{FStream}, F_{red}) \quad .
\end{aligned}
$$

1. Для определения терма удаления необходимо задать требуемое отображение для элементов типа-носителя высшего индуктивного типа потока, после чего показать, что такое отображение сохраняет внешний тип идентичности F_{eq}. При заданном доказательстве последнего факта α, требуемое определение может быть получено следующим образом:

$$
\begin{aligned}
out_{Stream} &\equiv \lambda\,(s : \mathrm{Stream})\,.\,\mathbf{Hel}(F_{type}, F_{\mathbf{eq}}, R \equiv \mathrm{FStream} \cdot \mathrm{Stream}, \\
&\quad (t : \mathrm{FType}) \mapsto \mathbf{split}(t, R, (S, t') \mapsto \mathbf{split}(t', R, (s, f) \mapsto \\
&\quad \mathbf{split}(f \cdot s \cdot \mathbb{S}\mathbb{0}, R, (a, s') \mapsto
\end{aligned}
$$

$$\mathbf{pair}\left(a, \mathrm{ana}_{\mathrm{Stream}} \cdot S \cdot f \cdot s'\right)\Big)\Big)\Big),$$
$$\alpha, s\Big) \quad .$$

Терм α может быть получен с использованием структуры эквивалентности на зависимых парах и типа идентичности, который получается из аргумента для первого приближения потока.

2. В предложенной схеме анаморфизм задаётся тривиально с использованием конструктора элемента высшего индуктивного типа:

$$\mathrm{ana}_{\mathrm{Stream}} \equiv \lambda\left(S : \mathbf{Type}\right) . \left(c : \Pi\, S\, .\, \mathrm{FStream} \cdot S\right) . \left(s : S\right) .$$
$$\mathbf{elN}\Big(\mathbf{pair}\left(S, s, c\right), F_{\mathrm{type}}, F_{\mathbf{eq}}\Big)$$

Предложенное решение было проверено с использованием средства Coq с дополнительными определениями, соответствующими отдельным положениям разновидности исчисления, предложенной автором. Исходный код спецификации, использованной для такой верификации, представлен в приложении В. □

В числе направлений дальнейшей работы в области определения индуктивных и коиндуктивных типов в предложенной автором разновидности исчисления следует, в первую очередь, снизить количество технических доказательств, необходимых для задания простых индуктивных типов. Предположительно, аналогичным образом могут быть определены и коиндуктивные типы. Кроме того, могут быть рассмотрены перспективы применения такой схемы для описания индексированных индуктивных типов с неоднородной структурой индекса. Такие типы могут задаваться в виде уравнения:

$$F' \;:\; \Pi K.\Sigma\, \mathbf{Type}.\left(\Pi 0.I\right)$$
$$F \;\equiv\; \lambda K.\lambda I.\, \mathbf{split}\left(\mathbf{Type}, F' \cdot 1, \Sigma 1.\left(1 \cdot 0\right) = 2\right) \quad .$$

3.2. Реализация макета программного средства

Для дальнейшего использования в контексте цели рассматриваемого исследования, предложенная разновидность исчисления должна быть реализована в виде

макета языка формальной спецификации и программного средства проверки типов термов такого языка.

Макет программного средства реализован автором на языке ECMAScript стандарта версии 6. Этот выбор обусловлен тем фактом, что среда исполнения ECMAScript позволяет легко транслировать в этот язык термы базового языка. Исполнение функций, соответствующих термам, возможно в едином пространстве имён. К тому же, язык ECMAScript поддерживает тип данных ассоциативных массивов (словарей) и прототипное объектно-ориентированное программирование. Эти средства широко используются при реализации программ.

Аналогичными перечисленным выше характеристиками с позиций трансляции и динамического исполнения обладают и многие языки семейства LISP. Достаточно слабое развитие средств поддержки разработки для этих языков обусловлено их невысокой популярностью. Возможности с позиций изменяемых и наследуемых ассоциативных массивов достаточно сильно различаются среди языков этого семейства. Тем не менее, такие среды также могут быть использованы для реализации макета базового языка. Преимуществом такого подхода может стать возможность использования встроенного в среду алгоритма синтаксического разбора.

3.2.1. Синтаксис

Синтаксис макета предлагаемого автором языка основан на синтаксисе семейства языков LISP. Это сделано, прежде всего, для упрощения реализации, а также в качестве задела для дальнейшего использования макета языка с целью описания предметно-ориентированных языков программирования. Кроме того, такой синтаксис позволяет вводить новые конструкции для сокращённой записи часто используемых шаблонов выражений. Синтаксис макета языка в нотации ABNF представлен на листинге 1. Следует отметить, что в этой нотации число слева от оператора * означает минимальное количество повторений.

Синтаксис макета языка состоит из двух основных элементов — атомы и списки. Атомы могут представлять собой натуральные числа, сокращения для описания

⟨*expr*⟩ ::= ⟨*atom*⟩ | ⟨*paren-list*⟩

⟨*symbol-or-braces*⟩ ::= ⟨*symbol*⟩ | ⟨*brace-list*⟩

⟨*expr-or-brackets*⟩ ::= ⟨*expr*⟩ | ⟨*bracket-list*⟩

⟨*paren-list*⟩ ::= ‘(’ 1*(expr LWSP) ‘)’

⟨*bracket-list*⟩ ::= ‘[’ 1*(expr LWSP) ‘]’

⟨*brace-list*⟩ ::= ‘{’ 1*(expr LWSP) ‘}’

⟨*atom*⟩ ::= ⟨*number*⟩ | ⟨*fin-type*⟩ | ⟨*fin-elem*⟩ | ⟨*symbol*⟩ | ‘_’

⟨*number*⟩ ::= *DIGIT

⟨*fin-type*⟩ ::= ‘#’ ⟨*number*⟩

⟨*fin-elem*⟩ ::= ⟨*number*⟩ ‘#’ ⟨*number*⟩

⟨*symbol*⟩ ::= ‘`any Unicode letter`’ *(‘`any Unicode non-whitespace character`’)

Листинг 1: Синтаксис базового языка в нотации ABNF

конечных типов и символы, которые используются в качестве имён переменных и ключевых слов. Использование Unicode в символах позволяет вводить синтаксические конструкции с греческими буквами, аналогично используемым в предыдущих разделах главы. Кроме того, дополнительный атом _ используется для введения новых метапеременных для рассматриваемого в подразделе 3.2.5 алгоритма частичного вывода типов.

Списки, в отличие от LISP-подобных языков, в рамках макета базового языка рассматриваются трёх видов — списки с круглыми, квадратными и фигурными скобками, соответственно. При этом на верхнем уровне в выражениях допускается использование только списков с круглыми скобками. Списки с фигурными и квадратными скобками используются при разборе синтаксических конструкций языка, которые приведены в следующем подразделе.

Списки с круглыми скобками предназначены для выделения основных синтаксических конструкций языка. Список $(x\ y_1\ y_2\ \ldots\ yk)$, как будет показано далее, может быть интерпретирован как последовательность приложений $x \cdot y_1 \cdot y_2 \ldots \cdot yk$.

Списки с фигурными скобками используются для описания типов переменных. На последнем месте в фигурных скобках должен стоять тип. Например, определение зависимой суммы (Σ {a A} B) в системе записи, которая используется в предыдущих разделах, может быть интерпретировано следующим образом: $\Sigma A.(a) \mapsto B(a)$.

Списки с квадратными скобками задают именованный терм. Кроме дополнительных синтаксических конструкций, рассматриваемых далее, списки с квадратными скобками используются при описании зависимых пар. Например, запись (**pair** T [a (x y z)] (f a)) интерпретируется как $\mathrm{pair}_T(x \cdot y \cdot z, (a \mapsto f \cdot a))$ или, эквивалентно, $\mathrm{pair}_T(x \cdot y \cdot z, f \cdot 0)$.

Для разбора использован метод комбинаторов парсеров на основе грамматик PEG. Стадию синтаксического разбора в целом можно охарактеризовать как преобразование строки символов в абстрактное синтаксическое дерево, структура которого представлена в виде грамматики в листинге 1.

3.2.2. Основные конструкции

На верхнем уровне допускаются следующие далее синтаксические конструкции.

Конструкция `define` (сокращение `def`) используется для введения новых термов в контекст. (**define** <<symbol> | {<symbol> <expr>}> <expr>). Опциональное использование фигурных скобок позволяет уточнить и проверить требуемый тип вводимого выражения.

Конструкция check (check <e_1 expr> <e_2 expr>) выполняет проверку того, что выражение e1 имеет тип e2.

Конструкция (import <string>) загружает в контекст содержимое внешнего файла.

Кроме того, на верхнем уровне можно использовать любые синтаксические конструкции выражений — в этом случае будет выполнена проверка типов выражения и вычисление его значения, а само выражение будет доступно под именем $.

Соответствие конструкций, вводимых в предыдущих разделах главы синтаксическим конструкциям языка приведено в таблице 3.1.

Таблица 3.1:
Соответствие между синтаксическими конструкциями языка и термами исчисления

Название	**Терм**	**Синтаксические конструкции**
Переменная	$0, 1, \ldots, n$	(**var** <number>)
Тип	Type	**Type**
Зависимое произведение	$\Pi(a : A).B.C$	(**pi** 1*(<symbol-or-braces>) <expr>) (Π 1*(<symbol-or-braces>) <expr>)
λ-абстракция	$\lambda(a : A).B.c$	(**fn** 1*(<symbol-or-braces>) <expr>) (λ 1*(<symbol-or-braces>) <expr>)
Приложение	$x \cdot y \cdot z$	(<expr> 1*(<expr>))
Зависимая сумма	$\Sigma(a : A).B.C$	(**sigma** 2*(<symbol-or-braces>)) (Σ 2*(<symbol-or-braces>))
Пара	$\mathrm{pair}_T(a,\mathrm{pair}_U(b,c))$	(**pair** <expr> 2*2(<expr-or-brackets>)) (**tuple** <expr> 2*<expr-or-brackets>)
Разделение	$\mathrm{split}(C,a,r)$	(**split** <expr> <expr> <expr>)
Тип идентичности	$a =_A b$	(**eq** [<A expr>] <a expr> <b expr>) (= [<A expr>] <a expr> <b expr>)
Рефлексивность	$\mathrm{refl}_A a$	(**refl** [<A expr>] <a expr>)
Удаление типа идентичности	$J(A,a,b,C,\gamma,x)$	(**tran** [<A expr>] <a expr> <b expr> <C expr> <γ expr> <x expr>)
Конструктор аксиомы унивалентности	$\mathrm{ua}_A(f,g,x)$	(**ua** [<A expr>] <f expr> <g expr> <x expr>)
Деструктор аксиомы унивалентности	$\mathrm{itoe}\gamma$	(**itoe** <γ expr>)
Конечный тип	$\#k$	<fin-type>
Элемент конечного типа	$i\#k$	<fin-elem>
Разбор вариантов	$\mathrm{case}_k(f,C,c_{0\ldots(k-1)})$	(**case** <k number> <C expr> <symbol> <f expr> *(<ci expr>))
Тип натуральных чисел	$\mathbb{N}$	Nat, ℕ
Натуральное число	$\mathbb{0}, \ldots$	<number>
Следующее за натуральным числом	$\mathbb{S}$	(**succ** <expr>)
Принцип индукции для натуральных чисел	$\mathrm{iter}_{\mathbb{N}}(n,A,c_0,c_s)$	(**iter** <n expr> <A expr> <c_0 expr> <cs expr>)
Высший индуктивный тип	$\mathbf{HIT}(A,I)$	(**HIT** <A expr> <I expr>)
Элемент высшего индуктивного типа	$\mathrm{elN}(a,A,I)$	(**elN** <a expr> <A expr> <I expr>)
Элемент типа идентичности для высшего индуктивного типа	$\mathrm{eqI}(k,[c_i],[\gamma_i],A,I)$	(**eqI** <k number> *(<ci expr>) *(<γi expr>) <A expr> <I expr>)
Удаление элемента высшего индуктивного типа	$\mathrm{Hel}(A,I,C,f,\alpha,e)$	(**Hel** <A expr> <I expr> <C expr> <f expr> <α expr> <e expr>)
Удаление типа идентичности высшего индуктивного типа	$\mathrm{Heq}(A,I,C,f,\alpha,a,b,\gamma)$	(**Heq** <A expr> <I expr> <C expr> <f expr> <α expr> <a expr> <b expr> <γ expr>)

В качестве выражения может быть использован также сокращение `adt`. Эта нетривиальная конструкция позволяет с использованием конечных типов в виде $\Sigma\#\mathrm{k}.\mathbf{Type}$ определить тип, задаваемый набором конструкторов. Эти механизмы аналогичны тем, которые используются в языке Haskell с поправкой на то, что для индуктивных типов необходимо использовать соответствующие конструкции, предложенные в подразделах 3.1.4 и 3.1.5.

Для зависимых произведений и λ-абстракций, зависимых сумм и let-записи в рамках макета базового языка вводится сокращённая запись нескольких вложенных термов с одним конструктором. Аналогичная запись использовалась в предыдущих разделах главы. В зависимой сумме и паре допускается указание символа-идентификатора для последнего компонента, что может быть использовано для реализации модулей. Для вложенных пар (кортежей) с использованием представленной далее схемы вывода типов реализовано сокращение **tuple**.

Для задания спецификаций внешних библиотек используется дополнительная примитивная конструкция $\left(\mathbf{extern}\ \langle\mathrm{s\ string}\rangle\ \langle\mathrm{n\ number}\rangle\ \langle\mathrm{e\ expr}\rangle\right)$. Идентификатор, состоящий из строки s и натурального числа-индекса n, используется при исполнении кода, а ожидаемый тип e используется при проверке типов. С использованием этой конструкции можно аксиоматически вводить функции, входящие в состав внешних библиотек, и их спецификации.

3.2.3. Преобразование в термы

Для каждого терма хранится следующая дополнительная информация: уровень; имя файла с исходным кодом и позиция в нём синтаксической конструкции, соответствующей терму; имена переменных и термов, ассоциированные с данным термом. Такой набор метаданных используется для вывода ошибок и для реализации нетривиальных синтаксических конструкций. Набор метаданных, в случае необходимости, может быть аналогично языку Clojure расширен пользовательскими атрибутами. Уровень терма используется для раннего обнаружения ошибок. Соответствие такого

уровня требуемому проверяется для всех подтермов при создании нового терма, а также в точках входа каждой из процедур частичного разрешения отношений.

Для термов удаления в реализации, в отличие от определений в предыдущих разделах, удобнее использовать вместо расширения контекста λ-абстракцию. Эквивалентность этих двух представлений тривиально следует из правила β-редукции. При этом от правил вывода в предыдущих разделах главы в первую очередь требуются минимализм и взаимная независимость. Однако для макета базового языка форма записи с λ-абстракцией позволяет избежать усложнения синтаксиса. Кроме того, в составе λ-абстракции могут быть указаны дополнительные аннотации типов.

Трансляция абстрактного синтаксиса в объекты, соответствующие термам, производится с использованием глобального и локального окружений имён. Такие окружения ставят в соответствие именам переменных их индексы де Брёйна. В глобальное окружение при этом попадают имена термов, заданные на верхнем уровне с помощью конструкции `define`.

Схема трансляции каждой синтаксической конструкции в терм определяется отдельно. Фактически, этап преобразования абстрактных синтаксических деревьев аналогичен стадии раскрытия макросов при интерпретации языков из семейства LISP. Отметим, что это свойство предполагается использовать в дальнейшей работе для эффективного описания абстрактного синтаксиса предметно-ориентированных языков.

Сокращённая схема диспетчеризации приведена в листинге 2. Согласно этой схеме, список известных синтаксических конструкций хранится в окружении `Macro` контекста. Это окружение сопоставляет каждому символу соответствующую ему схему разбора. В случае, если символ не является макросом или, если первый элемент списка не является символом, выражение интерпретируется как терм применения. Пример трансляции терма удаления типа идентичности J приведён в листинге 3. Для распознавания шаблонов, представленных в третьем столбце таблицы 3.1 используется набор вспомогательных функций `Macro`, позволяющий осуществлять сопоставление синтаксической конструкции с образцом. Аннотация `typenode` к переменной в образце позволяет проверить, что синтаксическая конструкция, сопо-

ставленная переменной, является символом (именем типа) или списком с круглыми скобками (типом — результатом приложения).

```
context.Macro = Environment({
  'pi': multiArg('pi', pi),
  'app': app,
  'tran': tran, /* ... */
})
export function macroexpand(form, context) {
  return form.match({
    'list': ([length, vector]) =>
      length === 0 ? fail(ReplError.syntax(form, 'Empty list')) :
      vector[0].match({
        'symbol': s => context.Macro(s).catch(() => context.Macro('app')),
        '_': () => context.Macro('app')
      })(length, vector, form, context, macroexpand),
    'symbol': s => s.startsWith('_')
      ? ok(Term.meta(context.level, context.newMeta()))
      : ((s === "Type") ? ok(Term.type(context.level, context.newUniverse()))
        : context.getVariable(s, form.md).fmap(i => Term.var(context.level, i))),
    '_': () => fail(ReplError.syntax(form, "Invalid form at expansion stage"))
  }).bind(x => ok(x.from(form)))
}
```

Листинг 2: Схема диспетчеризации стадии преобразования в термы

```
function tran(l, v, e, context, expand) {
  return Macro.construct(['list', 7, ['symbol', 'tran'],
    ['bind', 'type', 'typenode'],
    ['bind', 'left'], ['bind', 'right'],
    ['bind', 'template', 'typenode'],
    ['bind', 'path', 'typenode'],
    ['bind', 'arg']]).evalPlain(e, empty).bind(env =>
      Error.map([env.type, env.left, env.right, env.template, env.path, env.arg],
        x => expand(x, context))).
    fmap(([type, left, right, template, path, arg] =>
      Term.tran(context.level, type, left, right, template, path, arg)))
}
```

Листинг 3: Пример трансляции терма удаления типа идентичности J

3.2.4. Типизация

На стадии типизации выполняется проверка типа термов, полученных на предыдущей стадии. Задачей стадии типизации является: построить для терма t в контексте Γ новый терм T совместно с выводом $\Gamma \vdash t : T$.

Стадия типизации построена на основе схемы проверки типов, представленной в [20]. Для отношений типизации, совместимости, нормализации и структуры эквивалентности определены взаимно-рекурсивные функции их частичного разрешения, имена которых — typing, compat, normal и equivalence, соответственно.

Функция normal приводит термы-аргументы к головной нормальной форме. Для термов-аргументов функций compat и equivalence перед основной частью функции выполняется процедура приведения их к головной нормальной форме с использованием normal. В предположении, что исчисление обладает свойством нормализации, эти функции и функция normal вызываются только для термов, для которых предварительно была вызвана функция typing. Согласно свойству нормализации, корректно типизированные термы должны иметь нормальную форму. Вспомогательная функция unify выполняет нормализацию и проверяет, что внешний конструктор терма совпадает с требуемым.

Кроме того, при нормализации используется контекст подстановок — список известных термов, которые были подставлены на место определённых переменных в текущем терме. Такая особенность реализации позволяет исключить необходимость повторного вызова функций typing и normal для переменных.

Реализация функций typing и compat выполнена строго согласно правилам вывода, задающим соответствующие отношения. При этом в typing отношения типизации вида $\Gamma \vdash x : X$ в посылках интерпретируется в реализации как $\mathrm{T} := \mathrm{typing}(\Gamma, \mathrm{x})\ ;\ \mathrm{compat}(\mathrm{T}, \mathrm{X})$. Функция normal реализует приведение термов к головной нормальной форме, поэтому метод её реализации следующий. Выполняется последовательное сопоставление нормализуемого терма с образцом, соответствующим левой части правил редукции. В случае совпадения, к терму применяется

правило редукции. Если полученный в результате применения правила терм не находится в головной нормальной форме и отличается от поданного на вход, для него опять вызывается функция normal.

Функция equivalence вызывается из функции normal, а также при типизации термов **itoe** и **ua**. Её применение может быть неудачным в том случае, если производится попытка получения типа эквивалентности для типа, который находится в нейтральной форме. В этом случае для normal вычисление останавливается, а для typing проверка типов завершается с ошибкой.

Рекурсивное применение функции normal к подтермам нормализуемого терма позволяет получить его полную нормальную форму в случае необходимости. В рамках программной реализации объекты, соответствующие термам неизменяемы, при применении правил редукции возвращается новый экземпляр терма. В этой связи с целью повышения производительности используется кеширование типа и нормальной формы терма.

Пример реализации функций проверки типов для терма приложения представлен в листинге 4.Функция shift выполняет подстановку терма с известным или неизвестным типом.

3.2.5. Частичный вывод типов

С целью повышения эффективности использования макета базового языка, для него реализован частичный вывод типов. Наиболее актуальный сценарий применения вывода типов по итогам диссертационного исследования — это снижение количества повторов в текстах спецификаций на базовом языке, например, при аннотации типов вложенных зависимых пар.

Для вывода типов набор термов был расширен термом-метапеременной, который обозначается как «_» или «_<number>». Метапеременные идентифицируются по номерам. Вывод типов реализован в виде отдельной стадии между преобразованием дерева абстрактного синтаксиса в термы и проверкой типов. Таким образом, на вход стадии проверки типов всегда подаётся терм, в котором все метапеременные

```
'app': ([n, left, right]) => ({
  'typing': () =>
    typing(left, env).bind(tLeft =>
    typing(right, env).bind(tRight =>
            unify(tLeft, env, 'pi', ([m, arg, value]) =>
      compat(arg, tRight, env).bind(() =>
      normal(value, env.add_value(right, tRight))
          .fmap(nValue => nValue.shift(Value(right, tRight))))))),
  'normal': () =>
    normal(left, env).bind(nLeft =>
      nLeft.match({
        'lambda': ([m, arg, value]) =>
              normal(value, env.add_value(right)).bind(nValue =>
                        ok(nValue.shift(Value(right)))),
        '_': () => ok(Term.app(n, nLeft, right))
      })),
  'compat': T => ({
    'app': ([m, left1, right1]) =>
      compat(left, left1, env).bind(_ =>
        compat(right, right1, env).bind(_ => ok(true)))
  })
}),
```

Листинг 4: Реализация функций проверки типов для терма приложения

заменены на соответствующие термы. Ошибки, которые могут быть внесены на стадии вывода типов, не будут приняты алгоритмом типизации.

Общая схема вывода типов практически совпадает с проверкой типов. Основные отличия включают использование глобального контекста метапеременных, в котором хранятся термы, соответствующие метапеременным и дополнительная информация, связанная с ними, а именно: тип; структура типа эквивалентности. Функции проверки типов для метапеременных реализованы следующим образом.

Функция разрешения отношения типизации typing для метапеременной проверяет, записан ли тип для данной метапеременной в контексте. Если не записан и значение метапеременной неизвестно, создаётся и возвращается новая метапеременная. В контекст при этом записывается, что вновь созданная метапеременная является типом для данной. Если значение метапеременной известно, как правило,

в глобальном контексте уже должен быть указан её тип, однако если это не так, тип вычисляется стандартным образом.

Функция структуры типа эквивалентности $\mathrm{equivalence}$ для метапеременной реализована аналогичным образом — проверка, не сохранено ли значение в контексте, и, при необходимости, выделение новой метапеременной. Таким образом, функции typing и $\mathrm{equivalence}$ для метапеременных всегда завершают выполнение успешно. Функция нормализации normal для метапеременной выполняет подстановку терма, соответствующего переменной, если такой терм записан в глобальном контексте метапеременных.

Функция совместимости compat для метапеременной в правой или левой части ($_i \leqslant u$ или $u \leqslant _i$) выполняет подстановку в качестве значения метапеременной $_i$ терма t, который формируется следующим образом. Конструктор t совпадает с конструктором u, а все подтермы t являются новыми метапеременными. После такой подстановки выполняется функция разрешения совместимости compat для полученных t и u. Результатом такого рекурсивного применения является терм t', совпадающий по своей структуре с термом u. Однако все подтермы Type_j, входящие в состав t' используют новые переменные уровней универсумов j. В граф уровней универсумов добавляются рёбра, соответствующие $t' \leqslant u$ или $u \leqslant t'$, в зависимости от того, какой порядок рассматривался изначально. В отличие от прямой подстановки u вместо $_i$, такая схема на практике позволила расширить набор термов, для которых удаётся успешно вывести типы.

Для вывода типов используется оптимизация, основанная на следующем наблюдении. Согласно правилам вывода, определяющим используемую разновидность исчисления (как и в случае других распространённых разновидностей λ-исчисления с зависимыми типами), если $\Gamma \vdash x : T$ и $\Gamma \vdash T : U$, то $U \leqslant \mathrm{Type}$. Таким образом, при получении типа метапеременной, которая уже является типом некоторой другой метапеременной, может быть сразу возвращено значение Type.

На практике было отмечено, что для сложных по своей структуре термов может потребоваться несколько итераций вывода типов с сохраняемым между итерациями глобальным контекстом. В используемой в настоящей работе реализации для

каждого терма выполняется не более 5 итераций вывода типов, эта константа может быть изменена в коде реализации. Значение было получено в ходе практических экспериментов. Если после 5 итераций не для всех метапеременных были получены значения, стадия вывода типов завершается неудачно.

3.2.6. Особенности реализации

В глобальном состоянии хранится граф индексов универсумов, процесс построения которого рассматривался в разделе 2.1. После завершения работы основного алгоритма проверки типов выполняется проверка допустимости полученного терма с позиций графа уровней универсумов с помощью алгоритма Тарьяна. В случае успешного прохождения проверки, граф уровней универсумов сохраняется в глобальном контексте вместе с термом. Граф хранится в оптимизированном виде: все индексы, входящие в состав сильно связанных компонент заменяются на одну переменную-индекс. Такой подход необходим, чтобы ограничить рост количества таких переменных при использовании полиморфизма универсумов.

Полиморфизм универсумов — это свойство исчисления, согласно которому независимые подстановки одного и того же терма могут использовать различные уровни универсумов. При подстановке переменных из глобального контекста, индексы универсумов в подставляемых термах заменяются на новый набор индексов, а граф уровней дополняется соответствующими рёбрами. В этом случае два независимых вхождения терма могут быть интерпретированы как вхождения на разных уровнях универсумов. Это позволяет расширить набор термов, которые проходят проверку типов, не давая при этом возможности реализации парадоксов, аналогичных рассмотренным в [67]. При исключении ограничений на граф уровней универсумов такой парадокс может быть построен и для макета базового языка. Код макета базового языка, соответствующий структуре парадокса [67], представлен в листинге 5. Терм False успешно проходит типизацию и имеет тип #0, соответствующий ложному типу. В случае, если ограничения на граф уровней учитываются, терм False не проходит типизацию.

```
(define Pow (λ {S Type} (Π S Type)))
(define Univ (Π {X Type} (Π (Π (Pow (Pow X)) X) (Pow (Pow X)))))
(define PPUniv (Pow (Pow Univ)))
(define tau
   (λ { t PPUniv } { X Type } { f (Π (Pow (Pow X)) X) } { p (Pow X) }
      (t (λ { x Univ }
            (p (f ((x X) f)))))))
(define sig (λ { s Univ } ((s Univ) (fn { t PPUniv } (tau t)))))
(define Delta (λ { y Univ } (Π
   (Π { p (Pow Univ) } (sig y p) (p (tau (sig y))))
   #0)))
(define Omega (tau
   (fn { p (Pow Univ) } (pi { x Univ } (sig x p) (p x)) )))
(define False
   ((λ { O (Π {p (Pow Univ)} (pi { x Univ } (sig x p) (p x)) (p Omega) ) }
      (((O Delta) (λ { x Univ } { /2 (sig x Delta) }
         { /3 (Π { p (Pow Univ) } (sig x p) (p (tau (sig x)))) }
            (((/3 Delta) /2) (λ { p (Pow Univ) }
               (/3 (λ { y Univ } (p (tau (sig y)))))))))
      (λ { p (Pow Univ) } (O (λ { y Univ } (p (tau (sig y))))))))
   (λ { p (Pow Univ) } { /1 (Π { x Univ } (sig x p) (p x)) }
      ((/1 Omega) (λ { x Univ } (/1 (tau (sig x))))))))
```

Листинг 5:
Парадокс, возможный при исключении ограничений на граф уровней универсумов

Альтернативное решение, при котором уровни универсумов указываются вручную, было реализовано в первых версиях макета, однако показало себя сложным в применении на практике. Возможно также смешанное решение, используемое языком Agda: в каждое объявление пользователь вводит вручную необходимое количество переменных-уровней универсумов. На уровне исчисления реализуется решётка для таких переменных. Однако подобное решение усложняет представление исчисления как формальной системы, в связи с чем было использовано решение, описанное выше.

Вопросы корректности работы и вычислительной сложности реализации схемы проверки типов входят в число направлений дальнейшей работы. Решение таких вопросов требует предварительного доказательства свойств нормализации. Эта потребность обусловлена тем, что процедура нормализации входит в состав схемы проверки типов. В случае нарушения этого свойства для некоторых термов реализация схемы проверки типов может не завершиться. Кроме того, верификация свойства

корректности схемы проверки типов в терминах самого исчисления невозможна по причинам фундаментального характера: реализация корректного алгоритма проверки типов является, согласно соответствию Карри-Говарда, доказательством непротиворечивости исчисления. Тем не менее, в рамках дальнейшей работы предполагается реализовать проверку типов на базовом языке с доказательством более слабого свойства. Согласно такому свойству, для тех термов, которые успешно проходят проверку типов, может быть построен вывод их типа.

3.3. Выводы по третьей главе

По сокращённому набору предложенных автором правил редукции для высших индуктивных типов, которые представлены в первом разделе главы, можно сделать вывод о принципиальной возможности синтаксического описания этого аспекта гомотопической теории типов. Примеры, рассмотренные в первом разделе, в особенности — подходы к заданию индуктивных и коиндуктивных типов, свидетельствуют о достаточно широкой применимости высших индуктивных типов, в том числе, для описания сложных для существующих систем аспектов теории типов. При этом следует упомянуть, что ценой таких преимуществ является необходимость аксиоматического задания целого ряда типов, а именно: типов идентичности; конечных типов; типа натуральных чисел; высших индуктивных типов.

Второй раздел посвящён реализации макета языка программирования и формальной спецификации и программного средства проверки типов для такого языка на основе предложенной автором разновидности -исчисления с зависимыми типами. Реализация макета позволяет использовать исчисление в следующей главе при описании фрагментов семантики промышленных языков программирования и используемых на практике программ. Кроме того, с макетом программного средства связан целый ряд задач в рамках развития методов, исследуемых в настоящей работе. В первую очередь, к таким задачам относится использование макета для описания семантики предметно-ориентированных языков программирования.

Достоинства предметно-ориентированных языков программирования при их использовании в задачах формальной верификации программного обеспечения отмечены в расширенной библиографии [144].

На основе описанной в предыдущей главе разновидности λ-исчисления с зависимыми типами, в настоящей главе были разработаны макеты языка и программного средства, предназначенного, в свою очередь, построения формальных моделей и верификации свойств программ. В следующей главе рассматриваются вопросы использования разработанных макетов при описании формальной семантики промышленных языков программирования.

Глава 4

Статическая семантика языков программирования

Промежуточное представление для верификации программ, написанных на нескольких языках программирования, должно быть пригодно для эффективного описания семантики этих языков программирования.

В настоящей главе представлена разработанная автором общая модель для описания статической формальной семантики программ, которые реализованы в виде высокоуровневого промежуточного кода, соответствующего стандарту ECMA-335 [41], а именно:

- Части II (ECMA-335 Partition II), разделам 6–21 для статической семантики (разделы 1–4 являются вводными; раздел 5 определяет синтаксис; разделы 22–24 описывают логический и физический формат хранения промежуточного кода на диске);
- фрагментам Части III (ECMA-335 Partition III), разделам 3–4 для семантики типизации (рассматривались только фрагменты спецификации, относящиеся к инструкции).

С учётом общих сведений о стандарте ECMA-335, представленных в первом разделе настоящей главы, предлагаемая общая модель может быть в перспективе использована для описания семантики программ, написанных на различных, достаточно популярных языках программирования, таких как C# и Java.

Во втором разделе главы рассматривается процесс разработки созданной автором модели статической формальной семантики промежуточного кода стандарта ECMA-335 в соответствии с отмеченными ограничениями. Расширенная версия модели статической формальной семантики опубликована в [125].

В целях демонстрации третий раздел главы посвящён вопросам описания динамической формальной семантики минимального подмножества промежуточного представления стандарта ECMA-335 с использованием построенной модели. Этот раздел может рассматриваться в качестве введения для адаптации подхода к описанию семантики на основе монад к макету базового языка.

В четвёртом разделе модели статической и динамической семантики промежуточного кода используются автором для описания семантики фрагмента кода, входящего в состав кода CMS моделирования теплогидродинамических процессов в первом и втором контурах атомных электростанций [175].

4.1. Стандарт ECMA-335

Стандарт ECMA-335 (Common Language Infrastructure) [41] определяет систему типов, метаданные, виртуальную машину и систему команд для нее (Common Intermediate Language, далее — код CIL), а также задает ограничения для используемых библиотек. Все перечисленные положения в совокупности составляют инфраструктуру взаимодействия различных языков программирования, удовлетворяющих стандарту Common Language Infrastructure (CLI). Они обеспечивают возможность выполнения соответствующих этому стандарту программ на любой реализации стандарта, независимо от аппаратных и низкоуровневых системных особенностей платформы.

Существует несколько реализаций CLI, в частности: Microsoft .NET; Mono; Portable.NET; Silverlight. На настоящее время поддерживающие CLI компиляторы разработаны для достаточно большого количества языков. Примерами могут служить: C#; IronPython; несколько реализаций Lisp (L#, IronLisp, #S); реализация виртуальной машины Java IKVM.NET. На Wikipedia представлен неофициальный список компиляторов, поддерживающих CLI, в котором на настоящее время их более 50.

Программы, соответствующие стандарту ECMA-335, распространяются и разворачиваются в пакетах, называемых сборками. Согласно стандарту, сборка представляет

собой набор файлов в совокупности с метаданными — структурой, которая описывает файлы, входящие в сборку, типы данных, определяемые данной сборкой и некоторый дополнительный набор информации. В сборку могут входить как файлы, содержащие код CIL, так и файлы ресурсов, на содержание которых не накладывается никаких ограничений. Файл, содержащий код CIL и/или метаданные, называют модулем. Сборка обязательно включает хотя бы один файл, в котором и содержатся определяющие сборку метаданные. Сборки идентифицируются по имени, версии, региональным параметрам и, опционально, по открытому ключу издателя. Стандарт не запрещает динамическую подгрузку сборки версии, отличной от затребованной, если это допускается конфигурацией среды исполнения. В случае наличия криптографического ключа издателя, сборка содержит также хеш-коды входящих в её состав файлов и цифровую подпись для обеспечения целостности.

Основным видом объявлений CLI является тип данных. Инфраструктура CLI допускает определение типов данных двух видов, а именно — тип-значение и тип-ссылка. Данные ссылочных типов создаются и хранятся в памяти, пока в программе существуют доступные на них ссылки. Данные типа-значения хранятся локально и копируются при каждом использовании. Для единообразной работы с типами-ссылками и типами-значениями стандартом вводится операция упаковки (boxing), которая сохраняет в памяти значение как объект соответствующего типа. Ссылочный тип может быть унаследован от другого ссылочного типа (однако множественное наследование не допускается). Существует также специальный вид ссылочных типов — интерфейсы, используемые для определения множества методов, реализация которых, в своб очередь, требуется от типа, поддерживающего интерфейс. Ссылочный тип может реализовать произвольное количество интерфейсов, если он предоставляет все методы для каждого из интерфейсов.

Основными составными частями типа являются поля и методы. Поле — это типизированная область памяти, которая используется для хранения данных. Метод — это функция, включённая в состав типа. Методы идентифицируются по имени и сигнатуре. Сигнатура метода определяет тип возвращаемого методом значения, коли-

чество и типы параметров метода, способ вызова метода и количество обобщённых параметров метода.

обобщённые параметры являются реализацией типобезопасного параметрического полиморфизма первого порядка с частичной поддержкой ковариантности. Для каждого типа и метода может быть задано произвольное число обобщённых параметров вместе с ограничениями на типы, которые могут быть подставлены вместо этих параметров. Такой подход позволяет создавать структуры данных, существенно не зависящие от типа непосредственно данных, такие как список, хеш-таблица, очередь и другие.

В дополнение к основным составным частям типа, тип может также содержать дополнительные элементы — свойства и события, которые непосредственно не влияют на семантику программного кода. Для того, чтобы предоставить возможность расширения инфраструктуры CLI, вводится понятие пользовательских атрибутов. Пользовательский атрибут — это присоединённый к произвольному объекту метаданных объект определённого типа, который хранится непосредственно в метаданных. Атрибуты безопасности представляют собой выделенный в отдельную сущность вид пользовательских атрибутов, которые используются для проверки прав доступа пользователя к коду для его выполнения в средах с ограниченным уровнем доверия.

Одним из основных свойств CLI является безопасность операций над типами для существенного подмножества программ, называемых в стандарте верифицируемыми. С помощью рассматриваемой в настоящем разделе модели статической формальной семантики могут быть формализованы требования, предъявляемые стандартом к верифицируемым программам. Подробно этот вопрос рассмотрен автором в статье [125].

4.2. Разработка статической формальной семантики

В настоящем подразделе рассматривается описание разработанной в рамках данной работы общей модели статической формальной семантики стандарта ECMA-335.

В связи с достаточно большим объёмом модели, в разделе представлено её общее описание. Исходный код статической формальной семантики на базовом языке приведён в приложении D.

При разработке статической формальной семантики использовались расширения синтаксиса и определения, входящие в состав стандартной библиотеки макета базового языка. Такие определения описаны далее.

Сокращение `adt`, используемое для описания алгебраических типов данных было введено ранее в разделе 3.2.2. Конструкторы, задающие алгебраический тип, записываются в фигурных скобках. При этом в рамках такого определения в область видимости неявно вводятся сокращения для конструкторов вида `имя-типа/имя-конструктора`. Семантика таких сокращений аналогична конструкторам алгебраических типов данных в Haskell.

Тип `List T` представляет собой список элементов типа `T`, вектор `Vector T n` — представляет собой набор элементов типа `T` с заданной длиной `n`. Определение первого типа приведено в качестве примера ранее; тип вектора может быть задан с помощью удаления натуральных чисел. Тип `Map K V` представляет собой словарь, ставящий в соответствие каждому ключу — элементу типа `K` значение — элемент типа `V`, либо значение, показывающее, что словарь не содержит элемента, соответствующего требуемому ключу. В рамках макета базового языка используется тривиальная реализация такого типа на основе списков пар Σ`K.V`, требующая при этом разрешимости равенства на `K`. Для всех типов, используемых при описании статической семантики, равенство является разрешимым.

Кроме того, как будет отмечено далее, для описания формальной семантики стандарта ECMA-335 необходимо определение взаимно-индуктивных типов данных. Такие типы описываются с использованием подхода, аналогичного рассматриваемому в р. 3.1.4, однако вид K в случае взаимно-индуктивных типов представляет собой зависимую сумму типов, входящих в состав определения. Пример такого определения представлен в листинге 6.

4.2.1. Элементарные типы

К элементарным типам относятся:

- типы натуральных чисел с нулем — $\mathbb{N}$ и булевских значений — $\mathbf{B} = 0\#2$;
- тип **F** чисел с плавающей точкой, соответствующих стандарту ISO/IEC 60559 [71];
- тип **String** строк Unicode, соответствующих стандарту ISO/IEC 10646 [70];
- тип **Ide** допустимых идентификаторов, согласно п. II.5.3 ECMA-335 (здесь и далее таким образом обозначаются положения стандарта ECMA-335), представляющий собой пару из строки и доказательства для неё предиката, соответствующего п. II.5.3;
- тип $\mathbf{Ide}_{\text{dot}}$ последовательностей допустимых идентификаторов, разделённых точками, представляющий собой список, элементы которого имеют тип **Ide**;
- тип **Culture** множества строк, являющихся корректными, согласно стандартам ISO 3166 [68] и ISO 639 [69], кодами страны и региона.

Перечисленные типы либо были определены ранее в настоящей работе, либо множество значений их элементов вводится в других стандартах, или (в случае **Ide**) задаётся синтаксическим ограничением на множество символов в строке. Поскольку такие типы представляют собой множества, элементы которых с точки зрения исчисления не имеют особых свойств, в рамках данной работы они могут быть без ограничения общности отождествлены с типами натуральных чисел или конечными типами.

4.2.2. Сборки, модули и типы данных

В листинге 6 представлена формальная спецификация ссылок на модуль и сборку, сигнатуры метода и типа данных.

Тип $\mathbf{Module}_{\text{ref}}$ определяет ссылку на модуль. Модуль идентифицируется по имени и, для однозначности, по хеш-коду. Для простоты предположим, что хеш-коды известны для всех модулей, которые входят в сборки, необходимые для работы программы. Более того, потребуем, чтобы любой используемой в программе паре «имя-хеш» всегда соответствовал в точности один модуль. Тип **Version** определяет версию сборки и, согласно п. II.2.1.4 ECMA-335, представляет собой четыре неотри-

цательных целых числа. Тип ссылок на сборку $\textbf{Assembly}_{\text{ref}}$ содержит имя сборки, её версию, региональные параметры, хеш-сумму файла с метаданными и открытый ключ издателя для проверки целостности сборки.

Типы области видимости объявления **Scope** и тип ссылок на пользовательский тип $\textbf{Class}_{\text{ref}}$ определены взаимно-индуктивно (листинг 6). В настоящей модели определённые в стандарте виды областей видимости сведены к следующим двум:

- объявление верхнего уровня, для которого указаны ссылки на сборку и модуль;
- вложенное объявление, для которого указана ссылка на пользовательский тип данных, в состав которого входит рассматриваемое объявление.

Ссылка на пользовательский тип содержит область видимости объявления, разделённый точками идентификатор и число обобщённых параметров пользовательского типа.

$$
\begin{array}{lll}
\text{FScope} & \equiv \lambda\ (\text{Scope Class}_{\text{ref}} : \textbf{Type})\ .\ \text{adt} & \\
& \quad \{\ \text{Top} : (\ \Sigma\ (\text{assembly} : \text{Assembly}_{\text{ref}}\ )\ .\ (\text{module} : \text{Module}_{\text{ref}}\ ))\ \} & \\
& \quad \mid \{\ \text{Nested} : \text{Class}_{\text{ref}}\ \} & \\
\text{FClass}_{\text{ref}} & \equiv \lambda\ (\text{Scope Class}_{\text{ref}} : \textbf{Type}) & \\
& \quad \Sigma\ (\text{scope} : \text{Scope})\ (\text{name} : \text{Ide}_{\text{dot}})\ (\text{nparams} : \mathbb{N}) & \\
\text{scr} & \equiv \quad \text{Ind}_{\Sigma\textbf{Type}.\textbf{Type}}\ (\lambda\ (\Sigma\textbf{Type}.\textbf{Type})\ .\ (\textbf{split}\ (\Sigma\textbf{Type}.\textbf{Type},\ \textbf{var}\ 0, & \\
& \quad \lambda\ (\text{s c} : \textbf{Type})\ .\ \textbf{pair}\ (\text{FScope} \cdot \text{s} \cdot \text{c},\ \text{FClass}_{\text{ref}} \cdot \text{s} \cdot \text{c})))) & \\
\text{Scope} & \equiv \quad \textbf{fst}\ \text{scr} & \\
\text{Class}_{\text{ref}} & \equiv \quad \textbf{snd}\ \text{scr} &
\end{array}
$$

Листинг 6: Пример взаимно-индуктивного определения
— область видимости определения и ссылка на пользовательский тип данных

Способ вызова метода $\textbf{Method}_{\text{call-conv}}$ определяет, требуется ли от метода ссылка на экземпляр типа, внутри которого он определен (в случае, если метод определен в составе некоторого типа). Вид вызова $\textbf{Method}_{\text{call-kind}}$ показывает, является ли вызов данного метода стандартным (Default), либо необходима упаковка переменного числа параметров (Vararg), или вызов данного метода является вызовом функций кода, не соответствующего стандарту (Unmanaged). В стандарте определяются различные способы вызовов кода, не соответствующего стандарту ECMA-335, которые не рассматриваются в данной работе, так как её содержание не касается вопросов взаимодействия с такими программами и библиотеками.

Сигнатура метода $\mathbf{Method}_{\mathrm{sig}}$ содержит информацию о способе и виде вызова метода, типе возвращаемого значения, типах параметров и количестве обобщённых параметров метода. Типы данных **Type** включают в себя: типы обобщённых параметров типа или метода; пользовательские типы данных; управляемые и неуправляемые указатели на определённый тип; одномерный (вектор) и многомерный массив элементов определённого типа; типы с различными модификаторами; обобщённые типы с заданным списком обобщённых параметров и ссылки на методы с определённой сигнатурой. Стандартные примитивные типы данных могут считаться обычными пользовательскими типами данных, поэтому в явном виде в статическую семантику такие типы не входят. Сигнатура метода и тип данных определяются как взаимно-индуктивные типы.

4.2.3. Пользовательские типы данных

Тип **Access** определяет видимость пользовательского типа данных. Типы, определённые на верхнем уровне (непосредственно на уровне модуля) могут быть либо доступны (Public), либо недоступны (Private) из других сборок. Вложенные (определённые внутри других типов) типы, методы и поля могут иметь дополнительные ограничения на видимость.

Атрибуты специального имени **Special** представляют собой два флага, с помощью которых указывается, что среда исполнения или пользовательские инструментальные средства должны специальным образом интерпретировать имя данного типа. Такой атрибут определен и будет далее использоваться не только для типов, но и для полей, методов, свойств и событий. Примером, иллюстрирующим использование данного атрибута, может служить конструктор экземпляра некоторого типа. Такой конструктор, согласно II.10.5.1 ECMA-335, должен иметь имя `.ctor`, однако во многих языках программирования (например, в языке C#) он носит имя соответствующего типа.

Тип $\mathbf{Class}_{\mathrm{layout}}$ определяет расположение данных экземпляра рассматриваемого типа в памяти. Данные могут быть расположены средой исполнения автоматически

(Auto) или в порядке их определения (Sequential), или расположение данных может задаваться непосредственно для каждого из полей (Explicit).

Тип $\mathbf{Class}_{\mathrm{attr}}$ включает полный набор простых атрибутов типа. Кроме представленных ранее, определение типа может также включать флаги абстрактности, интерфейса, запрета на наследование, сериализуемости, а также порядка вызова статического конструктора.

Определение типа $\mathbf{Class}_{\mathrm{def}}$, помимо идентификатора и атрибутов содержит также ссылку на базовый тип и список ссылок на интерфейсы, которые реализуются рассматриваемым типом.

Тип $\mathbf{Generic}_{\mathrm{variance}}$ определяет, является ли обобщённый параметр ко- или контравариантным. Тип $\mathbf{Generic}_{\mathrm{constraint}}$ задает ограничения на тип обобщённых параметров. обобщённый параметр может быть ограничен на типы-значения, типы-ссылки, типы с конструктором без параметров или ссылочные типы с конструктором без параметров. Объявление обобщённого параметра $\mathbf{Generic}_{\mathrm{def}}$ содержит: имя обобщённого параметра; его номер в содержащем его типе; параметры ковариантности и ограничений на тип; список непосредственных ограничений на базовый тип и реализацию интерфейсов.

4.2.4. Поля и методы

Тип $\mathbf{Field}_{\mathrm{ref}}$ определяет идентификатор поля, который состоит из области видимости, типа и идентификатора поля.

Основные атрибуты поля задаются элементами типа $\mathbf{Field}_{\mathrm{attr}}$. Основными атрибутами поля являются видимость поля, атрибуты специального имени и 4 флага, которые определяют, является ли поле константным, неизменяемым, сериализуемым или статическим.

Поля могут быть инициализированы некоторым значением. Все допустимые инициализаторы полей задаются при помощи элементов типа $\mathbf{Field}_{\mathrm{init}}$. Если инициализатор поля не указан, используется пустое значение None монады Maybe.

Обявление поля $\mathbf{Field}_{\mathrm{def}}$ объединяет значения трех определённых выше типов.

Тип $\mathbf{Method}_{\text{ref}}$ определяет ссылку на метод. Метод идентифицируется по области определения, идентификатору и сигнатуре.

Тип **Param** определяет атрибуты параметра метода, а именно — тип, идентификатор и три флага, означающих опциональность параметра и флаги входного-выходного параметров.

Тип $\mathbf{Method}_{\text{code}}$ определяет, является код метода кодом CLI, либо его реализация должна предоставляться средой исполнения.

Тип $\mathbf{Method}_{\text{attr}}$ задает простые атрибуты метода. К простым атрибутам относятся видимость метода, определённый выше атрибут кода метода, атрибуты специального имени и десять флагов. Флагами задаются следующие ограничения на метод: абстрактность; запрет перегрузки виртуального метода; способы перегрузки метода, виртуальность; статический характер метода; ограничения для среды исполнения и для параллельного исполнения метода.

Объявление метода $\mathbf{Method}_{\text{def}}$ содержит ссылку на метод, атрибуты метода и набор атрибутов для каждого из параметров метода.

4.2.5. Окружения статической семантики

Окружения статической семантики каждому из видов идентификаторов, определённых ранее, ставят в соответствие объявление, обозначаемое таким идентификатором.

Простейшими примерами окружений являются окружения типов, полей и методов ($\mathbf{Class}_{\text{env}}$, $\mathbf{Method}_{\text{env}}$, $\mathbf{Field}_{\text{env}}$). Такие окружения определяются как словари — например, в случае окружения типов: `Map` $\mathbf{Class}_{\text{ref}}$ $\mathbf{Class}_{\text{def}}$. Окружение обобщённых параметров имеет более сложную структуру, так как обобщённые параметры могут принадлежать как типу, так и методу. Для объединения возможных вариантов вводится тип $\mathbf{Generic}_{\text{item}}$, элементы которого являются либо ссылками на пользовательские типы данных, либо ссылками на методы. Такой тип далее используется в качестве типа ключей словаря $\mathbf{Generic}_{\text{env}}$.

Основное окружение статической семантики **Env** объединяет все определённые выше окружения в один объект.

4.2.6. Тело метода

Тело метода, согласно стандарту CLI, состоит из списка локальных переменных и последовательности инструкций, которые могут быть разделены на блоки обработки исключений.

Стандартом определяются четыре типа блоков обработки исключений, которые перечислены далее.

Блок обработки исключений Finally вызывается как при возникновении исключения, так и при нормальном завершении выполнения блока кода. Блок такого типа предназначен для освобождения ресурсов, которые выделяются для использования в его кода. Обработчик Fault вызывается при возникновении исключений произвольного типа и предназначен для обработки исключительных ситуаций, которые не были непосредственно предусмотрены разработчиком. Блок Catch вызвыется при возникновении исключений определённого типа. Обработчик Filter вызывается при возникновении исключений произвольного типа и предназначен для фильтрации исключения по определённому критерию.

Объявление блока обработки исключений **Seh** состоит из номера первой и последней инструкции участка кода, для которого рассматриваются исключения, номера первой и последней инструкции участка кода, обрабатывающего исключения, и типа блока обработки исключения.

Объявление локальной переменной **Local** состоит из номера переменной, её типа и идентификатора. Обращение к локальной переменной в коде метода производится по номеру, идентификатор носит информативный характер.

Опишем некоторые дополнительные типы, которые используются в определениях инструкций. Тип видов сравнений **Compare** определяет допустимые виды сравнений для соответствующих инструкций, а именно: «равно» (Equals), «больше» (GreaterThan), «меньше» (LessThan).

Тип токена метаданных **Token** используется для доступа из кода программы к метаданным, задающим информацию о типах, и может являться ссылкой на поле (F), ссылкой на метод (M) или ссылкой на тип (T).

Возможные типы проверок аргументов **Check**, которые выполняются средой исполнения при выполнении определённых инструкций включают в себя проверку на совпадение типов (Typecheck), на выход за границы массива (Rangecheck) и на обращение к нулевой ссылке (Nullcheck).

Инструкции CIL для удобства определения разделены на группы по типу встроенных параметров. Первая группа ($\mathbf{Instr}_{none}$) содержит инструкции, которые не содержат встроенного параметра. В частности, к этой группе относятся инструкции, соответствующие битовым операциям. Группа $\mathbf{Instr}_{type}$ содержит в качестве встроенного параметра некоторый тип. В первую очередь, к этой группе относятся инструкции, связанные с преобразованием типов. К группам $\mathbf{Instr}_{arg}$ и $\mathbf{Instr}_{local}$ относятся инструкции, содержащие в качестве параметра номер аргумента метода или номер локальной переменной, соответственно. Группы $\mathbf{Instr}_{method}$ и $\mathbf{Instr}_{field}$ содержат инструкции, параметрами которых являются ссылки на методы или поля, соответственно. И, наконец, к группе $\mathbf{Instr}_{other}$ принадлежат прочие инструкции, которые не относятся ни к одной из перечисленных групп. Также, инструкции могут содержать один из префиксов ($\mathbf{Instr}_{prefix}$), которые изменяют семантику инструкции для среды исполнения. Тип **Instr** содержит все допустимые инструкции, вместе с встроенными параметрами.

Статическая семантика тела метода определяется парой типов $\mathbf{Method}_{item}$ и $\mathbf{Method}_{env}$. Тип $\mathbf{Method}_{item}$ полностью задает тело метода. Он содержит объявление метода, набор параметров метода и локальных переменных, число, определяющее наибольший допустимый размер стека исполнения для метода, флаг инициализации локальных переменных, список блоков обрабоки исключений метода, а также набор инструкций и префиксов. В окружении статической семантики **Env** изменяется окружение методов $\mathbf{Method}_{env}$, которое идентификатору метода ставит в соответствие тело метода.

4.2.7. Семантика типизации

Семантика типизации стандарта ECMA-335 определяется как функция типизации, которая каждой инструкции метода ставит в соответствие состояние стека вычислений с точки зрения типов данных.

Примитивные типы данных могут быть определены как ссылки на определённые пользовательские типы данных, которые являются частью стандартной библиотеки классов CLI. Пусть $\mathsf{Corlib}_{\mathrm{ref}}$ это сссылка на сборку стандартной библиотеки CLI, а $\mathsf{Corlib}_{\mathrm{main}}$ — ссылка на главный модуль сборки стандартной библиотеки. Ссылка на базовый стандартный тип `double` будет определяться следующим образом:

$$\begin{aligned}\mathsf{Double} \quad = \quad & \mathbf{Class}_{\mathrm{ref}}(\mathrm{Top}\,\mathsf{Corlib}_{\mathrm{ref}}\,\mathsf{Corlib}_{\mathrm{main}}, \\ & [(1,\mathtt{System}),(2,\mathtt{Double})],0).\end{aligned}$$

Аналогичным образом определяются ссылки и на другие примитивные типы данных CLI, а именно — Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32, UInt64, Boolean, IntPtr, UIntPtr, Single.

Определение семантики типизации в настоящей работе отличается от исходной версии, использовавшей в качестве базовой формальной модели теорию доменов. Средства λ-исчисления с зависимыми типами позволяют задать тип корректных по построению с позиций состояния стека наборов инструкций. Для этого, учитывая ограничения макета базового языка по заданию типов данных с циклическими ссылками, используется представление тела метода в форме ориентированного графа базовых блоков. Базовым блоком называется набор инструкций, не содержащий переходов.

Стек вычислений с точки зрения типов данных представляется как пара из натурального числа n и вектора типов данных длины n. Для определения совместимости стеков при переходе между базовыми блоками используется отношение `compatible` на типах данных, которое является разрешимым с использованием функции `is-compatible`. Такое отношение и функция его разрешения зависят от окружения статической семантики, задающего семантику определяемых типов даных.

Построение семантики типизации метода выполняется с помощью нескольких проходов по методу. В рамках первого прохода определяется количество базовых блоков, номера инструкций, открывающих базовые блоки, а также ребра графа базовых блоков. Далее выполняется проход по инструкциям каждого из базовых блоков, для каждой из которых определяется состояние стека после выполнения каждой из инструкций. Затем для каждого ребра графа базовых блоков производится слияние конечных состояний стеков вычислений — выбирается стек, состоящий из «наименьших» типов данных с позиций отношения `compatible`. В завершение для каждого ребра проверяется, что выбранное для базового блока состояние стека вычислений совместимо с состояниями стека вычислений для всех блоков, из которых выполняется переход в данный.

На этом определение статической формальной семантики стандарта ECMA-335 завершено. Некоторые примеры приложения статической семантики рассмотрены автором в [125].

4.3. Динамическая семантика подмножества CIL

В настоящем разделе рассматриваются вопросы постоения модели динамической семантики подмножества стандарта ECMA-335. Такая модель требуется для решения задач верификации свойств программ, представленных в промежуточном коде стандарта ECMA-335. С её помощью также демонстрируется применимость макета базового языка к описанию динамической семантики языков программирования с использованием подхода на основе монад.

Рассматривается динамическая формальная семантика минимального подмножества стандарта ECMA-335, достаточного для демонстрации применимости макета базового языка к задачам описания полной формальной семантики и верификации свойств программ. В качестве примера использования с помощью построенной модели будет описана динамическая формальная семантика фрагмента кода, приведённого в приложении Е. Исследуемый фрагмент входит в состав расчётного кода

CMS (Compressible Media Solver), который предназначен для вычисления в режиме реального времени теплогидравлических параметров в двухфазной сжимаемой неравновесной парогазокапельной среде с любым числом неконденсируемых газов. Код разработан специалистами ОАО «ВНИИАЭС» и в настоящее время широко применяется в программном обеспечении тренажёров по обучению управлением и отработке действий при чрезвычайных ситуациях персонала атомных электростанций [77].

4.3.1. Монады и преобразователи монад

Как отмечалось в главе 1, используемый в настоящей работе подход к описанию динамической семантики языков программирования на основе монад был предложен в [94], а один из наиболее известных примеров применения этого подхода описывается в [97]. Сам по себе тип монад вводится в терминах макета базового языка аналогично другим реализациям λ-исчисления с зависимыми типами. Монада представляет собой конструктор типа m и две операции — **return** и **bind**. Операция **return** позволяет получить элемент типа $\mathrm{m}\ \mathrm{A}$ из элемента типа A, а операция **bind** позволяет объединять функции, получающие в качестве аргументов элементы типа A и возвращающие $\mathrm{m}\ \mathrm{A}$ в цепочку. Композиция таких операций должна удовлетворять свойствам, которые равносильны коммутативности двух диаграмм в рамках теории категорий. Такие свойства называются аксиомами монад (monad laws).

***Определение* 4.1.** *Тип монад в базовом языке задаётся следующей формулой:*

$$
\begin{array}{lll}
\mathrm{Monad} & \equiv & \Sigma \\
\quad \mathrm{m} & : & \mathbf{Type} \to \mathbf{Type} \\
\quad \mathbf{return} & : & \Pi\ \left\{A : \mathbf{Type}\right\}\ .\ A \to \mathrm{m}\ A \\
\quad \mathbf{bind} & : & \Pi\ \left\{A\ B : \mathbf{Type}\right\}\ \left\{\ ma : \left(\mathrm{m}\ A\right)\ \right\}\ \left\{\ f : \left(\Pi\ A\ .\ \mathrm{m}\ B\right)\right\}\ .\ \mathrm{m}\ B \\
\quad \mathbf{return}_{\mathrm{idLeft}} & : & \Pi\ \left\{f\ x : _\right\}\ .\ \left(\mathbf{bind}\ \left(\mathbf{return}\ x\right)\ f\right) = f\ x \\
\quad \mathbf{return}_{\mathrm{idRight}} & : & \Pi\ \left\{g : _\right\}\ .\ \left(\mathbf{bind}\ g\ \mathbf{return}\right) = g \\
\quad \mathbf{bind}_{\mathrm{comp}} & : & \Pi\ \left\{f\ g\ v : _\right\}\ . \\
 & & \quad \left(\mathbf{bind}\ \left(\mathbf{bind}\ f\ v\right)\ g\right) = \mathbf{bind}\ v\ \left(\lambda\ \left\{x\ _\right\}\ \left(\mathbf{bind}\ g\ \left(f\ x\right)\right)\right)
\end{array}
$$

Описание формальной семантики языков программирования можно разделить на несколько независимых аспектов путём комбинации монад. В качестве примера, иллюстрирующего применение монад, рассмотрим монаду Maybe (доказательства законов монад опущено).

$$
\begin{aligned}
&\mathrm{Maybe} &&\equiv \Sigma \\
&\quad \mathrm{m} &&\equiv \lambda \left\{T : \mathbf{Type}\right\} \left(\mathrm{adt} \left\{ \mathrm{Nothing} : \#1 \right\} \left\{ \mathrm{Just} : \mathrm{T} \right\}\right) \\
&\quad \mathbf{return} &&\equiv \lambda \left\{T : \mathbf{Type}\right\} \left\{ t : T \right\} . \left(\mathrm{Just}\ t\ \right) \\
&\quad \mathbf{bind} &&\equiv \lambda \left\{A\ B : \mathbf{Type}\right\} \left\{ ma : \mathrm{m}\ A\right\} \left\{ f : \Pi A\ .\ \left(\mathrm{m}\ B\right) \right\} . \left(\mathbf{match}\ ma\right. \\
&&& \quad \left[\mathrm{Nothing} \quad \left(\lambda\left(\mathrm{o} : \#1\right).\ \mathrm{o}\right)\right] \\
&&& \quad \left.\left[\mathrm{Just} \quad \left(\lambda\ \left(\mathrm{a} : A\right).\ f \cdot \mathrm{a}\right)\right]\right)
\end{aligned}
$$

Элементом типа Maybe/m · T является либо некоторый элемент типа T с конструктором Just, или же элемент тривиального типа #1 с конструктором Nothing. Семантика последнего значения аналогична семантике нулевого указателя в языках семейства C. Таким образом, монада Maybe описывает обнуляемый тип, аналогично обобщённому типу `Nullable<T>` в языке C# и стандартной библиотеке CLI. При этом последняя (шестая) версия спецификации языка C# содержит и оператор «`?.`», семантика которого аналогична семантике операции **bind** монады Maybe [91]. В частности, эквивалентны два фрагмента кода, представленные в листинге 7.

```
// Фрагмент 1
var g1 = parent?.child?.child;

// Фрагмент 2
Child g1 = null;
if (parent != null) {
    if (parent.child != null) {
        g1 = parent.child.child;
    }
}
```

Листинг 7: Семантика оператора «`?.`» в языке C#

Как видно из примера, монада Maybe с помощью оператора **bind** позволяет скрыть проверки на возможный нулевой результат используемых операций. Кроме монады

Maybe при описании семантики, как правило, используются следующие стандартные монады:

Id — тривиальная монада, функции m и **return** осуществляют идентичное преобразование, операция **bind** — композицию.

Error — монада исключений; аналогична монаде Maybe. Однако вместо нулевого результата она использует некоторый заданный тип исключений. На такой монаде могут быть определены операции выброса исключений throw и обработки исключений catch.

Reader — монада окружения; позволяет считывать данные из некоторого неизменяемого окружения.

State — монада состояния; позволяет считывать и записывать состояние, которое представляет собой элемент некоторого заданного типа.

Для эффективной комбинации нескольких монад в один объект может быть использовано понятие преобразователей монад. Преобразователь монад представляет собой конструктор монад t и операцию lift («повышение»). Операция lift позволяет преобразовать значение типа монады-аргумента $\mathrm{m}\ A$ в значение типа преобразованной монады $\mathrm{t}\ \mathrm{m}\ A$.

***Определение* 4.2.** *Тип преобразователей монад в базовом языке задаётся следующей формулой:*

$$
\begin{aligned}
&\mathrm{MonadTrans} \equiv \Sigma \\
&\quad \mathrm{t} : \Pi \left\{M\ \mathrm{Monad}\right\} .\ \mathrm{Monad} \\
&\quad \mathrm{lift} : \Pi \left\{M\ \mathrm{Monad}\right\} \left\{A\ \mathbf{Type}\right\} .\ \left(M/m\ A\right) \to \left(\left(\mathrm{t}\ M\right)/m\ A\right) \\
&\quad \mathrm{lift}_{\mathrm{ret}} : \Pi \left\{x\ _\right\} .\ \left(\mathrm{lift}\ \left(\mathbf{return}_{\mathrm{m}}\ x\right)\right) = \mathbf{return}_{\mathrm{t}}\ x \\
&\quad \mathrm{lift}_{\mathbf{bind}} : \Pi \left\{x\ f\ _\right\} .\ \left(\mathrm{lift}\ \left(\mathbf{bind}\ x\ f\ \right)\right) = \mathbf{bind}\ \left(\mathrm{lift}\ x\right)\ \left(\lambda\ y.\ \left(\mathrm{lift}\ \left(\mathrm{f} \cdot y\right)\right)\right) \qquad .
\end{aligned}
$$

В настоящей работе как преобразователи монад описаны преобразователи монад состояния StateT, окружения ReaderT и исключений ErrorT.

4.3.2. Модель чисел с плавающей точкой

Фрагмент кода, динамическая семантика которого будет описана в последнем разделе настоящей главы, использует арифметические операции на числах с плавающей точкой. Поэтому при описании семантики такого фрагмента требуется задать и семантику вычислений над числами с плавающей точкой. Такая модель для среды Coq была разработана и апробирована на примере рассматриваемого далее фрагмента кода студентом 5 курса механико-математического факультета С.В. Антоновым в рамках дипломной работы, выполненной им под руководством автора. Краткая информация о модели приведена в приложении C. Модель была адаптирована автором для использования в рамках базового языка.

Несмотря на отличия между языком среды Coq (Gallina) и макетом базового языка с позиций синтаксиса, исходной формальной моделью для обоих языков является λ-исчисление с зависимыми типами. При этом модель чисел с плавающей точкой существенным образом использует такие компоненты стандартной библиотеки, как тип векторов булевских значений с заданной длиной, а также тип натуральных чисел не больше заданного. Оба таких типа могут быть описаны в терминах макета базового языка. Конструкция `Fixpoint` среды Coq используется в модели исключительно в контексте удаления натуральных чисел, поэтому в рамках настоящей работы все составленные с использованием этой конструкции функции были адаптированы в базовый язык с использованием терма удаления натуральных чисел $\mathrm{iter}_{\mathbb{N}}$.

В рамках настоящей работы предполагаем, что такая модель задана в виде типа, соответствующего интерфейсу для моделей арифметики на числах с плавающей точкой. В процессе адаптации к макету базового языка в модель и интерфейс были внесены следующие изменения:

- не использовались типы рациональных и целых чисел, т.к. на настоящее время такие числа не входят в состав стандартной библиотеки макета базового языка (при этом для целей демонстрации достаточно использовать преобразование из натуральных чисел);

- операции сравнения были реализованы как пара из типа — отношения сравнения и функции разрешения такого отношения;
- при этом в качестве отношения равенства были использованы типы идентичности, применение которых не повышает сложность модели в связи с предложенной автором расширенной интерпретации таких типов;
- из числа операций сравнения была реализована только одна — операция сравнения «меньше», т.к. прочие операции могут быть выражены с помощью функций разрешения отношений равенства и отношения «меньше» (эквивалентность такого выражения может быть показана для оригинальной модели).

***Определение* 4.3.** *Интерфейс арифметики на числах с плавающей точкой определяется по следующей формуле:*

$$
\begin{aligned}
\text{ApproxArith} \quad & \equiv \quad \Sigma \\
\quad \text{t} \quad & : \quad \textbf{Type} \\
\quad \text{sum, diff, mult, div} \quad & : \quad \Pi \text{t.t.t} \\
\quad \text{sqrt} \quad & : \quad \Pi \text{t.t} \\
\quad \text{from-natural} \quad & : \quad \Pi \mathbb{N}.\text{t} \\
\quad \text{decide-eq} \quad & : \quad \Pi\left(\text{x y : t}\right).\text{adt} \left\{\text{True : x = y}\right\} \left\{\text{False : } \Pi\left(\text{x=y}\right).\#0\right\} \\
\quad \text{lt} \quad & : \quad \Pi \text{t.t.}\textbf{Type} \\
\quad \text{decide-lt} \quad & : \quad \Pi\left(\text{x y : t}\right).\ \text{adt} \left\{\text{True : lt} \cdot \text{x} \cdot \text{y}\right\} \\
& \qquad \left\{\text{False : } \Pi\left(\text{lt} \cdot \text{x} \cdot \text{y}\right).\#0\right\}
\end{aligned}
$$

4.3.3. Стек преобразователей монад динамической семантики

В рамках общего подхода к описанию динамической семантики на основе монад, семантика описывается в терминах операций над некоторым стеком преобразователей монад.

***Определение* 4.4.** *Стек преобразователей монад динамической семантики подмножества управляемого кода стандарта ECMA-335 задаётся следующей формулой:*

$$\text{ECMA 335}_{\text{method}} \quad \equiv \quad \lambda \left(\text{m : Monad}\right) .$$

$$
\begin{aligned}
& \mathrm{ErrorT} \cdot \mathrm{Exception} \cdot \\
& \big(\mathrm{StateT} \cdot \mathrm{GlobalState} \cdot \\
& \big(\mathrm{StateT} \cdot \big(\mathrm{List} \cdot \mathrm{FunctionState}\big) \cdot \\
& \big(\mathrm{StateT} \cdot \mathrm{FunctionState} \cdot \\
& \ \mathrm{m}\ \big)\big)\big)\big) \\
\mathrm{ECMA\ 335}_{\mathrm{dynamic}} \quad \equiv \quad & \lambda\ \big(\mathrm{m} : \mathrm{Monad}\big)\ . \\
& \mathrm{ReaderT}\ \big(\mathrm{Map} \cdot \mathrm{Method}_{\mathrm{ref}} \cdot \big(\mathrm{ECMA\ 335}_{\mathrm{method}} \cdot \mathrm{m}\big)\big) \cdot \\
& \big(\mathrm{ECMA\ 335}_{\mathrm{method}} \cdot \mathrm{m}\big) \qquad .
\end{aligned}
$$

На вершине стека находится преобразователь монад исключений с типом исключений Exception. В рамках динамической семантики подмножества управляемого кода, используется упрощённая семантика исключений, а именно: в качестве типа исключений рассматривается конечный тип #1. Элемент 0#1 этого типа задаёт единственное исключение, которое может возникнуть в подмножестве управляемого кода — `System.NullReferenceException`. Стек вызовов при этом не хранится в типе исключения, однако в целом используемый стек преобразователей монад допускает выделение такой информации. Далее используется запись $\mathrm{ECMA\ 335}_{\mathrm{dynamic}} \cdot \mathrm{m} \cdot \#1$ для обозначения типа, полученного применением к #1 конструктора типов монады $\mathrm{ECMA\ 335}_{\mathrm{dynamic}} \cdot \mathrm{m}$.

Для описания глобального состояния GlobalState, изменяемого методом, ограничимся рассмотрением кучи, содержащей конечный, известный статически набор одномерных массивов (векторов) типа `float64`, длина которых известна до момента выполнения программы, и набора статических переменных типа float64. Динамическая семантика рассматриваемого фрагмента кода (и кода CMS в целом) может быть описана в рамках этих ограничений, т.к. код генерируется по конфигурации теплогидравлической системы и работает с набором данных, представленных в форме массивов известной длины. При этом элементы типа `float64` в рамках динамической семантики описываются типом fpnum-double, соответствующим модели чисел с плавающей точкой двойной точности, описанной в предыдущем подразделе.

Для такого глобального состояния определим вспомогательные функции:

- getStatic, setStatic — функции получения и установки значения статической переменной в глобальном состоянии;
- getArrayElement, setArrayElement — функции получения и установки значения элемента одного из массивов глобального состояния.

Следующий за глобальным состоянием преобразователь монад состояний определяет стек вызовов, которых хранит локальное состояние исполняемых функций. В нижней части стека находится преобразователь монад состояний, сохраняющий локальное состояние функции, которая исполняется в настоящий момент. Локальное состояние FunctionState хранит значения аргументов, локальных переменных и состояние стека исполнения. Для локального состояния определим вспомогательные функции:

- getArgument, getLocal, setArgument, setLocal — функции получения и установки значений аргументов и локальных переменных в локальном состоянии функции;
- push, pop — функции работы со стеком исполнения.

Для стека вызовов вводятся две вспомогательные функции save, load, которые предназначены для загрузки и сохранения состояния функции на стек.

В последующих разделах все вспомогательные функции считаются заданными на стеке преобразователей монад. Это может быть сделано с использованием операций повышения преобразователей монад, находящихся выше в стеке, чем преобразователь, на котором определена вспомогательная функция.

4.3.4. Динамическая семантика инструкций

Динамическая семантика тела метода для подмножества управляемого кода, представленная в настоящем подразделе, охватывает следующие аспекты спецификации модели исполнения промежуточного кода:

- статические поля уровня модуля и их чтение и запись с помощью инструкций `ldsfld`, `stsfld`;
- вызов статических методов стандартной библиотеки `call`, рекурсия при этом не поддерживается;

- управляемые указатели и запись значений по таким указателям с помощью инструкции `stind`;
- операции чтения и записи локальных переменных и аргументов (инструкции `ldarg`, `starg`, `ldloc`, `stloc`);
- арифметические операции для чисел с плавающей точкой (инструкции `add`, `sub`, `mul`, `div`);
- операции сравнения (инструкции `cle`, `cge`, `clt`, `cgt` с суффиксом `.un` — без проверки на переполнение), в терминах которых определена семантика инструкций условного перехода;
- операция дублирования значения на вершине стека исполнения `dup`.

Семантика инструкций чтения и записи статических полей описывается следующими формулами:

$$
\begin{aligned}
\text{ldsfld, stsfld} \quad &: \quad \Pi\left(\text{m} : \text{Monad}\right)\left(\text{f} : \text{Field}_{\text{ref}}\right).\ \text{ECMA 335}_{\text{dynamic}} \cdot \text{m} \cdot \#1 \\
\text{ldsfld} \quad &\equiv \quad \lambda\ \text{m}\ .\ \text{f}\ .\ \textbf{bind}\ \left(\text{getStatic f}\right)\ \text{push} \\
\text{stsfld} \quad &\equiv \quad \lambda\ \text{m}\ .\ \text{f}\ .\ \textbf{bind}\ \text{pop}\ \left(\text{setStatic f}\right) \qquad .
\end{aligned}
$$

Семантика операций чтения и записи локальных переменных и аргументов представляется аналогично с тем отличием, что вместо ссылки на поле $\textbf{Field}_{\text{ref}}$ используется номер локальной переменной или аргумента, соответственно, а вместо функций getStatic, setStatic используются аналогичные функции для FunctionState в нижней части стека монад. Кроме того, если тип управляемых указателей интерпретировать в семантике подмножества управляемого кода как пару из идентификатора массива и номера элемента в массиве, аналогичным образом с использованием функций getArrayElement и setArrayElement описывается и семантика инструкций чтения и записи по указателю `ldind` и `stind`.

Для операции дублирования `dup` семантика задаётся тривиально в виде композиции снятия значения со стека и двойной установки такого значения:

$$
\begin{aligned}
\text{dup} \quad &: \quad \Pi\left(\text{m} : \text{Monad}\right).\ \text{ECMA 335}_{\text{dynamic}} \cdot \text{m} \cdot \#1 \\
\text{dup} \quad &\equiv \quad \lambda\ \text{m}\ .\ \textbf{bind}\ \text{pop}\ \left(\lambda \text{x}.\ \textbf{bind}\ \left(\text{push x}\right)\ \left(\lambda\ \text{y}.\ \text{push x}\right)\right) \qquad .
\end{aligned}
$$

Арифметические операции над числами с плавающей точкой в динамической семантке описываются в терминах упомянутой ранее модели с плавающей точкой, например, для инструкции сложения:

$$\mathrm{add}_{r_8} \;:\; \Pi\,\big(m : \mathrm{Monad}\big)\,.\,\mathrm{ECMA\ 335}_{\mathrm{dynamic}} \cdot m \cdot \#1$$

$$\mathrm{add}_{r_8} \;\equiv\; \lambda\, m\,.\,\mathbf{bind}\ \mathrm{pop}\ \big(\lambda x\,.\,\mathbf{bind}\ \mathrm{pop}\ \big(\lambda\, y\,.\,\mathrm{push} \cdot \big(\mathrm{addFp} \cdot x \cdot y\big)\big)\big) \quad .$$

Аналогичным образом реализованы операции сравнения, на примере инструкции `clt` (в этой формуле считается, что целые числа моделируются типом #2)

$$\mathrm{clt}_{r_8} \;:\; \Pi\,\big(m : \mathrm{Monad}\big)\,.\,\mathrm{ECMA\ 335}_{\mathrm{dynamic}} \cdot m \cdot \#1$$

$$\mathrm{clt}_{r_8} \;\equiv\; \lambda\, m\,.\,\mathbf{bind}\ \mathrm{pop}\ \big(\lambda x\,.\,\mathbf{bind}\ \mathrm{pop}\ \big(\lambda\, y\,.\,\mathrm{push} \cdot \mathbf{fst} \cdot \big(\mathrm{decide\text{-}lt} \cdot y \cdot x\big)\big)\big) \quad .$$

Следует отметить, что семантика целых чисел в примере не рассматривается. Использование целых чисел в целевом фрагменте кода сводится к анализу результатов сравнения. Поэтому вместо целых чисел используется конечный тип #2 из двух элементов.

В отличие от перечисленных выше инструкций, семантика инструкции `call` нетривиальна, поскольку требует получения произвольного, заданного в сигнатуре функции, количества аргументов на стеке. Для получения аргументов используется итерация функции **bind**. Затем выполняется сохранение состояния вызывающей функции, после чего создаётся новое исходное состояние вызова, после чего выполняется целевая функция. В заключение, после завершения выполнения функции, если она возвращает значение, такое значение устанавливается на вершину стека вычислений для новой машины. Функция $\mathrm{ECMA\ 335}_{\mathrm{value}}$ описывает все допустимые значения данных в памяти. Таких значений на настоящее время поддерживается три типа: числа с плавающей точкой; целые числа (только для представления булевского типа); управляемые указатели на элемент массива на куче. Описанная схема реализована следующей формулой:

$$\begin{aligned}
\mathrm{getArgs} \;&:\; \Pi\,\big(m : \mathrm{Monad}\big)\,\big(\mathrm{numArgs} : \mathbb{N}\big)\,.\,\mathrm{ECMA\ 335}_{\mathrm{dynamic}} \cdot m \cdot \\
&\qquad \big(\mathrm{List\ ECMA\ 335}_{\mathrm{value}}\big) \\
\mathrm{getArgs} \;&\equiv\; \lambda\, m\ \mathrm{numArgs}\,.\,\mathbf{iter}_{\mathbb{N}}\ \big(\mathrm{numArgs},\ \mathrm{ECMA\ 335}_{\mathrm{dynamic}} \cdot m \cdot \\
&\qquad \big(\mathrm{List\ ECMA\ 335}_{\mathrm{value}}\big),
\end{aligned}$$

$$
\begin{aligned}
& \quad \textbf{return}\ \text{empty}, \\
& \quad \lambda\ (\text{n} : \mathbb{N})\ (\text{l} : \text{List ECMA } 335_{\text{value}})\ .\ \textbf{bind}\ \text{pop}\ (\text{cons} \cdot \text{l})) \\
\text{prepareCall} \quad & \equiv \quad \lambda\ \text{m r args}.\ \textbf{bind}\ (\text{mkInitialState} \cdot \text{r} \cdot \text{args})\ \text{setRunningState} \\
\text{checkReturn} \quad & \equiv \quad \lambda\ \text{m r}\ .\ (\textbf{match}\ (\text{hasReturn} \cdot \text{r}) \\
& \quad [\text{False load}] \\
& \quad [\text{True} \quad (\textbf{bind}\ \text{pop}\ (\lambda\ \text{val}.\ \textbf{bind}\ \text{load}\ (\lambda\ _\ .\ \text{push} \cdot \text{val})))]) \\
\text{call} \quad & : \quad \Pi\ (\text{m} : \text{Monad})\ (\text{r} : \text{Method}_{\text{ref}})\ .\ \text{ECMA } 335_{\text{dynamic}} \cdot \text{m} \\
\text{call} \quad & \equiv \quad \lambda\ \text{m r}\ .\ \textbf{bind}\ (\text{getArgs} \cdot \text{m} \cdot \text{r})\ (\lambda\ \text{args}. \\
& \quad \textbf{bind}\ \text{save}\ (\lambda\ _\ . \\
& \quad \textbf{bind}\ (\text{prepareCall} \cdot \text{m} \cdot \text{r} \cdot \text{args})\ (\lambda\ _\ . \\
& \quad \textbf{bind}\ \text{getEnv}\ (\lambda\ \text{sem}. \\
& \quad \textbf{bind}\ \text{sem checkReturn}))))\ .
\end{aligned}
$$

Таким образом, описана семантика всех инструкций, требуемых для описания семантики фрагмента кода.

4.3.5. Формальная семантика фрагмента кода

Заключительным шагом при описании динамической семантики фрагмента кода является определение функции динамической семантики, которая элементам поддерживаемого подмножества статической семантики ставит в соответствие их динамическую семантику.

В настоящей работе, поскольку динамическая семантика описывается в целях демонстрации работоспособности предложенных решений, семантическая функция определяется для тела метода в некотором окружении статической семантики (содержащем объявления стандартных типов) и динамической семантики (содержащем семантику стандартных функций, в случае рассматриваемого участка кода — функции `Math.Sqrt`).

Семантическая функция для тела метода использует семантику типизации: базовые блоки транслируются раздельно, после чего объекты, описывающие семан-

тику базовых блоков, объединяются в объект, описывающий семантику метода с использованием графа базовых блоков и функции **bind**.

Фрагмент кода CMS моделирования теплогидравлических процессов в первом и втором контурах атомных электростанций, используемый для демонстрации приведён в приложении E. Первоначальный вариант кода написан на языке C. Функция `ysparg` принимает шесть входных параметров и записывает значения трёх выходных по указателям. С использованием компилятора LCC, расширенного генератором кода [59], соответствующего стандарту ECMA-335, код был скомпилирован в управляемый код. После этого код был отредактирован вручную с целью снижения количества используемых инструкций, а также для замены неуправляемых указателей в аргументах функции на управляемые указатели. Верификация изменений была проведена путём выполнения двух фрагментов кода на заданных входных данных, в том числе — на представленных в таблице C.2. Было получено, что исходный код на языке C и код, полученный с помощью транслятора LCC, дают отличающиеся результаты. Это объясняется тем, что, согласно стандарту, на стеке исполнения находятся машинные числа с плавающей точкой. Для архитектуры центрального процессора x86 машинные числа с плавающей точкой имеют размер 80 бит, тогда как параметры компиляции кода на языке C принудительно устанавливают двойную точность вычислений (размер — 64 бита). В то же время, внесение ручных правок в код на результат не повлияло.

Инструкции управляемого кода были переписаны вручную в виде объекта статической семантики. Для такого объекта с использованием семантической функции была получена динамическая семантика фрагмента кода. Сравнение результатов, полученных с помощью нормализации динамической семантики исследуемого фрагмента кода и результатов, полученных в среде Coq для того же фрагмента кода, адаптированного вручную, показало, что обе модели завершают вычисления с идентичным результатом.

4.4. Выводы по четвёртой главе

Методы описания формальной семантики на основе стека преобразователей монад можно отнести к гибриду операционной и денотационной семантики. С одной стороны, семантика представляется как функция в терминах λ-исчисления с зависимыми типами, т.е. обладает свойствами денотационной семантики. С другой стороны, само описание семантики имеет характер операционного, поскольку стек монад позволяет построить модель используемой в спецификации языка абстрактной машины. Такой гибридный характер подхода с использованием монад продемонстрирован в разделе 4.3 при построении динамической семантики подмножества стандарта ECMA-335.

В целом, применительно к цели исследования, по результатам, полученным в настоящей главе могут быть сделаны следующие выводы:

1. результаты разработки статической семантики стандарта ECMA-335 показывают, что макеты языка и программного средства могут быть использованы на практике для описания статической семантики существующих востребованных языков программирования;
2. результаты описания динамической семантики фрагмента кода и сравнения результатов вычислений с результатами, полученными в среде Coq позволяют утверждать, что для данного частного случая была подтверждена корректность макетов языка и программного средства как реализации λ-исчисления с зависимыми типами относительно реализации среды Coq.

Глава 5

Модель динамического параллельного исполнения программ

Одной из основных тенденций современного этапа развития вычислительных средств и систем высокой производительности является увеличение количества параллельно обрабатываемых потоков вычислений. При разработке или модернизации существующего программного обеспечения эту тенденцию необходимо учитывать. В этой связи актуальным является процесс адаптации программ для их эффективного использования в вычислительной среде параллельной архитектуры.

Подходы к параллельному исполнению программ делят на статический и динамический. При статическом параллельном исполнении различные вычислительные потоки программы распределяются по вычислительным устройствам до запуска программы. При динамическом параллельном исполнении решение породить новый вычислительный поток принимается в процессе выполнения программы. В процессе адаптации программы для параллельного исполнения важно убедиться, что определенные свойства программы, в первую очередь — результаты вычислений на отдельных этапах, сохраняются. Одной из гарантий выполнения этих и ряда других функциональных требований, предъявляемых к программе, является возможность ее верификации с использованием модели, описывающей семантику программы.

В настоящей главе представлена базовая модель динамического параллельного исполнения программ, функции которых не имеют побочных эффектов, для случая, когда взаимодействие потоков вычислений может осуществляться путем порождения нового потока, который вычисляет значение некоторой функции программы, или ожидания результата вычислений от некоторого потока, порожденного ранее. Примерами реализации подобного подхода являются библиотека MultiLisp [44]

и система российской разработки NewTS [177]. Под функциями, не имеющими побочных эффектов, в настоящей работе понимаются функции, возвращаемое значение которых зависит только от входных аргументов, и которые в процессе вычислений не изменяют значение глобальных переменных.

Для изучения свойств параллельного исполнения программ представляется целесообразным, прежде всего, абстрагироваться от вычислений, которые выполняются в программе, и исследовать свойства программы исключительно с позиций порядка исполнения отдельных ее блоков. С этой целью в первом разделе настоящей главы определяется язык управления потоком исполнения.

В настоящей главе модель динамического квазипараллельного исполнения программы представлена с использованием подхода к ее описанию на основе денотационной семантики [150] с тем отличием, что в качестве денотаций используются термы представленной в главах 2, 3 разновидности лямбда-исчисления с зависимыми типами. Для модели доказан набор утверждений, гарантирующих корректность исполнения программы с точки зрения результата ее исполнения. Подобные утверждения были доказаны для MultiLisp и NewTS с использованием операционной семантики. Однако денотационная семантика позволила получить такие результаты в более естественном и простом для интерпретации виде.

Представленная далее модель динамического параллельного исполнения программ описана со следующими ограничениями.

1. Считается определенной динамическая семантика некоторых блоков кода программы.
2. Отсутствует глобальное изменяемое состояние. Это ограничение означает, что результат выполнения любой функции программы зависит только от входных данных.
3. Функция, являющаяся точкой входа (entry point) программы, возвращает некоторое значение, которое считается результатом выполнения программы.
4. Взаимодействие вычислительных потоков программы производится с помощью системных вызовов двух видов: spawn — создание нового вычислительного потока,

точкой входа которого является указанная функция; wait — ожидание завершения потока с определенным идентификатором и получение возвращенного потоком значения.

5. Обработку системных вызовов и планирование выполнения потоков осуществляет внешняя по отношению к программе система. Модель обработки системных вызовов и планирования потоков представлена в настоящей работе.
6. Определены два специальных режима исполнения программы, которые далее называются «режим исполнения по порядку» (in-order evaluation) и «режим исполнения по требованию» (lazy evaluation). При исполнении программы в этих режимах считается, что одновременно производить вычисление может только один поток.
7. Под режимом исполнения программы по порядку понимается такой способ планирования потоков вычислений, при котором после системного вызова spawn выполнение вызывающего потока приостанавливается до того времени, когда будет завершен поток, созданный в результате обработки системного вызова. Такой режим считается эквивалентным последовательному исполнению программы, при котором системный вызов spawn работает как вызов функции программы, а системный вызов wait получает значение, возвращенное функцией. Эквивалентность может быть определена и проверена в случае, когда определена динамическая семантика программы.
8. Под режимом исполнения программы по требованию понимается такой способ планирования потоков, при котором до первого системного вызова wait выполняется поток, соответствующий точке входа программы (далее — «главный поток»), а затем управление переходит к потоку, завершения которого ожидает главный поток. В результате каждого системного вызова wait управление переходит к тому потоку, завершения которого ожидает предыдущий. Таким образом, производятся только те вычисления, которые непосредственно необходимы для получения результата программы.

Такие ограничения относительно адекватно описывают семантику процессов параллельного исполнения программ без побочных эффектов. В частности, пунктом

2 допускается использование общих блоков памяти при условии, что они используются только для чтения. Добавление других способов взаимодействия потоков вычислений является одним из направлений дальнейших исследований.

5.1. Язык управления потоком исполнения

Язык управления потоком исполнения (управляющий язык) представляет собой императивный язык программирования, основное предназначение которого — описание порядка исполнения команд произвольного языка программирования. Прежде всего, введем параметры, с помощью которых достигается абстракция от деталей реализации императивного языка.

Тип **Cf** определяет статическую семантику выражений расширяемого языка программирования. Такие выражения используются для абстракции производимых программой вычислений, а также для задания исходного локального состояния функции при вызове или создании потока. Тип **Mf** определяет функции, выполняющие слияние локального состояния выполняемой функции с результатом вызова некоторой другой функции. Тип **Bf** определяет функции, возвращающие булевское значение. В зависимости от результата выполнения таких функций, может быть выполнена та или иная ветвь программы. Тип **Tf** определяет функции, которые содержат команды языка управления потоком исполнения. Кроме того, пусть S — состояние исполнения функций исходного языка. Если ограничить рассмотрение конечными наборами функций каждого из перечисленных типов, что является адекватным ограничением для существующих программ, типы **Cf**, **Mf**, **Bf**, **Tf** можно без ограничения общности принять за некоторые конечные типы #k. В этом случае окружения динамической семантики расширяемого императивного языка программирования можно определить в виде функций, имеющих перечисленные далее типы:

$$\mathrm{Ce} \;:\; \prod\big(\mathrm{id} : \mathrm{Cf}\big)\;.\;\big(\mathrm{initial} : \mathrm{S}\big)\;.\;\mathrm{S}$$

$$\mathrm{Me} \;:\; \prod\big(\mathrm{id} : \mathrm{Mf}\big)\;.\;\big(\mathrm{trunk\ branch} : \mathrm{S}\big)\;.\;\mathrm{S}$$

$$\mathrm{Be} \;:\; \prod\big(\mathrm{id} : \mathrm{Bf}\big)\;.\;\big(\mathrm{state} : \mathrm{S}\big)\;.\;\#2$$

Перечислим параметры, отражающие специфику модели динамического параллельного исполнения. Тип **TId** определяет тип идентификаторов потоков, порождаемых при динамическом параллельном исполнении; в рамках настоящей работы он считается равным типу натуральных чисел . Тип **Var** определяет набор идентификаторов переменных, в которых хранятся идентификаторы потоков.

Для упрощения изложения предположим, что идентификаторы потоков неизменяемы. Кроме того, предположим, что потоки не имеют информации о собственном идентификаторе. В рамках настоящей модели единственное возможное использование такой информации заключалось бы в возможности блокировки потока путём вызова функции ожидания завершения самого себя.

***Определение* 5.1.** *Набор команд языка управления потоком исполнения задаётся следующей формулой:*

$$
\begin{aligned}
\mathrm{St} \;\equiv\; \mathrm{Ind}\Big(\lambda\big(\mathrm{St} : \mathbf{Type}\big).\ \mathrm{adt}\ \ & \big\{\ \mathrm{skip} : \#1\ \big\} \\
& \big\{\ \mathrm{comp} : \mathrm{Cf}\ \big\} \\
& \big\{\ \mathrm{seq} : \Sigma\big(s_1 : \mathrm{St}\big).\big(s_2 : \mathrm{St}\big)\ \big\} \\
& \big\{\ \text{if-then-else} : \Sigma\big(b : \mathrm{Bf}\big).\big(s_t : \mathrm{St}\big).\big(s_f : \mathrm{St}\big)\ \big\} \\
& \big\{\ \text{while-do} : \Sigma\big(b : \mathrm{Bf}\big).\big(s : \mathrm{St}\big)\ \big\} \\
& \big\{\ \mathrm{spawn} : \Sigma\big(t : \mathrm{Tf}\big).\big(c : \mathrm{Cf}\big).\big(v : \mathrm{Var}\big)\ \big\} \\
& \big\{\ \mathrm{wait} : \Sigma\big(v : \mathrm{Var}\big).\big(m : \mathrm{Mf}\big)\ \big\} \\
\Big)\quad .
\end{aligned}
$$

1. Команда skip передает управление далее, не изменяя локальное состояние.
2. Для любой функции c из Cf, команда $\mathrm{comp} \cdot c$ выполняет вычисление, которое задается функцией c, и передает управление далее, изменив локальное состояние.
3. Для любых команд $s1$ и $s2$ команда $\mathrm{seq} \cdot \mathbf{pair}\big(s_1\ ,\ s_2\ \big)$ выполняет последовательно команды $s1$ и $s2$.
4. Для любой функции b из Bf и для любых команд s_t, s_f, команда $\text{if-then-else} \cdot \mathbf{tuple}\big(b,\ s_t\ ,\ s_f\ \big)$ вычисляет значение условия b. В случае истинности условия, управление передается команде s_t. В случае ложности условия, управление передается команде s_f.

5. Для любой функции b из Bf, и для любой команды s команда $\text{while-do} \cdot \mathbf{pair}(b, s)$ представляет собой цикл с условием b и телом s.
6. Для любого идентификатора переменной v из Var, для любой функции t из Tf, для любой функции c из Cf, команда $\text{spawn} \cdot \mathbf{tuple}(t, c, v)$ создаёт новый поток, выполняющий функцию t с начальным состоянием, задаваемым c, и сохраняет идентификатор потока в переменной v.
7. Для любой переменной v из Var и для любой функции m из Mf, команда $\text{wait} \cdot \mathbf{pair}(v, m)$ приостанавливает выполнение текущего потока до завершения потока v. После завершения потока v выполняется слияние локальных состояний выполняемой функции и потока с помощью m.

Отметим, что корректность использования тех или иных функций гарантируется статической семантикой используемого для их описания языка. Функции слияния в простейшем случае производят сохранение результата вычислений в локальной переменной. К языку управления потоком исполнения может быть добавлена команда синхронного вызова call. Однако ожидаемая семантика этой команды эквивалентна последовательному выполнению команд spawn и wait без учёта установки значения переменной-идентификатора потока. По этой причине команда call не вошла в минимальный список команд языка.

***Определение* 5.2.** *Статическое окружение функций языка управления потоком исполнения Te и статическая семантика программы на языке управления потоком исполнения $Prog$ задаются следующими формулами:*

$$\begin{aligned} Te &\equiv \Pi\ (name : Tf)\ .\ St \\ Prog &\equiv \Sigma\ (C : Ce)\ .\ (M : Me)\ .\ (B : Be)\ .\ (T : Te)\ .\ (entry : Tf) \end{aligned}$$

Статическая семантика программы в дополнение к окружениям семантики функций содержит информацию о точке входа entry в программу.

5.2. Монада возобновлений и реактивный параллелизм

Основные положения подхода на основе монад к описанию динамической семантики языков программирования были представлены в предыдущей главе в разделе 4.3. В настоящем разделе описаны дополнительные монады, необходимые для определения модели динамического параллельного исполнения.

Для описания денотационной семантики процесса параллельного исполнения в настоящей работе используется монада возобновлений (resumption monad). Монада возобновлений и соответствующий ей преобразователь монад над категорией полных частично упорядоченных множеств были формализованы N. S. Papaspyrou в работе [98]. Основное предназначение монады возобновлений — определение семантики выражений с недетерминированным порядком вычисления. Исходное определение конструктора типов монады основывается на рекурсивном уравнении доменов в рамках теории доменов:

$$\mathsf{ResT}(M)(D) \cong D + M(\mathsf{ResT}(M)(D)),$$

где M — монада, с помощью которой представляются неделимые вычисления, которые выполняются за один шаг, а D — домен результатов выполнения программы.

Тип, получаемый с помощью конструктора монады возобновлений, представляет собой либо результат вычисления, либо некоторое вычисление, которое можно считать неделимым. Результатом последнего вновь является элемент типа возобновлений. Семантика программы с помощью преобразователя монад возобновлений выражается в виде последовательности шагов вычислений, которые либо завершаются с некоторым результатом, либо требуют продолжения вычислений. Недетерминированный порядок вычислений выражается с помощью использования в на нижнем уровне стека монад M монады многовариантности (монада подмножеств, powerdomain), так, что на каждом шаге вычисление может возвращать множество всех возможных следующих шагов в случае, если следующий

шаг недетерминирован. Таким образом, с помощью комбинации преобразователя монад возобновлений и монады недетерминированности появляется возможность выразить параллельные вычисления с необходимой степенью детализации в виде высокоуровневого денотационного описания.

Монада многовариантности в терминах базового языка может быть представлена только для типов с разрешимым равенством. С таким ограничением конструктор типа монады представляет собой непустой список всех возможных результатов вычисления; оператор **return** возвращает список с единственным элементом, а оператор **bind** применяется для всех элементов списка с последующим слиянием списка результатов и удалением идентичных элементов.

В системах на основе λ-исчисления с зависимыми типами для отражения потенциальной незавершимости монада возобновлений и аналогичные ей конструкции, как правило, рассматриваются в виде коиндуктивного типа [38]. Схема представления коиндуктивных типов в виде последовательности конечных приближений позволяет описать преобразователь монад возобновлений в виде, идентичном представленному выше уравнению на доменах. Существующми системами, такими как Coq, подобное определение не допускается (листинг 8), как и в случае индуктивных типов.

```
CoInductive ResT (m : Type -> Type) (a : Type) :=
|  done (x : a)
|  step (s : m (ResT m a)).

> Error: Non strictly positive occurrence of "ResT" in
>  "m (ResT m a) -> ResT m a".
```

Листинг 8: Ошибка, выводимая Coq
при определении преобразователя монад возобновлений в виде уравнения доменов

В случае, когда аргумент m равен тривиальной монаде Id, преобразователь монад возобновлений в коиндуктивной форме вырождается в монаду частичности (Partial), которая отражает результат вычисления, которое может не завершиться. С помощью анаморфизма может быть задана функция run, которая преобразует преобразователь монад возобновлений в монаду частичности:

$\mathrm{run} \;:\; \Pi\left(\mathrm{ResT}\cdot \mathrm{m}\cdot \mathrm{a}\right).\,\mathrm{Partial}\cdot\left(\mathrm{m}\cdot \mathrm{a}\right)$.

В своем базовом виде монада возобновлений, как было отмечено ранее, выражает случай недетерминированного порядка вычислений. Однако для определения семантики динамического параллельного исполнения, при котором потоки могут создаваться в процессе работы программы, этого недостаточно. Для формального описания процессов планирования в операционных системах У. Л. Харрисоном в [60] была предложена модель так называемого реактивного параллелизма (reactive concurrency) на основе модифицированной монады возобновлений. Смысл этого подхода заключается в том, что исполнение программы рассматривается как постоянный обмен сообщениями (системными вызовами) между вычислительными потоками программы и операционной системой. Обмен системными вызовами рассматривается как пара «запрос программы — ответ операционной системы». Простейшая такая пара $(\mathrm{Cont}, \mathrm{Ack})$ используется для представления неделимого шага вычислений. Другие пары связаны с разного рода взаимодействием между процессами. Реактивная форма преобразователя монад возобновлений, которая используется для представления такой динамической семантики, соответствует следующему уравнению на доменах:

$$\begin{aligned}\mathsf{ReactT}(M)(D) \cong D + \\ +(Req \times (Rsp \to M(\mathsf{ReactT}(M)(D)))),\end{aligned}$$

где Req — множество системных вызовов; Rsp — множество ответов системы; M и D, как и в базовой форме монады возобновлений представляют собой, соответственно, исходную монаду, с помощью которой отображаются неделимые вычисления, и домен результатов выполнения программы.

Для монады возобновлений в реактивной форме определяется функция signal, с помощью которой могут быть выражены системные вызовы. С помощью преобразователя монад возобновлений в следующем разделе определяется модель динамического параллельного исполнения программ, записанных на языке управления потоком исполнения.

5.3. Модель динамического параллельного исполнения

Базовая модель динамического параллельного исполнения определяется следующим образом. Сначала в терминах преобразователя монад возобновлений в реактивной форме описывается динамическая семантика языка управления потоком исполнения. Динамическая семантика в соответствие каждой функции программы ставит последовательность системных вызовов и вычислений. Затем определяется модель ядра системы динамического параллельного исполнения. Основные функции ядра — это управление потоками вычислений и обработка системных вызовов.

Следует отметить, что приведённые в тексте работы формулы изменены для удобства чтения и отражают общие подходы к построению; при проверке на вход программному средству подавались формулы, отличные от приводимых.

5.3.1. Динамическая семантика языка управления потоком исполнения

Динамическая денотационная семантика языка управления потоком исполнения каждой программе ставит в соответствие функцию, которая представляет значение программы во время исполнения. Динамическая семантика определяется с помощью модульного подхода с системными вызовами, реализованными с помощью преобразователя монад возобновлений в реактивной форме.

Прежде всего, определим домен состояния переменных, в которых хранятся идентификаторы потоков:

$$V = Map \cdot Var \cdot TId.$$

Затем определим множество допустимых запросов и ответов, которые требуются для определения преобразователя монад возобновлений в реактивной форме.

***Определение* 5.3.** *Типы допустимых запросов (Req) и ответов (Rsp) потоков в модели динамического параллельного исполнения программ определяются следующей формулой:*

$$\mathrm{Req} \equiv \mathrm{adt}\left\{ \mathrm{Cont} : \#1 \right\} \left\{ \mathrm{SpawnReq} : \Sigma \left(\mathrm{id} : \mathrm{Tf}\right) \left(\mathrm{st} : \mathrm{S}\right) \right\} \left\{ \mathrm{WaitReq} : \mathrm{TId} \right\}$$

$$\mathrm{Rsp} \equiv \lambda \left(\mathrm{kind} : \#3\right) . \mathbf{case}_3 \left(\mathrm{kind}, \mathbf{Type}, \#1, \mathrm{TId}, \mathrm{S}\right)$$

Для того, чтобы типы ожидаемых ответов на запросы совпадали с типами запросов, определение монады возобновлений в реактивной форме модифицируется следующим образом: тип Rsp зависит от типа запроса **fst** $(r : \mathrm{Req})$.

В настоящей модели допустимы три перечисленных далее вида системных вызовов.

1. Тривиальный системный вызов Cont представляет следующий шаг вычислений. В качестве реакции системы ожидается ответ Ack, который означает, что программа может продолжить выполнение.
2. Системный вызов SpawnReq представляет запрос на порождение нового потока вычислений. Точкой входа потока является некоторая функция с идентификатором id с исходным локальным состоянием st.
3. Системный вызов WaitReq представляет запрос на ожидание результата от потока вычислений с идентификатором типа Var.

***Определение* 5.4.** *Стек преобразователей монад динамической семантики управляющего языка определяется следующей формулой:*

$$\begin{aligned} \mathrm{CFL} \quad &= \quad \lambda \left(m : \mathrm{Monad}\right) \\ &\quad \mathrm{ReaderT} \cdot \mathrm{Prog} \cdot \\ &\quad \left(\mathrm{StateT} \cdot \mathrm{V} \cdot \right. \\ &\quad \left(\mathrm{StateT} \cdot \mathrm{S} \cdot \right. \\ &\quad \left.\left. m\right)\right) \\ \mathrm{CFL}_{\mathrm{dynamic}} \quad &\equiv \quad \mathrm{ReactT} \cdot \mathrm{Req} \cdot \mathrm{Rsp} \cdot \left(\mathrm{CFL} \cdot \mathrm{Id}\right) \qquad . \end{aligned}$$

Определение стека преобразователей монад разделено на две части. Первая формула CFL определяет общую форму стека, которая используется как в настоящем подразделе, так и в следующем, с различными уточняющими параметрами. Вторая формула $\mathrm{CFL}_{\mathrm{dynamic}}$ уточняет определение стека для решения задач настоящего подраздела. На верхнем уровне стека находится монада возобновлений в реактивной форме ReactT. С ее помощью, как уже было отмечено ранее, реализована передача системных вызовов на уровень модели системы. На следующем уровне

стека находится преобразователь монад окружения ReaderT, с помощью которой задается информация о программе, включая, в частности, динамическое значение функций из Cf, Bf, Mf. Далее находятся преобразователи монад состояния StateT, которые обеспечивают работу с локальным состоянием и со значением переменных, хранящих идентификаторы потоков. Для различения операций над этими двумя состояниями, к именам операций в дальнейшем будет добавляться индекс $_V$ или $_S$. На «дне» стека в данном случае находится тождественная монада Id.

Определим индуктивно динамическую семантику языка управления потоком исполнения (приложение F.1, $CFL_{dynamic}$). Значение команды skip — функция, возвращающая локальное состояние. Значение команды comp — функция, выполняющая изменение состояния в соответствии с динамической семантикой соответствующего модификатора состояния. Значение команды if-then-else — функция, возвращающая, в зависимости от значения условия, значение одной или другой ветви вычислений. Значение команды while-do — рекурсивная функция, заданная в терминах преобразователя монад возобновлений в реактивной форме. Эта функция выполняет вычисление условия и, в случае его ложности, возвращает текущее значение состояния вычислений. В случае истинности условия функция выполняет системный вызов Cont, при ответе на который начинает следующую итерацию. Значение команды seq — композиция функций, соответствующих значению команд s_1 и s_2. Значение команды spawn — функция, которая производит системный вызов SpawnReq и сохраняет полученный от системы идентификатор созданного системой потока в соответствующей локальной переменной. Значение команды wait — функция, которая производит системный вызов WaitReq и выполняет слияние полученного от системы значения с локальным состоянием.

Определенная таким образом динамическая семантика позволяет перейти к описанию модели управляющего ядра системы.

5.3.2. Модель управляющего ядра системы

Модель управляющего ядра системы динамического параллельного исполнения описывает процесс обработки системных вызовов, а также определяет, каким образом система работает с потоками исполнения.

Прежде всего, определим базовые понятия, а именно — состояние потока вычисления и состояние системы.

Определение 5.5. *Состояние потока вычислений* Thread *и состояние системы* System *задаются следующими формулами:*

$$
\begin{aligned}
\mathrm{Thread} \quad &\equiv \quad \Sigma \quad \left(\mathrm{tid\ pid} : \mathrm{TId}\right) . \\
&\qquad\quad \left(\mathrm{name} : \mathrm{Tf}\right) . \\
&\qquad\quad \left(\mathbf{var}_0 : \mathrm{V}\right) . \left(\mathrm{state}_0 : \mathrm{S}\right) . \\
&\qquad\quad \left(\mathrm{step} : \mathbb{N}\right) \\
&\qquad\quad \left(\mathbf{var} : \mathrm{V}\right) . \left(\mathrm{state} : \mathrm{S}\right) . \\
&\qquad\quad \left(\mathrm{semantics} : \mathrm{CFL}_{\mathrm{dynamic}} \cdot \mathrm{S}\ \right) \\
\mathrm{System} \quad &\equiv \quad \Sigma \quad \left(\mathrm{threads} : \mathrm{Map} \cdot \mathrm{TId} \cdot \mathrm{Thread}\right) . \\
&\qquad\quad \left(\mathrm{maxid} : \mathrm{TId}\right)
\end{aligned}
$$

Состояние потока включает в себя:

- идентификатор потока tid;
- идентификатор породившего его потока pid;
- объект, отражающий семантику производимых вычислений $\mathrm{semantics}$;
- исходное и текущее состояние вычислений $\mathbf{var}_0$, state_0, $\mathbf{var}$, state;
- количество шагов (в терминах монады возобновлений), которые были выполнены для данного потока step.

Определим сокращения для функций над System:

getThread, setThread — получают и устанавливают состояние потока по идентификатору (функция $\mathrm{setThread}$ возвращает новый элемент System);

nextId — возвращает пару из нового идентификатора потока и состояния системы с изменённым maxid.

Для любых потоков t1 и t2 отношение $\mathrm{leq}_{\mathrm{Thread}} \cdot t_1 \cdot t2$ означает, что поток t2 является результатом исполнения нескольких шагов потока t1. В этом случае должны совпадать идентификаторы потоков, идентификаторы породивших их потоков и исходные состояния, а количество выполненных шагов t2 должно быть не меньше, чем количество выполненных шагов t1. В этом определении существенным образом используется условие отсутствия побочных эффектов у функций: процесс выполнения зависит только от начального состояния. Состояние системы определяется набором состояний потоков и первым свободным идентификатором потока. На типе состояний системы отношение порядка $\mathrm{leq}_{\mathrm{System}}$ определяется стандартным для произведения типов способом. Таким образом, для любых двух состояний системы s1 и s2, отношение $\mathrm{leq}_{\mathrm{System}} \cdot s_1 \cdot s2$ означает, что состояние s2 является результатом исполнения нескольких шагов некоторых потоков системы s1.

Стек монад, над которым определяется процесс работы системы, соответствует стеку монад динамической семантики языка управления потоком исполнения за исключением того факта, что монада возобновлений присутствует здесь в базовой, а не в реактивной форме. Кроме того, в нижней части стека монад вместо тождественной монады Id используется монада многовариантности P, с помощью которой выражается недетерминированность вычислений:

***Определение* 5.6.** *Стек монад CFL_{system} задаётся следующей формулой:*

$$\mathrm{CFL}_{\mathrm{system}} \equiv \mathrm{ResT} \cdot (\mathrm{CFL} \cdot \mathrm{P})$$

Функция handle обрабатывает системный вызов от некоторого заранее определенного потока (назовем его запланированным) и соответствующим образом изменяет состояние системы. Для её определения используются три вспомогательные функции, перечисленные далее. Вспомогательная функция apply выполняет шаг вычислений путём подстановки известных данных. Вспомогательные функции loadContext и saveContext выполняют загрузку и сохранение контекста исполнения потоков. Последняя функция при этом возвращает изменённый объект потока с инкрементированным счётчиком шагов.

Системный вызов Cont обрабатывается путём загрузки контекста исполнения выбранного потока, выполнения шага вычислений и сохранения контекста исполнения в записи с идентификатором потока. Системный вызов SpawnReq создаёт новую запись потока, увеличивает максимальный идентификатор потока и выполняет шаг вычислений текущего потока аналогично вызову Cont. Системный вызов WaitReq в случае, если поток, от которого ожидается значение завершён, обрабатывается аналогично вызову Cont с помощью подстановки результата. Если поток, от которого ожидается значение не завершён, обработчик системного вызова WaitReq возвращает управление системе. Последний случай исключается предикатом can-handle.

Предикат can-handle определяет, может ли в некотором состоянии системы быть выполнен следующий шаг определенного потока.

В начальном состоянии System0, которое зависит от статической семантики программы и исходного состояния, в системе присутствует только один поток с идентификатором $\mathbb{0}$, соответствующий точке входа программы с исходными параметрами. Функция step-full выполняет недетерминированный шаг вычислений. Если существуют потоки, которые могут быть выполнены, результатом применения этой функции будет список результатов выполнения одного шага вычислений для каждого из доступных для выполнения потоков. В противном случае, состояние системы не изменяется. Функция get-result возвращает результат вычисления точки входа, если в рассматриваемом состоянии системы вычисление было завершено, и None в противном случае.

Семантика модели динамического параллельного исполнения может быть выражена с помощью функции step-full и функции run способом, который аналогичен использованному ранее для описания семантики команды while-do языка управления потоком исполнения:

$$\text{semantics} \quad : \quad \Pi \left(\text{p} : \text{Prog}\right) \left(\text{s} : \text{S}\right) . \text{Partial} \cdot \left(\text{CFL} \cdot \text{P} \cdot \text{S}\right)$$

$$\begin{aligned} \text{semantics} \quad \equiv \quad & \lambda \left(\text{p} : \text{Prog}\right) \left(\text{s} : \text{S}\right) . \text{run} \cdot \mathbf{elN}\Big(\mathbf{tuple}\Big(\text{System}, \text{System}_0 \cdot \text{p} \cdot \text{s}, \\ & \lambda\left(\text{sys} : \text{System}\right) . \left(\text{n} : \mathbb{N}\right) . \\ & \mathbf{iter}_{\mathbb{N}} \left(\text{n}, _, \mathbf{return}\ \text{sys}, \left(\text{k}, \text{p}\right) \mapsto \mathbf{match} \left(\text{semantics} \cdot \left(\text{get}_{\text{Map}} \cdot \left(\text{threads} \cdot \text{p}\right) \cdot 0\right)\right) \\ & \Big[\text{Done} \quad \left(\text{Done} \cdot \text{p}\right)\Big] \end{aligned}$$

$$\left[\text{Step}\quad\left(\text{Step}\cdot\left(\textbf{bind}\ \text{p step-full}\right)\right)\right]\left.\right)\left.\right)\left.\right) \quad .$$

Определенная таким образом функция sem позволяет получить множество конечных состояний системы для любого порядка выполнения потоков.

5.4. Свойства модели

В настоящем разделе установлены свойства модели динамического параллельного исполнения. В первую очередь эти свойства связаны с двумя специальными режимами исполнения программы — последовательным режимом исполнения и режимом исполнения по требованию.

5.4.1. Последовательный режим исполнения

Под последовательным режимом исполнения программы понимается такой способ планирования потоков, при котором после системного вызова SpawnReq выполнение вызывающего потока приостанавливается, пока не будет завершен созданный в результате обработки системного вызова поток. Такой режим считается эквивалентным последовательному исполнению программы, при котором системный вызов spawn работает как вызов функции программы. Функция step-seq осуществляет изменение состояния системы, соответствующее выбору из списка доступных потока с наибольшим идентификатором из множества тех потоков, которые могут быть запланированы. Функция sem-seq определяет семантику последовательного исполнения программы, аналогично функции sem, за исключением того, что модификация состояния осуществляется с помощью функции step-seq.

Для дальнейшего рассмотрения необходимо ввести набор терминов, которые будут использоваться в формулировках лемм и доказательствах теорем. Поток *может быть запланирован* в некотором состоянии системы, если значение предиката can-handle для этого состояния и потока истинно. *Планированием* назовем один из результатов недетерминированного выбора следующего потока для выполнения в функции step-full.

Следующие далее леммы устанавливают свойства модели, связанные с последовательным режимом исполнения.

***Лемма* 5.1.** *Вычисление, которое возвращает handle детерминировано:*

$$\text{dummy} : S$$

$$\text{handle-single} : \Pi\ (\text{prog} : \text{Prog})\ .\ (\text{system} : \text{System})\ .\ (\text{thread} : \text{Thread})$$
$$\Sigma\ (x : \text{System})\ .$$
$$\textbf{let}\ t \equiv (\text{run} \cdot (\text{handle} \cdot \text{thread} \cdot \text{system})) \cdot \text{prog} \cdot \text{empty} \cdot \text{dummy}$$
$$\textbf{in}\ t = x$$

Доказательство. Требуемое значение x может быть получено с помощью подстановки в результат функции handle требуемых параметров prog, empty, dummy. □

***Определение* 5.7.** *Функция get-result получает результат выполнения точки входа программы из состояния системы:*

$$\text{get-result} : \Pi\ (\text{system} : \text{System})\ .\ \text{Maybe} \cdot S$$
$$\text{get-result} \equiv \lambda\ (\text{system} : \text{System})\ .\ \textbf{let}$$
$$\text{entry-thread} \equiv \text{get}_{\text{Map}} \cdot (\text{threads} \cdot \text{system}) \cdot 0$$
$$\textbf{in}\ \ \textbf{match}\ (\text{semantics thread})$$
$$[\text{Done Maybe/Just}]$$
$$[\text{_Maybe/None}]$$

***Определение* 5.8.** *Свойство завершимой в режиме последовательного исполнения программы выражается с использованием следующей функции:*

$$\text{seq-property} : \Pi(\Pi\text{System}.\textbf{Type}).\textbf{Type}$$
$$\text{seq-property} \equiv \lambda\ (\text{prop} : \Pi\text{System}.\textbf{Type}).$$
$$\Pi\ (p : \text{Prog})\ .\ (s_0 : S)\ .\ (n : \mathbb{N})\ .$$
$$\textbf{let}$$
$$\text{sem} \equiv \textbf{iter}_{\mathbb{N}}\big(n, \text{CFL} \cdot \text{Id} \cdot \text{System}, \textbf{return}\ \text{System}_0 \cdot p,$$
$$(m, \text{prev}) \mapsto \textbf{bind}\ \text{prev step-seq}\big)$$
$$\text{res} \equiv \text{sem} \cdot p \cdot \text{empty} \cdot s0$$
$$\text{entry} \equiv \text{get}_{\text{Map}} \cdot (\text{threads} \cdot \text{res}) \cdot 0$$

$$
\begin{aligned}
&\textbf{in} \quad \textbf{match}\ (\text{semantics} \cdot \text{entry}) \\
&\qquad [\text{Done}\ (\lambda\ _.\ \text{prop} \cdot \text{res})] \\
&\qquad [\text{Step}\ (\lambda\ _.\ \#1)] \quad .
\end{aligned}
$$

Таким образом, для незавершимой в режиме последовательного исполнения программы свойство вырождается в тривиальное, а выполнение предиката требуется только для состояний, в которых точка входа программы завершена. Аналогично могут быть определены свойства в любом из режимов исполнения. В частности, для step-full аналогичную функцию назовем full-property. Для удобства использования такой функции в последующих доказательствах, первые два параметра (статическая семантика программы и исходное состояние) в функцию full-property будут передаваться в качестве параметров.

***Лемма* 5.2.** *При завершении выполнения программы в режиме последовательного исполнения все потоки находятся в состоянии завершённых вычислений:*

$$
\begin{aligned}
&\text{seq-all-threads-complete} : \textbf{let} \\
&\quad \text{prop} \ \equiv\ \lambda\ (s : \text{System})\ .\ \Pi\ (i : \text{TId})\ .\ (\text{leq}_{\mathbb{N}} \cdot i \cdot (\text{maxid} \cdot s))\ .\ \textbf{match}\ (\text{semantics} \cdot (\text{get}_{\text{Map}} \cdot s \cdot i)) \\
&\qquad\qquad [\text{Done}\ (\lambda\ _.\ \#1)] \\
&\qquad\qquad [\text{Step}\ (\lambda\ _.\ \#0)] \\
&\qquad \textbf{in} \quad \text{seq-property} \cdot \text{prop} \quad .
\end{aligned}
$$

Доказательство. На каждом шаге функция step-seq выполняет шаг потока из числа доступных для выполнения с максимальным идентификатором. Рассмотрим номер шага, в результате которого точка входа переходит из незавершённого состояния в завершённое. На этом шаге должны быть завершены все потоки, кроме точки входа, иначе будет запланировано выполнение другого потока. □

Следующая теорема устанавливает основное свойство последовательного режима исполнения. Формальное доказательство теоремы 5.1 и приведённой далее теоремы 5.2 находится в процессе разработки. В настоящей работе такие теоремы доказываются без построения терма-доказательства в терминах λ-исчисления с зависимыми типами.

***Теорема* 5.1.** *Если программа завершается в режиме последовательного исполнения, то она завершается и при любом другом планировании потоков с тем же результатом, то есть:*

$$\text{sequential-minimal} \ : \ \text{seq-property} \cdot \left(\lambda \left(\text{s-seq} : \text{System}\right) . \text{full-property} \cdot p \cdot s_0 \cdot \left(\lambda \left(\text{f-seq} : \text{List} \cdot \text{System}\right) . \text{All f-seq} \left(\lambda \left(\text{f-res} : \text{System}\right) . \text{s-seq} = \text{f-res}\right)\right)\right)$$

Доказательство. Как было отмечено ранее, если программа не завершается в режиме последовательного исполнения, свойство seq-property для любого предиката вырождается в тривиальное.

В случае, если на некотором шаге n последовательное исполнение завершилось с результатом s1, рассмотрим состояния на каждом из шагов n-1, …, 0 и покажем по индукции, что, при любом планировании из такого состояния, выполнение программы завершится с результатом s1. Для шага n-1 по лемме 5.2 не завершён только нулевой поток, который и будет запланирован при любом планировании.

Пусть требуемое утверждение выполняется для шага k+1. Для шага k, рассмотрим по индукции количество потоков, выполнение которых может быть запланировано. Если может быть запланирован только один поток, результат выполнения step-seq будет совпадать с единственным результатом выполнения step-full. Обозначим идентификатор дополнительного потока как i1. Считаем, что $i_1 \neq 0$.

По предположению, в режиме последовательного выполнения все потоки программы завершаются с требуемым результатом за $n_0 - n$ шагов. По определению step-seq следует, что каждый из потоков, которые могут быть запланированы, за не более чем $n_0 - n$ шагов переходит в состояние, в котором этот поток не может быть запланирован, независимо от планирования потоков. Таким образом, если для выполнения планируется поток с идентификатором отличным от i1, состояние системы будет изменяться в точности так, как и для случая m потоков. Поток с идентификатором i1 за не более чем $n_0 - n$ шагов перейдёт в состояние, когда он не может быть запланирован. Следовательно, за не более чем $n_0 - n$ шагов система прийдёт в одно из состояний, которое находится в отношении $\text{leq}_{\text{System}}$ с состоянием последовательного выполнения

на шаге $k+1$. По предположению ндукции, для таких состояний утверждение теоремы выполняется. На этом доказательство теоремы завершено. □

5.4.2. Режим исполнения по требованию

Под режимом исполнения программы по требованию понимается такой способ планирования потоков, при котором планируется к выполнению поток с идентификатором 0 до того момента, пока он не будет прерван системным вызовом WaitReq. В этом случае планируется тот поток, значение от которого ожидается точкой входа. Далее такая схема распространяется рекурсивно.

Следует отметить, что любой поток может ожидать завершения только потоков с большим, чем у него идентификатором.

***Определение* 5.9.** *Семантика параллельного исполнения задаётся следующими тремя функцями:*

$$\begin{aligned} \text{schedule-lazy} &: \Pi \text{System}.\text{Id}.\text{Id} \\ \text{step-lazy} &: \Pi \text{System}.\text{CFL}_{\text{system}} \cdot \text{System} \\ \text{sem-lazy} &: \Pi \text{Prog}.\text{S}.\ \text{Partial} \cdot \left(\text{CFL} \cdot \text{P} \cdot \text{System}\right) \end{aligned}$$

Функция schedule-lazy проверяет состояние потока с заданным идентификатором. Если этот поток завершил работу или может быть запланирован, возвращается его идентификатор. Если же поток ожидает завершения другого потока, функция возвращает идентификатор этого потока.

Функция step-lazy, с учётом сделанного выше замечания, получает идентификатор потока, запланированного согласно определению режима исполнения по требованию. Функция sem-lazy определяет семантику программы в режиме исполнения по требованию.

***Теорема* 5.2.** *Если программа завершается при некотором планировании step, она завершается в режиме исполнения по требованию с тем же результатом.*

Доказательство. Справедливость утверждения теоремы может быть показана аналогично теореме 5.1. А именно, по индукции от максимального номера шага (n) к

исходному состоянию рассматриваются состояния системы. Для каждого из таких состояний выполняется индукция по количеству потоков, доступных для выполнения. В случае одного доступного для выполнения потока все методы планирования дают идентичный результат, т.к. ситуация взаимной блокировки в модели исключена по построению. При наличии нескольких потоков, выбранный в режиме исполнения по требованию поток совпадает с выбранным функцией step, утверждение теоремы выполняется по предположению индукции.

Пусть выбранный при рассматриваемом планировании step поток отличается от выбранного в режиме исполнения по требованию. Тогда имеет место следующая ситуация: состояние системы содержит цепь потоков с идентификаторами $0 < i_1 < \ldots < i_k$, каждый предыдущий из которых ожидает завершения последующего, а последний i_k может быть запланирован. Случай, когда точка входа может быть запланирована, является частным для рассматриваемого. Тогда при планировании step исполняться на данном шаге будет поток с идентификатором j, не входящим в цепь.

Рассмотрим дальнейшие n-k шагов при планировании step. Если ни на одном из этих шагов точка входа не ожидает результата выполнения j-го потока, то с позиций режима исполнения по требованию результат этого шага может быть проигнорирован (j-й поток не будет запланирован) и утверждение теоремы выполняется по предположению индукции. В противном случае при режиме исполнения программы по требованию, начиная с рассматриваемого состояния, поток с идентификатором j будет запланирован на некотором шаге, после которого состояние системы придёт к рассматриваемому виду.

Таким образом, рассмотрены все требуемые случаи для шага индукции, и доказательство теоремы завершено. □

5.5. Примеры

В настоящем разделе приведены две серии примеров, с использованием которых поясняется устройство и применение модели. Первая серия примеров связана с вопро-

сами завершимости и незавершимости программ в различных режимах исполнения. В рамках заданных для модели ограничений, эти вопросы являются наиболее интересными для рассмотрения. Вторая серия примеров связана с приложением модели к фрагменту кода CMS (приложение E), модель которого была получена в разделе 4.3.

5.5.1. Завершимость в различных режимах исполнения

Рассмотрим некоторые примеры, поясняющие устройство модели. Для этого введём простейший язык, локальным состоянием которого является целое число. В следующей формуле указаны основные элементы семантики рассматриваемого языка:

$$
\begin{aligned}
S &\equiv \text{Integer} \\
Cf &\equiv \text{adt} \left\{ \text{inc dec} : \#1 \right\} \left\{ \text{load} : S \right\} \\
Ce &\equiv \lambda \left(\text{name} : Cf\right) . \textbf{match} \text{ name} \\
&\quad \left[\text{inc} \quad \left(\lambda \ \#1 \left(n : S\right) . n + 1\right) \right] \\
&\quad \left[\text{dec} \quad \left(\lambda \ \#1 \left(n : S\right) . n - 1\right) \right] \\
&\quad \left[\text{load} \quad \left(\lambda \left(m \ n : S\right) . m\right) \right] \\
Mf &\equiv \text{adt} \left\{ \text{get}_1 \ \text{get}_2 \ \text{sum} : \#1 \right\} \\
Me &\equiv \lambda \left(\text{name} : Mf\right) . \textbf{match} \text{ name} \\
&\quad \left[\text{get}_1 \quad \left(\lambda \ \#1 \left(\text{trunk branch} : S\right) . \text{trunk}\right)\right] \\
&\quad \left[\text{get}_2 \quad \left(\lambda \ \#1 \left(\text{trunk branch} : S\right) . \text{branch}\right)\right] \\
&\quad \left[\text{sum} \quad \left(\lambda \ \#1 \left(\text{trunk branch} : S\right) . \text{trunk} + \text{branch}\right)\right] \\
Bf &\equiv \text{adt} \left\{ \text{lt gt} : \text{integer} \right\} \\
Be &\equiv \lambda \left(\text{name} : Bf\right) . \textbf{match} \text{ name} \\
&\quad \left[\text{lt} \quad \left(\lambda \left(\text{right left} : S\right) . \text{lt}_{\text{Integer}} \cdot \text{left} \cdot \text{right}\right)\right] \\
&\quad \left[\text{gt} \quad \left(\lambda \left(\text{right left} : S\right) . \text{gt}_{\text{Integer}} \cdot \text{left} \cdot \text{right}\right)\right]
\end{aligned}
$$

Функции Cf, выполняющие вычисление, состоят из операций инкремента и декремента, а также группы функций load, которые загружают в локальное состояние некоторую целочисленную константу. В языке определяются три функции, выполняющие слияние. Функции get1 и get2 в качестве итогового локального состояния возвращают первый (состояние, возвращенное вызванной функцией) и второй

(локальное состояние исполняемой функции) аргументы соответственно. Функция sum возвращает сумму возвращенного и исходного состояния. Функции сравнения lt и gt выполняют сравнение локального состояния с константой.

Для удобства будем использовать запись seq* как правоассоциативное сокращение для нескольких последовательно выполняемых команд. Первый пример показывает различие между режимами исполнения по порядку и по требованию:

$$
\begin{aligned}
\mathrm{Var}_1 &\equiv \#2 \\
\mathrm{Tf}_1 &\equiv \mathrm{adt}\left\{\ \mathrm{main\ fun} : \#1\ \right\} \\
\mathrm{Te}_1 &\equiv \lambda\left(\mathrm{name} : \mathrm{Tf}\right) . \mathbf{match}\ \mathrm{name} \\
&\quad \big[\mathrm{main} \quad \big(\lambda\ \#1\ .\ \big(\mathrm{seq}^*\ \big[\\
&\qquad\qquad \left(\mathrm{spawn} \cdot \mathbf{tuple}\left(\mathrm{fun},\ \mathrm{load} \cdot 1,\ 0\#2\right)\right), \\
&\qquad\qquad \left(\mathrm{spawn} \cdot \mathbf{tuple}\left(\mathrm{fun},\ \mathrm{load} \cdot 2,\ 1\#2\right)\right), \\
&\qquad\qquad \left(\mathrm{wait} \cdot \mathbf{pair}\left(0\#2,\ \mathrm{get}_2\right)\right)\big]\big)\big)\big] \\
&\quad \left[\mathrm{fun} \quad \left(\lambda\ \#1\ .\ \mathrm{skip}\right)\right] \\
\mathrm{Prog}_1 &\equiv \mathbf{tuple}\left(\mathrm{Ce},\ \mathrm{Me},\ \mathrm{Be},\ \mathrm{Te}_1,\ \mathrm{main}\right) \qquad .
\end{aligned}
$$

Точка входа main порождает 2 потока, вычисляющие значения функции fun для различных исходных состояний. Далее точка входа ожидает завершения вычисления потока с идентификатором в переменной 0#2. В случае режима исполнения по порядку после исполнения первой строки main управление будет передано вновь порожденному потоку до тех пор, пока его вычисление не завершится. Аналогичным образом произойдет обработка второй строки. На момент выполнения третьей строки вычисления для каждого из порожденных потоков будут завершены. В случае режима исполнения по требованию, выполнение main будет проходить непрерывно до третьей строки. Далее, поток, соответствующий точке входа, будет заблокирован и управление, согласно функции schedule-lazy будет передано потоку, идентификатор которого был сохранен в переменной 0#2. После завершения этого потока управление будет передано потоку, соответствующему точке входа программы. Таким образом, в режиме исполнения по требованию после завершения вычисления функции main,

поток, идентификатор которого был сохранен в переменной v1, будет находиться в исходном состоянии и управление ему передано не будет.

Внесём незначительные модификации в первый пример, чтобы проиллюстрировать случай, при котором исполнение программы по порядку не будет завершено, тогда как исполнение программы по требованию завершится:

$$\begin{aligned}
\mathrm{Te}_2\left(\mathrm{main}\right) &\equiv \mathrm{Te}_1\left(\mathrm{main}\right)\\
\mathrm{Te}_2\left(\mathrm{fun}\right) &\equiv \lambda\,\#1.\ \text{while-do}\cdot\left(\mathrm{gt}\cdot 1\right)\cdot \mathrm{skip}\\
\mathrm{Prog}_2 &\equiv \mathbf{tuple}\left(\mathrm{Ce},\ \mathrm{Me},\ \mathrm{Be},\ \mathrm{Te}_2,\ \mathrm{main}\right)
\end{aligned}$$

В этом примере функция fun в случае, когда аргумент больше 1, переходит в бесконечный цикл. Для 1 и меньших аргументов функция эквивалентна рассматриваемой в первом примере. В режиме исполнения по порядку после обработки второй строки управление будет передано вновь порожденному потоку. Как следствие — вычисление значения точки входа не будет завершено, хотя, как можно заметить из структуры программы, значение, возвращаемое не завершающимся потоком не требуется для вычисления значения, возвращаемого точкой входа в программу. В режиме исполнения по требованию, как и в предыдущем примере, второму потоку не будет передано управление и, следовательно, вычисление значения, возвращаемого точкой входа будет завершено успешно.

Согласно доказанным теоремам 5.1 и 5.2, поведение программы при параллельном исполнении определяется поведением программы в режимах исполнения по порядку и по требованию, которые являются последовательными. При параллельном исполнении программы из второго примера, при наличии достаточного количества вычислительных устройств (хотя бы двух), вычисление потока, соответствующего точке входа, также будет завершено. Более того, может быть сформулировано и показано, что при справедливом планировании (fair scheduling), при котором каждый из доступных для выполнения потоков получит управление за конечное время, вычисление потока, соответствующего точке входа, также будет завершено.

5.5.2. Параллельное исполнение фрагмента кода CMS

В разделе 4.3 ранее была описана модель динамической семантики фрагмента кода CMS. Настоящий раздел посвящён вопросам приложения модели динамического параллельного исполнения к этому фрагменту исходного кода.

Заметим, что семантику функции `ysparg`, согласно определению 4.4, можно рассматривать как элемент следующего типа:

$$\text{ysparg} \quad : \quad \text{ECMA 335}_{\text{method}} \cdot \text{Id} \cdot \#1 \quad .$$

Путём подстановки требуемых параметров, функция `ysparg` может быть приведена к виду, который требуется моделью динамического параллельного исполнения программ с состоянием, тип которого представлен в следующей формуле:

$$\text{ECMA 335}_{\text{state}} \quad \equiv \quad \Sigma \left(\text{g} : \text{GlobalState}\right) \left(\text{l} : \text{FunctionState}\right) \left(\text{s} : \text{List} \cdot \text{FunctionState}\right) \quad .$$

Тип указателей на элемент на куче heapArrayElement в используемой упрощённой модели динамической семантики содержит информацию о том, к элементу какого из массивов производится доступ. При использовании функции `ysparg` на практике в задачах моделирования теплогидравлических процессов в реакторах атомных электростанций эта функция вызывается с известными статически указателями на элементы массива [175].

Точка входа для простого случая, при котором рассматриваются отдельные массивы для входных и выходных потоков part1 и part2 с размером total, может быть задана с использованием следующей формулы ($\text{prepareCallensuremath}'$ — это модификация функции prepareCall):

$$\begin{aligned}
\text{CMS}_{\text{main}} \quad \equiv \quad & \text{seq*} \Big[\\
& \text{spawn} \cdot \textbf{tuple}\big(\text{ysparg}, \text{prepareCall}' \cdot \text{cons*} \big[\text{p-out}, \text{p-in-out}, \text{p-in-in}, \text{d}_1, \text{d}_2, \text{dtt}, \\
& \ \text{mkRef} \cdot \text{part}_1 \cdot 0, \text{mkRef} \cdot \text{part}_2 \cdot 0, \text{mkRef} \cdot \text{nf} \cdot 0\big], 0\big), \\
& \ldots \\
& \text{spawn} \cdot \textbf{tuple}\big(\text{ysparg}, \text{prepareCall}' \cdot \text{cons*} \big[\text{p-out}, \text{p-in-out}, \text{p-in-in}, \text{d}_1, \text{d}_2, \text{dtt}, \\
& \ \text{mkRef} \cdot \text{part}_1 \ . \ \big(\text{total-1}\big), \text{mkref} \cdot \text{part}_2 \cdot \big(\text{total-1}\big), \text{mkref} \cdot \text{nf} \cdot \big(\text{total-1}\big)\big], \text{total-1}\big), \\
& \text{skip},
\end{aligned}$$

$$\text{wait} \cdot 0 \cdot \left(\text{mergeArrays} \cdot 0\right)$$

$$\ldots$$

$$\text{wait} \cdot \left(\text{total-1}\right) \cdot \left(\text{mergeArrays} \cdot \left(\text{total-1}\right)\right)$$

$$\Big] \quad .$$

Отметим, что представленный пример предназначен для демонстрации работы модели динамического параллельного исполнения программ. Он имеет недостаточно сложную структуру по отношению к фрагментам кода, динамическое параллельное исполнение которых целесообразно на практике. Тем не менее, в рамках НИР по разработке алгоритма параллельного исполнения программ автором и И.С. Астаповым [175] было выявлено, что основные свойства, на которые опираются формулы настоящего раздела, сохраняются для каждой стадии кода CMS. К числу таких свойств относится, в первую очередь, наличие статической информации о расположении в памяти каждого из элементов массива, используемых при моделировании.

5.6. Выводы по пятой главе

Предложенная модель процессов динамического параллельного исполнения программ может быть с незначительными изменениями использована для любых императивных языков. С помощью расширения набора системных вызовов и, соответственно, расширения семантики языка управления потоком исполнения программ, в модель могут быть введены различные методы обмена сообщениями, а также, при необходимости — память с общим доступом.

Следует отметить отличия применяемого преобразователя монад возобновлений, заданного в рамках теории типов, по сравнению с оригинальным определением Папаспироу на основе теории доменов. Способность теории доменов отражать частичные (незавершимые) функции позволяет определить функцию $\text{run} : \text{ResT} \cdot \text{m} \cdot \text{a} \rightarrow \text{m} \cdot \text{a}$ — без использования монады частичности. Объект типа $\text{m} \cdot \text{a}$ удобнее для доказательства свойств программ. При определении функции run Папаспироу опирается на комбинатор неподвижной точки, который, как отмечено в предыдущих главах,

в рамках теории типов позволяет построить парадокс. В этой связи определение аналогичной функции run в рамках теории типов невозможно.

Для того, чтобы использовать модель динамического параллельного исполнения на практике, потребуется её доработка с позиций описания доступа к памяти.

Автоматическая гранулярность операций из Cf, Bf, Mf на уровне отдельных доступов к памяти может быть обеспечена в случае, если семантика исходного языка в модели динамического параллельного исполнения будет представлена в форме стека преобразователей монад. Предположим, что тип $\mathrm{Base}_{\mathrm{state}}$ содержит информацию о состоянии исполнения в рамках динамической семантики расширяемого языка, а также что динамическая семантика может быть факторизована относительно типа, реализующего интерфейс преобразователя монад состояния $\mathrm{StateT} \cdot \mathrm{Base}_{\mathrm{state}}$. Последнее означает, что семантика исходного языка может быть представлена в следующем виде для некоторого преобразователя монад T:

$$\Pi \left(\mathrm{State}_{\mathrm{impl}} : \mathrm{StateT\text{-}interface}\right) . \left(\mathrm{m} : \mathrm{Monad}\right) . \mathrm{State}_{\mathrm{impl}} \cdot \left(\mathrm{T} \cdot \mathrm{m}\right) \qquad .$$

Интерфейс StateT-interface в дополнение к функциям и свойствам, входящим в определение преобразователя монад, содержит также функции частичного чтения (gets) и обновления (modify) состояния. Такой интерфейс, дополненный стандартными функциями полного изменения состояния get и put используется, например, в библиотеке преобразователей монад mtl для языка Haskell в виде класса MonadState. В качестве альтернативной реализации этого интерфейса в модели динамического параллельного исполнения программ может применяться модифицированный преобразователь монад состояния, который используется для выделения шагов в семантике исходного языка, гранулированных по доступу к памяти. Последовательность вычислений с состоянием в данной реализации описывается как список шагов с указанием, осуществляется на данном шаге частичное чтение или обновление состояния.

Кроме того, возможности λ-исчисления с зависимыми типами могут быть приложены для статического описания свойства непересечения параллельно выполняемых функций по областям чтения-записи.

Заключение

Диссертационное исследование, результаты которого представлены в настоящей работе, посвящено процессам описания формальных моделей компьютерных программ и языков программирования, используемых в исходных кодах программ. Целью исследования является разработка математических моделей и основанного на них макета программного средства, которые, в свою очередь, предназначены для построения формальных моделей программ и верификации их функциональных свойств, включая программы, написанные на нескольких языках программирования, с помощью промежуточного представления, основанного на модели λ-исчисления с зависимыми типами.

В диссертации получены следующие, перечисленные далее основные результаты.

1. Предложена новая разновидность лямбда-исчисления с зависимыми типами, обеспечивающая с поддержку нетривиальных типов идентичности путём введения дополнительных, не рассматривавшихся ранее правил редукции для элементов типов идентичности.

2. Разрабатываемая разновидность реализована в форме макета языка программирования и программного средства, в которых могут быть использованы спецификации, существенно использующие нетривиальные типы идентичности с помощью реализации предложенных автором правил редукции.

3. Построена новая модель статической формальной семантики промежуточного кода, соответствующего подмножеству стандарта ECMA-335, поддерживающая обобщённые типы согласно четвёртой редакции стандарта.

4. Построена новая формальная модель динамического параллельного исполнения программ, которая рассматривается как модификатор формальной семантики языка программирования.

С позиций практического применения новых методов и средств, представленных в настоящей работе, эффективные результаты могут быть получены в задачах формальной верификации программ, которые транслируются в промежуточное представление стандарта ECMA-335. Следует отметить, что работы на этом направлении целесообразно проводить в пилотном режиме, учитывая необходимость расширения динамической семантики подмножества управляемого кода.

Модель динамического параллельного исполнения программ в её расширенном варианте, допускающем статическую проверку независимости параллельно исполняемых процессов, может быть использована в качестве основы для языка динамического распределения вычислительных заданий для крупномасштабных Grid-систем и систем высокопроизводительных вычислений. В качестве примера существующего, используемого на практике аналога таких языков можно привести средство DAGMan системы планирования заданий HTCondor.

С учётом выводов, полученных в ходе исследования и анализа мирового опыта разработки языков программирования, представленных автором в расширенной библиографии, в качестве основного направления дальнейших исследований процессов описания формальных моделей компьютерных программ и языков программирования следует выделить применение полученных математических моделей и средств в рамках методологии языково-ориентированного программирования. Концепция промежуточного представления для дедуктивной верификации программ, реализованная автором в настоящей работе в форме макета базового языка и программного средства, соответствует требованиям к средствам верификации программ, написанных на нескольких предметно-ориентированных языках программирования. Таким образом, в первую очередь для дальнейшего развития исследований на этом направлении необходима разработка средств для эффективного построения формальных спецификаций большого количества предметно-ориентированных языков программирования на основе построенных в настоящей работе макетов.

Список рисунков

2.1 Использование индексов де Брёйна в нетипизированном λ-исчислении 45

C.1 Пример числа с плавающей точкой одинарной точности 185

Список таблиц

2.1 Соответствие Карри-Говарда 55

3.1 Соответствие между синтаксическими конструкциями языка и термами исчисления 87

A.1 Сводная статистика по плотности дефектов, обнаруженных в рамках исследования Coverity Scan в 2008–2012 гг. 177

A.2 Результаты анализа по количеству языков программирования, используемых при разработке программных продуктов с открытым исходным кодом 178

C.1 Сравнение результатов вычислений модели с аппаратной реализацией чисел с плавающей точкой 188

C.2 Сравнение относительной погрешности вычислений с использованием модели чисел с плавающей точкой и модели вычислений на рациональных числах с результатами, полученными аналитически 188

Список листингов

1 Синтаксис базового языка в нотации ABNF 85

2 Схема диспетчеризации стадии преобразования в термы 90

3 Пример трансляции терма удаления типа идентичности J 90

4 Реализация функций проверки типов для терма приложения 93

5 Парадокс, возможный при исключении ограничений на граф уровней универсумов 96

6 Пример взаимно-индуктивного определения — область видимости определения и ссылка на пользовательский тип данных 105

7 Семантика оператора «?.» в языке C# 114

8 Ошибка, выводимая Coq при определении преобразователя монад возобновлений в виде уравнения доменов 132

9 Модель числа с плавающей точкой 186

10 Интерфейс модели арифметики на приближениях действительных чисел 186

Литература

1. Abel Andreas. Towards Generic Programming with Sized Types // Mathematics of Program Construction / Ed. by Tarmo Uustalu. — Springer Berlin Heidelberg, 2006. — Lecture Notes in Computer Science no. 4014. — P. 10–28. — doi:10.1007/11783596_4.

2. Adámek Jiri, Milius Stefan, Moss Lawrence. Initial algebras and terminal coalgebras: a survey. — 2010. —June. — Draft, available online. URL: https://www.tu-braunschweig.de/Medien-DB/iti/survey_full.pdf (online; accessed: 2015-05-20).

3. Altenkirch Thorsten. Constructions, Inductive Types and Strong Normalization : Ph. D. thesis / Thorsten Altenkirch ; University of Edinburgh. — Edinburhg, UK, 1993. — 183 p. — URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.38.2661&rep=rep1&type=pdf (online; accessed: 2015-05-20).

4. Altenkirch Thorsten, McBride C. Towards observational type theory. — 2006. — Manuscript, available online. URL: http://synrc.com/publications/cat/Temp/Logic/OTT.pdf (online; accessed: 2015-05-20).

5. Altenkirch Thorsten, McBride Conor, Swierstra Wouter. Observational Equality, Now! // Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007 / Ed. by Aaron Stump, Hongwei Xi. — 2007. — P. 57–68.

6. Appel A.W. Foundational proof-carrying code // Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. — IEEE Comput. Soc, 2001. — P. 247–256. — doi:10.1109/LICS.2001.932501.

7. Appel Andrew W. Verified software toolchain // Programming Languages and Systems. — Springer, 2011. — P. 1–17. — URL: http://link.springer.com/chapter/10.1007/978-3-642-19718-5_1 (online; accessed: 2015-02-25).

8. Awodey Steve, Warren Michael A. Homotopy theoretic models of identity types // Mathematical Proceedings of the Cambridge Philosophical Society. — 2009. — Vol. 146. — P. 45–55.

9. Baier Christel, Katoen Joost-Pieter. Principles Of Model Checking. — MIT Press, 2008. — 950 p.

10. Bansal Sorav, Aiken Alex. Automatic generation of peephole superoptimizers // ACM Sigplan Notices. — Vol. 41. — ACM, 2006. — P. 394–403. — doi:10.1145/1168918.1168906.

11. Benke Marcin, Dybjer Peter, Jansson Patrik. Universes for generic programs and proofs in dependent type theory // Nord. J. Comput. — 2003. — Vol. 10, no. 4. — P. 265–289. — URL: http://www.researchgate.net/profile/Peter_Dybjer/publication/220673187_Universes_for_Generic_Programs_and_Proofs_in_Dependent_Type_Theory/links/0912f50cb23ed2c8b7000000.pdf (online; accessed: 2015-02-25).

12. Benton Nick, Kennedy Andrew, Varming Carsten. Some Domain Theory and Denotational Semantics in Coq // Theorem Proving in Higher Order Logics / Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, Makarius Wenzel. — Springer Berlin Heidelberg, 2009. — Lecture Notes in Computer Science no. 5674. — P. 115–130. — doi:10.1007/978-3-642-03359-9_10.

13. Bessey B Y A L, Block Ken. A few Billion Lines of code Later using static Analysis to find Bugs in the Real World // Communications of the ACM. — 2010. — Vol. 53, no. 2. — P. 66–75. — doi:10.1145/1646353.1646374.

14. Bezem Marc, Coquand Thierry, Huber Simon. A model of type theory in cubical sets // 19th International Conference on Types for Proofs and Programs (TYPES 2013). — Vol. 26. — 2014. — P. 107–128. — URL: https://www.google.com/books?hl=en&lr=&id=ohEtBAAAQBAJ&oi=fnd&pg=PA107&dq=cubical+sets+coquand&ots=IrzfSdj_vL&sig=DSBKDVmoLf1hWgdssaTTN0Wn9qM (online; accessed: 2015-06-07).

15. Blazy Rine, Dargaye Zaynah, Leroy Xavier. Formal Verification of a C Compiler Front-End // FM 2006: Formal Methods. — Lecture Notes in Computer Science 4085. — Springer Berlin Heidelberg, 2006. — P. 460–475. — doi:10.1007/11813040_31.

16. Boehm Barry. A view of 20th and 21st century software engineering // ICSE'06 Proceedings of the 28th international conference on Software engineering. — ACM Press, 2006. — P. 12–29. — doi:10.1145/1134285.1134288.

17. Borrás Cari. Overexposure of radiation therapy patients in Panama: problem recognition and follow-up measures // Revista Panamericana de Salud Pública. — 2006. — September. — Vol. 20, no. 2-3. — P. 173–187. — doi:10.1590/S1020-49892006000800014.

18. Reversible Debugging Software : Rep. / University of Cambridge, Judge Business School ; Executor: Tom Britton, L Jeng, G Carver. — Cambdidge, UK : 2013. — URL: http://download.microsoft.com/documents/rus/visualstudio/03_CambridgeUniversity_study-time_and_cost_saved_using_RDBs-January_20....pdf (online; accessed: 2015-05-31).

19. Certified Complexity (CerCo) / Roberto M. Amadio, Nicolas Ayache, Francois Bobot et al. // Foundational and Practical Aspects of Resource Analysis / Ed. by Ugo Dal Lago, Ricardo Peña.

— Springer International Publishing, 2013. —August. — Lecture Notes in Computer Science. — P. 1–18. — doi:10.1007/978-3-319-12466-7_1.

20. Chapman James Maitland. Type checking and normalisation : Ph.D. thesis / James Maitland Chapman ; University of Nottingham. — Nottingham, UK, 2008. —October. — v+115 p. — URL: http://eprints.nottingham.ac.uk/id/eprint/10824.

21. Church Alonzo. A Set of Postulates for the Foundation of Logic // The Annals of Mathematics. — 1932. —April. — Vol. 33, no. 2. — P. 346–366. — doi:10.2307/1968337.

22. Church Alonzo. An Unsolvable Problem of Elementary Number Theory // American Journal of Mathematics. — 1936. —April. — Vol. 58, no. 2. — P. 345–345. — doi:10.2307/2371045.

23. Church Alonzo. A note on the Entscheidungsproblem // The Journal of Symbolic Logic. — 1936. —June. — Vol. 1, no. 1. — P. 40–41. — doi:10.2307/2269326.

24. Church Alonzo. A formulation of the simple theory of types // The Journal of Symbolic Logic. — 1940. —March. — Vol. 5, no. 02. — P. 56–68. — doi:10.2307/2266170.

25. Clarke E. M., Grumberg Orna, Peled Doron A. Model Checking. — Cambridge, Mass : The MIT Press, 1999. —January. — 314 p. — ISBN: 978-0-262-03270-4.

26. CompCert - Publications. — 2015. — URL: http://compcert.inria.fr/publi.html#chapters (дата обращения: 2015-02-25).

27. Comprehensive Formal Verification of an OS Microkernel / Gerwin Klein, June Andronick, Kevin Elphinstone et al. // ACM Trans. Comput. Syst. — 2014. —February. — Vol. 32, no. 1. — P. 2:1–2:70. — doi:10.1145/2560537.

28. Coquand Thierry. An Analysis of Girard's Paradox // In Symposium on Logic in Computer Science. — IEEE Computer Society Press, 1986. — P. 227–236. — URL: http://hal.inria.fr/docs/00/07/60/23/PDF/RR-0531.pdf.

29. Metamathematical investigations of a calculus of constructions : Rep. : RR-1088 / INRIA ; Executor: Thierry Coquand. — Rocquencourt, France : 1989. — 32 p. URL: https://hal.inria.fr/inria-00075471/PDF/RR-1088.pdf.

30. Coquand Thierry, Huet Gérard. The calculus of constructions // Information and Computation. — 1988. —February. — Vol. 76, no. 2-3. — P. 95–120. — doi:10.1016/0890-5401(88)90005-3.

31. Coverity Open Source Report : Rep. / Coverity Inc. ; Executor: Coverity : 2008. — 25 p. URL: http://www.coverity.com/library/pdf/Coverity-Scan_Open_Source_Report_2008.pdf (online; accessed: 2015-05-20).

32. Coverity Scan Open Source Report : Rep. / Coverity Inc. ; Executor: Coverity : 2009. — 35 p. URL: http://www.coverity.com/library/pdf/2009CoverityScanOpenSourceReport.pdf (online; accessed: 2015-05-20).

33. Coverity Scan: 2010 Open Source Integrity Report : Rep. / Coverity Inc. ; Executor: Coverity : 2010. — 29 p. URL: http://www.coverity.com/library/pdf/coverity-scan-2010-open-source-integrity-report.pdf.

34. Coverity Scan: 2011 Open Source Integrity Report : Rep. / Coverity Inc. ; Executor: Coverity : 2011. — 30 p. URL: http://www.coverity.com/library/pdf/coverity-scan-2011-open-source-integrity-report.pdf (online; accessed: 2015-05-20).

35. Coverity Scan : 2012 Open Source Report : Rep. / Coverity Inc. ; Executor: Coverity : 2012. — 61 p. URL: http://wpcme.coverity.com/wp-content/uploads/2012-Coverity-Scan-Report.pdf (online; accessed: 2015-05-20).

36. Coverity Scan: 2013 Open Source Report : Rep. / Coverity Inc. ; Executor: Coverity : 2013. — 25 p. URL: http://softwareintegrity.coverity.com/rs/coverity/images/2013-Coverity-Scan-Report.pdf (online; accessed: 2015-05-20).

37. Danial Al. Cloc – Count Lines of Code. — 2013. — URL: http://cloc.sourceforge.net/ (online; accessed: 2014-12-02).

38. Danielsson Nils Anders. Operational Semantics Using the Partiality Monad // Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. — ICFP ’12. — New York, NY, USA : ACM, 2012. — P. 127–138. — doi:10.1145/2364527.2364546.

39. De Bakker J.W., De Roeve W. A calculus for recursive program schemes // 1st International Colloquium on Automata, Lanugages and Programming. — 1972. — P. 167–196. — URL: http://oai.cwi.nl/oai/asset/9145/9145A.pdf (online; accessed: 2015-05-20).

40. Dijkstra Edsger W. Guarded commands, non-determinacy and formal derivation of programs // Communications of the ACM. — 1975. — August. — Vol. 18, no. 8. — P. 453–457. — doi:10.1145/360933.360975.

41. ECMA International. Standard ECMA-335 - Common Language Infrastructure (CLI). — 4 edition. — Geneva, Switzerland, 2006. — June. — URL: http://www.ecma-international.org/publications/standards/Ecma-335.htm.

42. Elliott Conal M. Higher-order unification with dependent function types // Rewriting Techniques and Applications / Ed. by G. Goos, J. Hartmanis, D. Barstow et al. —

Berlin, Heidelberg : Springer Berlin Heidelberg, 1989. — Vol. 355. — P. 121–136. — doi:10.1007/3-540-51081-8_104.

43. Fibred Data Types / Neil Ghani, Lorenzo Malatesta, Fredrik Nordvall Forsberg, Anton Setzer // Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science. — IEEE Computer Society, 2013. —June. — P. 243–252. — doi:10.1109/LICS.2013.30.

44. Flanagan Cormac, Felleisen Matthias. The semantics of future and its use in program optimization // Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — ACM, 1995. — P. 209–220. — doi:10.1145/199448.199484.

45. Floyd Robert W. Assigning Meanings to Programs // Mathematical Aspects of Computer Science / Ed. by J T Schwartz. — Vol. 19 of Proceedings of Symposia in Applied Mathematics. — American Mathematical Society, 1967. — P. 19–32. — doi:10.1007/978-94-011-1793-7_4.

46. Formal Verification of a Microkernel Used in Dependable Software Systems / C Baumann, B Beckert, H Blasum, T Bormer // Computer Safety, Reliability, and Security, Proceedings. — 2009. — Vol. 5775. — P. 187–200. — doi:10.1007/978-3-642-04468-7_16.

47. Forsberg Fredrik Nordvall, Setzer Anton. Inductive-inductive definitions // Computer Science Logic. — Springer, 2010. — P. 454–468. — doi:10.1007/978-3-642-15205-4_35.

48. Fumex Clément, Ghani Neil, Johann Patricia. Indexed induction and coinduction, fibrationally // Algebra and Coalgebra in Computer Science. — Springer, 2011. — P. 176–191. — doi:10.1007/978-3-642-22944-2_13.

49. GNU Project. Gzip 1.2.4 source code. — 1993. — URL: `ftp://ftp.gnu.org/gnu/gzip/gzip-1.2.4.tar.gz`.

50. GNU Project. Gzip 1.6 source code. — 2013. — URL: `ftp://ftp.gnu.org/gnu/gzip/gzip-1.6.tar.gz` (online; accessed: 2015-05-23).

51. Ghani Neil, Hancock Peter. Containers, monads and induction recursion // Mathematical Structures in Computer Science. — 2014. —November. — Vol. FirstView. — P. 1–25. — doi:10.1017/S0960129514000127.

52. Giménez Eduarde. Codifying Guarded Definitions with Recursive Schemes // Types for Proofs and Programs: International Workshop TYPES'94, Bastad, Sweden, June 6-10, 1994. Selected Papers. — Springer, 1995. — Vol. 996 of Lecture Notes in Computer Science. — P. 39–59. — doi:10.1007/3-540-60579-7_3.

53. Girard Jean-Yves. Interpretation fonctionelle et eleimination des coupures dans l'arithmetique d'ordre superieure : These d'etat / Jean-Yves Girard ; Université Paris VII. — 1972. — URL: http://www.cs.cmu.edu/~kw/scans/girard72thesis.pdf (online; accessed: 2012-04-22).

54. Gonthier Georges. Formal proof–the four-color theorem // Notices of the AMS. — 2008. — Vol. 55, no. 11. — P. 1382–1393. — URL: http://www.ams.org/notices/200811/tx081101382p.pdf (online; accessed: 2015-02-25).

55. Grattage Jonathan. An Overview of QML With a Concrete Implementation in Haskell // Electronic Notes in Theoretical Computer Science. — 2011. —February. — Vol. 270, no. 1. — P. 165–174. — doi:10.1016/j.entcs.2011.01.015.

56. Gzip — GNU Project — Free Software Foundation. — 2010. — URL: http://www.gnu.org/software/gzip/ (online; accessed: 2015-05-23).

57. Gödel Kurt. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I // Monatshefte für Mathematik und Physik. — 1931. —December. — Vol. 38-38, no. 1. — P. 173–198. — doi:10.1007/BF01700692.

58. Towards a mathematical theory of processes : Rep. : TR 25.125 / IBM Laboratory Vienna ; Executor: H Bekić. — Vienna, Austria : 1971. —December. — URL: http://homepages.cs.ncl.ac.uk/cliff.jones/publications/LNCS177-Bekic/Bekic5.pdf (online; accessed: 2015-05-14).

59. Hanson David R. LCC.NET: targeting the. NET Common Intermediate Language from Standard C // Software: Practice and Experience. — 2004. — Vol. 34, no. 3. — P. 265–286. — doi:10.1002/spe.563.

60. Harrison William L. The essence of multitasking // Algebraic Methodology and Software Technology. — Springer, 2006. — P. 158–172. — doi:10.1007/11784180_14.

61. Hilbert David, Ackermann Wilhelm. Grundzüge der theoretischen Logik. — Berlin : J. Springer, 1928. — 120 p.

62. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. — 1969. —October. — Vol. 12, no. 10. — P. 576–580. — doi:10.1145/363235.363259.

63. Hoare C. A. R. Communicating sequential processes // Communications of the ACM. — 1978. —August. — Vol. 21, no. 8. — P. 666–677. — doi:10.1145/359576.359585.

64. Homotopical patch theory / Carlo Angiuli, Edward Morehouse, Daniel R. Licata, Robert Harper // Proceedings of the 19th ACM SIGPLAN international conference on Functional programming. — ACM, 2014. — P. 243–256. — doi:10.1145/2628136.2628158.

65. The Gilbreath Trick: A Case Study in Axiomatisation and Proof Development in the Coq Proof Assistant : Rep. : RR-1511 / INRIA ; Executor: Huet, Gerard : 1991. — 25 p. URL: https://hal.archives-ouvertes.fr/docs/00/07/50/51/PDF/RR-1511.pdf (online; accessed: 2015-02-24).

66. The Watts New? Collection: Columns by the SEI's Watts Humphrey : Rep. : CMU/SEI-2009-SR-024 / Carnegie Mellon University, Software Engineering Institute ; Executor: Watts S Humphrey. — Pittsburgh PA : 2009. — P. 216–216. URL: http://www.sei.cmu.edu/library/assets/watts-new-compiled.pdf (online; accessed: 2015-02-21).

67. Hurkens Antonius J. C. A simplification of Girard's paradox // Typed Lambda Calculi and Applications / Ed. by Gerhard Goos, Juris Hartmanis, Jan van Leeuwen et al. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1995. — Vol. 902. — P. 266–278. — doi:10.1007/BFb0014058.

68. ISO. ISO 3166:1988: Codes for the representation of names of countries. — Geneva, Switzerland : International Organization for Standardization, 1988. — 58 p. — URL: http://www.din.de/gremien/nas/nabd/iso3166ma/codlstp1/index.html (online; accessed: 2015-06-05).

69. ISO. ISO 639:1988 – Codes for the representation of names of languages. — 1988. — URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22109.

70. ISO. ISO/IEC 10646-1:1993, Information technology. Universal Multiple-Octet Coded Character Set (UCS). Part 1: Architecture and Basic Multilingual Plane. — Geneva, Switzerland : International Organization for Standardization, 1993. — 754 p. — About two dozen corrections and amendments to this Standard have been published. URL: http://www.iso.ch/cate/d18741.html.

71. ISO. ISO/IEC/IEEE 60559:2011 Information technology — Microprocessor Systems — Floating-Point arithmetic. — Geneva, Switzerland : International Organization for Standardization, 2011. — 58 p. — URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57469.

72. Indelicato Greg. A guide to the project management body of knowledge (PMBOK ® guide), fourth edition / Ed. by David Cleland. — Project Management Institute, 2009. —June. — Vol. 40 of Project Management Journal. — 104 p. — ISBN: 978-1-933890-51-7.

73. The International Exascale Software Project roadmap / J. Dongarra, P. Beckman, T. Moore et al. // International Journal of High Performance Computing Applications. — 2011. —January. — Vol. 25, no. 1. — P. 3–60. — doi:10.1177/1094342010391989.

74. Jourdan Jacques-Henri, Leroy Xavier, Pottier Francois. Validating LR(1) Parsers // Proceedings of the 21st European Symposium on Programming. — Vol. 7211. — 2012. — P. 397–416. — doi:10.1007/978-3-642-28869-2_20.

75. Kleene Stephen Cole. Introduction to Metamathematics. — New York : Van Nostrand, 1952. — 280 p. — ISBN: 0-7204-2103-9.

76. Kleene S. C., Rosser J. B. The Inconsistency of Certain Formal Logics // The Annals of Mathematics. — 1935. — July. — Vol. 36, no. 3. — P. 630–630. — doi:10.2307/1968646.

77. Kroshilin A. E., Maidanik V. N. A new approach to calculating parameters of thermohydraulic networks for vapor-gas-liquid flows // Thermal Engineering. — 2007. — May. — Vol. 54, no. 5. — P. 368–374. — doi:10.1134/S0040601507050060.

78. Leroy Xavier. Formal Verification of a Realistic Compiler // Communications of the ACM. — 2009. — Vol. 52, no. 7. — P. 107–115. — doi:10.1145/1538788.1538814.

79. Leveson N. G., Turner C. S. An Investigation of the Therac-25 Accidents // Computer. — 1993. — July. — Vol. 26, no. 7. — P. 18–41. — doi:10.1109/MC.1993.274940.

80. Licata Daniel R., Harper Robert. Canonicity for 2-dimensional type theory // Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — Vol. 47. — Philadelphia, PA, USA : ACM, 2012. — P. 337–348. — doi:10.1145/2103656.2103697.

81. Lindsey C H. A history of ALGOL 68 // History of Programming Languages II / Ed. by Thomas J Bergin, Richard G Gibson. — New York, NY, USA : ACM, 1996. — P. 27–96. — doi:10.1145/234286.1057810.

82. Longo Giuseppe. On Church's formal theory of functions and functionals // Annals of Pure and Applied Logic. — 1988. — November. — Vol. 40, no. 2. — P. 93–133. — doi:10.1016/0168-0072(88)90017-6.

83. Luo Zhaohui. ECC, an extended calculus of constructions // [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. — IEEE Comput. Soc. Press, 1989. — P. 386–395. — doi:10.1109/LICS.1989.39193.

84. Luo Zhaohui. An Extended Calculus of Constructions : Ph.D. thesis / Zhaohui Luo ; University of Edinburgh. — Edinburgh, UK, 1990. — 132 p. — URL: http://www.cs.rhul.ac.uk/~zhaohui/THESIS90.ps (online; accessed: 2015-02-21).

85. MacQueen David B. Using dependent types to express modular structure // Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '86. — ACM Press, 1986. — P. 277–286. — doi:10.1145/512644.512670.

86. Maroneze André. Verified compilation and worst-case execution time : Ph.D. thesis / André Maroneze ; Université de Rennes 1. — 2014. —June. — URL: https://hal.archives-ouvertes.fr/tel-01064869/document (online; accessed: 2015-02-27).

87. Martin-Löf Per. Intuitionistic type theory / Ed. by Giovanni Sambin. — Napoli : Bibliopolis, 1984. — Vol. 1 of Studies in Proof Theory. — iv+91 p. — ISBN: 88-7088-105-9.

88. Matthes Ralph. An induction principle for nested datatypes in intensional type theory. // J. Funct. Program. — 2009. — Vol. 19. — P. 439–468. — doi:10.1017/S095679680900731X.

89. McKusick Marshall Kirk, Neville-Neil George V. The design and implementation of the FreeBSD operating system. — Addison-Wesley Professional, 2004. — 720 p. — URL: https://msdn.microsoft.com/en-us/magazine/dn683793.aspx (online; accessed: 2015-02-21).

90. Mendler Nax Paul. Inductive types and type constraints in the second-order lambda calculus // Annals of Pure and Applied Logic. — 1991. —March. — Vol. 51, no. 1-2. — P. 159–172. — doi:10.1016/0168-0072(91)90069-X.

91. Michaelis Mark. A C# 6.0 Language Preview // MSDN Magazine. — 2014. —October. — Vol. 29, no. 10. — P. 18–26.

92. Milner R. A calculus of communicating systems. — Secaucus, NJ, USA : Springer-Verlag New York, Inc, 1982. — 171 p. — ISBN: 0-387-10235-3. — doi:10.1007/3-540-10235-3.

93. Mitchell John C, Plotkin Gordon D. Abstract types have existential type // ACM Transactions on Programming Languages and Systems. — 1988. —July. — Vol. 10, no. 3. — P. 470–502. — doi:10.1145/44501.45065.

94. Moggi Eugenio. Notions of computation and monads // Information and Computation. — 1991. —July. — Vol. 93, no. 1. — P. 55–92. — doi:10.1016/0890-5401(91)90052-4.

95. Necula George C. Proof-carrying code // Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97. — ACM Press, 1997. — P. 106–119. — doi:10.1145/263699.263712.

96. Nordström Bengt, Petersson Kent, Smith Jan M. Programming in Martin-Löf's Type Theory. — Oxford University Press, 1990. — x+201 p. — URL: http://www.cse.chalmers.se/research/group/logic/book/.

97. Papaspyrou Nikolaos S. A Formal Semantics for the C Programming Language : Doctoral Dissertation / Nikolaos S Papaspyrou ; National Technical University of Athenes. — Athenes, Greece, 1998. — URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.124.5712&rep=rep1&type=pdf (online; accessed: 2015-01-20).

98. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency : Rep. : CSD-SW-TR-2-01 / National Technical University of Athens, School of Electrical and Computer Engineering, Software Engineering Laboratory ; Executor: Nikolaos S Papaspyrou. — Athenes, Greece : 2001. — URL: http://www.softlab.ntua.gr/research/techrep/CSD-SW-TR-2-01.pdf.

99. Park D.M.R. Fixpoint induction and proofs of program properties // Proceedings of the Fifth Annual Machine Intelligence Workshop. — Edinburgh University Press, 1969. — P. 59–78.

100. Paulin-Mohring C. Extracting Fomega's programs from proofs in the calculus of constructions // Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89. — ACM Press, 1989. — P. 89–104. — doi:10.1145/75277.75285.

101. Paulin-Mohring Christine. Inductive Definitions in the System Coq - Rules and Properties // TLCA '93 Proceedings of the International Conference on Typed Lambda Calculi and Applications. — Lecture Notes in Computer Science. — Springer-Verlag London, UK, 1993. — P. 328–345. — doi:10.1007/BFb0037116.

102. Pfenning F, Paulin-Mohring C. Inductively defined types in the Calculus of Constructions // Proceedings of Mathematical Foundations of Programming Semantics. — Springer-Verlag, 1990. — Vol. 442 of Lecture Notes in Computer Science. — P. 209–228. — doi:10.1007/BFb0040259.

103. Pnueli Amir. The temporal logic of programs // 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). — IEEE, 1977. —September. — P. 46–57. — doi:10.1109/SFCS.1977.32.

104. Pollack Robert, Harper Robert. Type Checking, Universe Polymorphism and Typical Ambiguity in the Calculus of Constructions // Proceedings of the International Joint Conference on Theory and Practice of Software Development Barcelona, Spain, March 13–17. — Springer Berlin Heidelberg, 1989. — P. 241–256. — doi:10.1007/3-540-50940-2_39.

105. Post Emil Leon. Finite combinatory processes — formulation 1 // The Journal of Symbolic Logic. — 1936. — Vol. 1. — P. 103–105. — doi:10.2307/2269031.

106. A formal approach to the error localization : Preprint : 169 / IIS SB RAS ; Executor: A.V. Promsky. — Novosibirsk : 2012. — P. 32. URL: http://www.iis.nsk.su/files/preprints/169.pdf (online; accessed: 2015-03-17).

107. Puchkov F, Shapchenko K. Static Analysis Method for Detecting Buffer Overflow Vulnerabilities // Programming and Computer Software. — 2005. — Vol. 31, no. 4. — P. 179–189. — doi:10.1007/s11086-005-0030-8.

108. Quipper / Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross et al. // Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13. — ACM Press, 2013. — P. 333–333. — doi:10.1145/2491956.2462177.

109. The economic impacts of inadequate infrastructure for software testing : Planning Report : 02-3 (7007.011) / National Institute of Standards and Technology ; Executor: RTI : 2002. — URL: http://www.nist.gov/director/planning/upload/report02-3.pdf (online; accessed: 2015-02-21).

110. Rice H. G. Classes of recursively enumerable sets and their decision problems // Transactions of the American Mathematical Society. — 1953. — February. — Vol. 74, no. 2. — P. 358–358. — doi:10.1090/S0002-9947-1953-0053041-6.

111. Rouselakis Yannis. Compilation to quantum circuits for a language with quantum data and control // Federated Conference on. — 2013. — P. 1549 – 1556. — URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6644223.

112. Scalable Three-Dimensional Thermal-Hydraulic Best-Estimate Code BAGIRA / V A Vasenin, M A Krivchikov, A E Kroshilin et al. // Proceedings of 2012 International Congress on Advances in Nuclear Power Plants (ICAPP'12). — Chicago, USA, 2012. — P. 2042–2050. — URL: https://inis.iaea.org/search/search.aspx?orig_q=RN:44065633.

113. Schmidt David A. Denotational Semantics: A Methodology for Language Development. — Boston : Allyn and Bacon, 1986. — ISBN: 0-697-06849-2. — URL: http://www.bcl.hamilton.ie/~barak/teach/F2008/NUIM/CS424/texts/ds.pdf (online; accessed: 2015-02-23).

114. Outline of a Mathematical Theory of Computation : Rep. : PRG02 / Oxford University Computing Laboratory ; Executor: Dana S Scott. — Oxford, England : 1970. — November. — P. 30. URL: http://ropas.snu.ac.kr/~kwang/520/readings/sco70.pdf (online; accessed: 2015-02-21).

115. Self-certification: Bootstrapping certified typecheckers in F* with Coq / Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, Juan Chen // ACM SIGPLAN Notices. — Vol. 47. — ACM, 2012. — P. 571–584. — doi:10.1145/2103621.2103723.

116. Strachey Christopher. Fundamental concepts in programming languages // Higher-order and symbolic computation. — 2000. — P. 11–49. — doi:10.1023/A:1010000313106.

117. Tarjan R. Depth-First Search and Linear Graph Algorithms // SIAM Journal on Computing. — 1972. — June. — Vol. 1, no. 2. — P. 146–160. — doi:10.1137/0201010.

118. Tate Ross. Equality saturation: using equational reasoning to optimize imperative functions : Ph.D. thesis / Ross Tate ; University of California. — San Diego, USA, 2012. — URL: https://escholarship.org/uc/item/3gk5h8jp.pdf (online; accessed: 2015-02-25).

119. The FreeBSD Project. Исходный код ОС FreeBSD 2.0.5. — 1995. — URL: ftp://ftp-archive.freebsd.org/pub/FreeBSD-Archive/old-releases/i386/2.0.5-RELEASE/src/.

120. The FreeBSD Project. Исходный код ОС FreeBSD 8.4. — 2013. — URL: ftp://ftp.freebsd.org/pub/FreeBSD/releases/i386/8.4-RELEASE/src/.

121. The Univalent Foundations Program. Homotopy Type Theory: Univalent Foundations of Mathematics. — Princeton, NJ : Institute for Advanced Study, 2013. — x+465 p. — URL: http://homotopytypetheory.org/book.

122. Thompson Simon. Type theory and functional programming. — Redwood City, CA : Addison Wesley Longman Publishing Co., Inc, 1991. — 388 p. — ISBN: 0-201-41667-0. — URL: http://sources.haskell.cz/TypeTheoryandFunctionalProgramming.pdf.

123. Turing A. M. On Computable Numbers, with an Application to the Entscheidungsproblem // Proceedings of the London Mathematical Society. — 1937. — Vol. s2-42, no. 1. — P. 230–265. — doi:10.1112/plms/s2-42.1.230.

124. Turing A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction // Proceedings of the London Mathematical Society. — 1938. — January. — Vol. s2-43, no. 6. — P. 544–546. — doi:10.1112/plms/s2-43.6.544.

125. Vasenin V. A., Krivchikov M. A. ECMA-335 static formal semantics // Programming and Computer Software. — 2012. — July. — Vol. 38, no. 4. — P. 183–188. — doi:10.1134/S0361768812040056.

126. Viva64. CppCat: Search for bugs in C/C++ code. — 2014. — URL: http://www.cppcat.com/.

127. Viva64. PVS-Studio Static Code Analyzer for C/C++/C++11. — 2014. — URL: http://www.viva64.com/en/pvs-studio/.

128. Voevodsky Vladimir. A very short note on the homotopy lambda-calculus. — 2006. — URL: http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/Hlambda_short_current.pdf.

129. Voevodsky Vladimir. Univalent foundations project. NSF grant application. — 2010. — URL: http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/univalent_foundations_project.pdf (дата обращения: 2015-02-25).

130. Weaver Robert Andrew. The Safety of Software – Constructing and Assuring Arguments : PhD / Robert Andrew Weaver ; Department of Computer Science, University of York. — 2003. — URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.3627&rep=rep1&type=pdf (online; accessed: 2014-10-11).

131. A compact kernel for the calculus of inductive constructions / Andrea Asperti, Wilmer Ricciotti, C. Sacerdoti Coen, Enrico Tassi // Sadhana. — 2009. — Vol. 34, no. 1. — P. 71–144. — doi:10.1007/s12046-009-0003-3.

132. de Bruijn N. G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem // Indagationes Mathematicae (Proceedings). — 1972. — Vol. 75, no. 5. — P. 381–392. — doi:10.1016/1385-7258(72)90034-0.

133. seL4: Formal Verification of an OS Kernel / Gerwin Klein, Kevin Elphinstone, Gernot Heiser et al. // Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. — SOSP ’09. — New York, NY, USA : ACM, 2009. — P. 207–220. — doi:10.1145/1629575.1629596.

134. The semantics of Power and ARM multiprocessor machine code / Jade Alglave, Anthony Fox, Samin Ishtiaq et al. // Proceedings of the 4th workshop on Declarative aspects of multicore programming. — ACM, 2009. — P. 13–24. — doi:10.1145/1481839.1481842.

135. x86-TSO: a rigorous and usable programmer’s model for x86 multiprocessors / Peter Sewell, Susmit Sarkar, Scott Owens et al. // Communications of the ACM. — 2010. — Vol. 53, no. 7. — P. 89–97. — doi:10.1145/1785414.1785443.

136. Ануреев И.С. Операционно-онтологический подход к формальной спецификации языков программирования // Программирование. — 2009. — № 1. — С. 50–60.

137. Ануреев И.С., Марьясов И.В., Непомнящий В.А. Верификация C-программ на основе смешанной аксиоматической семантики // Моделирование и анализ информационных систем. — 2010. — Т. 17, № 3. — С. 5–28.

138. Архипова М.В. Генерация тестов для семантических анализаторов // Вычислительные методы и программирование: новые вычислительные технологии. — 2006. — № 2. — С. 55–70.

139. Бурдонов И.Б. Теория конформности для функционального тестирования программных систем на основе формальных моделей : Диссертация на соискание учёной степени доктора физико-математических наук 05.13.17 / И.Б. Бурдонов ; Институт Системного Программирования Российской Академии Наук. — Москва, 2008. — 434 с.

— URL: http://www.ispras.ru/publications/teoriya_konformnosti_dlya_funktsionalnogo_testirovaniya_programmnykh_sistem_na_osnove_formalnykh_mod.pdf.

140. Васенин В.А., Кривчиков М.А. Модель динамического параллельного исполнения программ // Программирование. — 2013. — № 1. — С. 45–59.

141. Васенин В. А. Модернизация экономики и новые аспекты инженерии программ // Программная инженерия. — 2012. — № 2. — С. 2–17.

142. Васенин В. А., Кривчиков М. А. Статическая семантика стандарта ECMA-335 // Программирование. — 2012. — № 4. — С. 3–16.

143. Васенин В. А., Кривчиков М. А. Формальные модели программ и языков программирования. Часть 1. Библиографический обзор 1930—1989 гг // Программная инженерия. — 2015. — № 5. — С. 10–19.

144. Васенин В. А., Кривчиков М. А. Формальные модели программ и языков программирования. Часть 2. Современное состояние исследований // Программная инженерия. — 2015. — № 6. — С. 24–33.

145. Васенин В. А., Роганов В. А. Технология автоматизированной модификации исходных текстов программ при помощи системы микроправил // Ломоносовские чтения 2013. — Секция механики. — Издательство Московского университета Москва, 2013. — С. 29–29.

146. Водомеров А. Н. Построение формальной модели Т-системы и исследование ее корректности // Вычислительные методы и программирование: новые вычислительные технологии. — 2006. — № 2. — С. 71–78.

147. ГОСТ Р ИСО/МЭК 15408-1-2008. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 1. Введение и общая модель. — 2008.

148. ГОСТ Р ИСО/МЭК 15408-2-2008. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 2. Функциональные требования безопасности. — 2008.

149. ГОСТ Р ИСО/МЭК 15408-3-2008. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 3. Требования доверия к безопасности. — 2008.

150. Ершов Ю.Л. Непрерывные решетки и А-пространства // Доклады Академии Наук СССР. — 1972. — Т. 207, № 3. — С. 523–526.

151. Замулин А.В. Алгебраическая семантика императивного языка программирования // Программирование. — 2003. — № 6. — С. 51–64.

152. Исследование проблем эквивалентности и эквивалентных преобразований программ методами теории схем программ и неклассических логик : отчет о НИР/НИОКР : 94-01-00054-а / Факультет вычислительной математики и кибернетики Московского государственного университета им. М.В.Ломоносова (МГУ ВМиК) ; исполн.: Р.И. Подловченко, И.В. Горская, В.А. Захаров и др. — М. : 1994.

153. Ковалев М.С., Далингер Я.М., Мяготин А.В. Формальная верификация программной реализации алгоритма пирамидальной сортировки на языке СИ-0. // Научно-технические ведомости Санкт-Петербургского государственного политехнического университета. Информатика. Телекоммуникации. Управление. — 2010. — № 103. — С. 83–92.

154. Кондратьев Д.А., Промский А.В. Разработка самоприменимой системы верификации. Теория и практика // Моделирование и анализ информационных систем. — 2014. — № 6. — С. 71–82.

155. Критически важные объекты и кибертерроризм. Часть 1. Системный подход к организации противодействия / О. О. Андреев, В. А. Васенин, К. А. Шапченко и др. ; Под ред. В. А. Васенин. — М. : МЦНМО, 2008. — 391 с. — ISBN: 978-5-94057-416-3.

156. Критически важные объекты и кибертерроризм. Часть 2. Аспекты программной реализации средств противодействия / О. О. Андреев, А. С. Шундеев, С. А. Афонин и др. ; Под ред. В. А. Васенин. — М. : МЦНМО, 2008. — 607 с. — ISBN: 978-5-94057-417-0.

157. Кропачева М.С., Легалов А.И. Формальная верификация программ, написанных на функционально-потоковом языке параллельного программирования // Моделирование и анализ информационных систем. — 2012. — № 5. — С. 81–99.

158. Кудрявцева И.А. Формальная операционная семантика модельных регистровых языков программирования // Новые образовательные стратегии в современном информационном пространстве. — СПб : РГПУ, 2007. — С. 170–176.

159. Кулямин В. В., Петренко А. К. Развитие подхода к разработке тестов UniTESK // Труды Института системного программирования РАН. — 2014. — Т. 26, № I. — С. 9–26.

160. Липаев В.В. Программная инженерия. Методологические основы. — М. : ТЕИС, 2006. — 608 с. — ISBN: 5-7598-0424-3.

161. Ляпунов А. А. О логических схемах программ // Проблемы кибернетики: сборник статей. — 1958. — № 1. — С. 46–74.

162. Малаховски Я. М., Корнеев Г. А. Применение зависимых систем типов со структурной индукцией для верификации реактивных программ // Научно-технический вестник информационных технологий, механики и оптики. — 2012. — № 6. — С. 63–67. — URL: http://ntv.ifmo.ru/file/journal/120.pdf#page=63 (дата обращения: 2015-03-23).

163. Марков, А.А. Теория алгорифмов. Труды математического института имени В.А. Стеклова № XLII. — Москва, Ленинград : Изд-во АН СССР, 1954. — 377 с.

164. Мешвелиани С. Д. О зависимых типах и интуиционизме в программировании математики // Программные системы: теория и приложения. — 2014. — Т. 5, № 3. — С. 27–50. — URL: ftp://95.129.137.165/rented/psta/www/read/psta2014_3_27-50.pdf (дата обращения: 2015-03-23).

165. Миграция от MPI к платформе OpenTS: эксперимент с приложениями PovRay и ALCMD / С. М. Абрамов, И. М. Загоровский, М. Р. Коваленко и др. // Международная конференция ”Программные системы: теория и приложения”. — Наука, Физматлит, 2006. — С. 265–275.

166. На пути к верификации C#-программ: трехуровневый подход / В.А. Непомнящий, И.С. Ануреев, И.В. Дубрановский, А.В. Промский // Программирование. — 2006. — № 4. — С. 4–20.

167. Недеструктивный переход от последовательных программ к параллельным для суперЭВМ / В. А. Васенин, А. Е. Крошилин, В. Е. Крошилин, В. А. Роганов // Труды XVIII Байкальской Всероссийской конференции. — Т. 3 из Информационные и математические технологии в науке и управлении. — Институт систем энергетики им.Мелентьева СО РАН г.Иркутск, 2013. — С. 224–230.

168. Непомнящий В.А., Ануреев И.С., Промский А.В. На пути к верификации C программ. Аксиоматическая семантика C-kernel // Программирование. — 2003. — № 6. — С. 65–80.

169. Нис З.Я. Преобразования программ: контроль семантической корректности // Известия высших учебных заведений. Северо-Кавказский регион. Серия: Естественные науки. — 2010. — № 1. — С. 18–21.

170. Пирс Бенджамин. Типы в языках программирования. — Лямбда пресс, Добросвет, 2011. — 656+xxiv с. — ISBN: 978-5-9902824-1-4.

171. Подловченко Р.И. А. А. Ляпунов и А. П. Ершов в теории схем программ и развитие ее логических концепций // Андрей Петрович Ершов — ученый и человек / Под ред. А.Г. Марчук. — Новосибирск : Издательство СО РАН, 2006. — URL: http://www.computer-museum.ru/books/n_ershov/2_ershov_teoria.htm (дата обращения: 2014-04-04).

172. Подловченко Р.И., Молчанов А.Э. Разрешимость эквивалентности в перегородчатых моделях программ // Моделирование и анализ информационных систем. — 2014. — № 2. — С. 56–70.

173. Подход UniTesK к разработке тестов / И.Б. Бурдонов, А.С. Косачев, В.В. Кулямин, А.К. Петренко // Программирование. — 2003. — № 6. — С. 25–43.

174. Пресс-служба Роскосмоса. Основные положения Заключения Межведомственной комиссии по анализу причин нештатной ситуации, возникшей в процессе проведения летных испытаний космического аппарата «Фобос-Грунт». — 2012. — URL: http://www.roscosmos.ru/18126/ (дата обращения: 2014-10-21).

175. Распараллеливание теплогидравлического расчетного кода CMS в составе полномасштабной суперкомпьютерной модели «Виртуальная АЭС» / В А Васенин, И С Астапов, В А Роганов и др. // Ломоносовские чтения 2013. Секция механики. — Москва : Изд-во Московского университета, 2013. — С. 27–28.

176. Средства суперкомпьютерных систем для работы с агент-ориентированными моделями / В. А. Васенин, В. Л. Макаров, А. Р. Бахтизин и др. // Программная инженерия. — 2011. — № 3. — С. 2–14.

177. Т-подход к автоматизированному распараллеливанию программ: идеи, решения, перспективы / В. А. Васенин, Е. А. Степанов, И. М. Конев, А. Н. Водомеров ; Ed. by В. А. Садовничий. — МЦНМО, 2008. — 460 p. — ISBN: 978-5-94057-414-9.

178. Шилов Н.В. Пример верификации в проекте F@BOOL@, основанном на булевских решателях // Моделирование и анализ информационных систем. — 2010. — № 4. — С. 111–124.

179. Шилов Н.В. Основы синтаксиса, семантики, трансляции и верификации программ: учебное пособие. — Новосибирск : НГУ, 2011. — 292 с.

180. Янов Ю. И. О логических схемах алгоритмов // Проблемы кибернетики: сборник статей. — 1958. — № 1. — С. 75–127.

Работы автора по теме диссертации

181. Кривчиков, М.А. Статическая формальная семантика стандарта ECMA-335 / В.А. Васенин, М.А. Кривчиков // Программирование. — 2012. — №4. — С. 3–16. Английский перевод: V.A. Vasenin, M.A. Krivchikov. ECMA-335 Static Formal Semantics // Programming and Computer Software. — 2012. — Vol. 38, no. 4. — P. 183–188. doi:10.1134/S0361768812040056
182. Кривчиков, М.А. Распараллеливание расчетного кода улучшенной оценки «БАГИРА» для моделирования трехмерной теплогидродинамики многофазных сред в составе полномасштабной суперкомпьютерной модели «Виртуальная АЭС» / В.А. Васенин, М.А. Кривчиков, А.Е. Крошилин, В.Е. Крошилин, А.Д. Рагулин, В.А. Роганов // Программная инженерия. — 2012. — № 6. — С. 15–23.
183. Кривчиков, М.А. Модель динамического параллельного исполнения программ / В.А. Васенин, М.А. Кривчиков // Программирование. — 2013. — №3. — С. 45–59. Английский перевод: A Model of Dynamical Concurrent Program Execution / V.A. Vasenin, M.A. Krivchikov // Programming and Computer Software. — 2013. — Vol. 39, no. 1. — P. 1–9. doi: 10.1134/S0361768813010076
184. Кривчиков, М.А. Формальные модели программ и языков программирования. Часть 1. Библиографический обзор 1930 – 1989 гг / В.А. Васенин, М.А. Кривчиков // Программная инженерия. — 2015. — № 5. — С. 10–19.
185. Кривчиков, М.А. Формальные модели программ и языков программирования. Часть 2. Современное состояние исследований / В.А. Васенин, М.А. Кривчиков // Программная инженерия. — 2015. — №6. — С. 24–33.
186. Кривчиков, М.А. Scalable Three-Dimensional Thermal-Hydraulic Best-Estimate Code BAGIRA / V.A. Vasenin, M.A. Krivchikov, A.E. Kroshilin, V.E. Kroshilin, V.A. Roganov // Proceedings of 2012 International Congress on Advances in Nuclear Power Plants (ICAPP'12). — Chicago, USA, 2012. — P. 2042–2050.
187. Кривчиков, М.А. К формальному описанию программ для распределенной вычислительной среды грид-архитектуры / В.А. Васенин, М.А. Кривчиков // Ломоносовские чтения. Тезисы докладов научной конференции. Секция механики. 16-25 апреля. — Изд-во Московского университета — 2012. — С. 32–33.
188. Кривчиков, М.А. Распараллеливание теплогидравлического расчетного кода CMS в составе полномасштабной суперкомпьютерной модели «Виртуальная АЭС» / В.А. Васенин, И.С. Астапов, В.А. Роганов, В.Н. Майданик, М.А. Кривчиков // Ломоносовские чтения. Тезисы докладов. Секция механики. 15–23 апреля 2013 г. — Изд-во Московского университета. — 2013. — С.27–28.

189. Кривчиков, М.А. Предметно-ориентированные языки с заданной формальной семантикой на основе лямбда-исчисления с зависимыми типами / В.А. Васенин, М.А. Кривчиков // Международная конференция «Мальцевские чтения». Тезисы докладов. [Электронный ресурс]. — URL: http://www.math.nsc.ru/conference/malmeet/14/Malmeet2014.pdf . 2014. — С.25–25.
190. Кривчиков, М.А. Приложение одной разновидности типизированного лямбда-исчисления к построению формальных моделей программ / В.А. Васенин, М.А. Кривчиков // Ломоносовские чтения. Тезисы докладов. Секция механики. 14–23 апреля 2014 г. — Изд-во Московского университета. — 2014. — С. 39–39.
191. Кривчиков, М.А. Языково-ориентированное программирование для формальной верификации программного обеспечения / В.А. Васенин, М.А. Кривчиков. Материалы четвертой научно-практической конференции «Актуальные проблемы системной и программной инженерии». Сборник научных трудов. — Изд-во НИУ ВШЭ. — 2015. — С.30–31.

Приложение А

Данные по состоянию программной инженерии

А.1. Увеличение размера исходного кода программного обеспечения

Для анализа изменения размера исходного кода при развитии программного обеспечения были выбраны следующие два программных продукта с открытым исходным кодом: ядро операционной системы FreeBSD [89] и широко используемая утилита сжатия данных GNU gzip [56]. Анализ проводился с использованием утилиты для подсчёта количества строк исходного кода CLOC версии 1.58 [37]. Указанное далее количество строк представляет собой общее количество строк исходного кода на языках C и C++ без учёта комментариев и пустых строк.

Исходный код ядра операционной системы FreeBSD версии 2.0.5 [119] (выпущена в 1995 г.) для платформ на базе архитектуры i386 состоит из 1066 файлов с исходным кодом на языке C общим объёмом более 280 тыс. строк. Исходный код ядра FreeBSD версии 8.4 [120] (выпущена в 2013 г.) состоит из 8860 файлов общим объёмом более 3360 тыс. строк. Таким образом, за 18 лет количество файлов с исходным кодом выросло более чем в 8 раз, а общее количество строк исходного кода — более чем в 11 раз. Необходимо отметить, что такое изменение в объёме исходного кода монолитного ядра операционной системы может быть объяснено, в первую очередь, включением в состав ядра новых драйверов устройств, которые появились за рассматриваемый промежуток времени.

Исходный код утилиты сжатия данных GNU gzip версии 1.2.4 [49] (выпущена в 1993 г.) содержится в 34 файлах общим объёмом более 5.8 тыс. строк. Исходный код утилиты GNU gzip версии 1.6 [50] (выпущена в 2013 г.) содержится в 216 файлах общим объёмом более 42 тыс. строк. Таким образом, за 20 лет количество файлов с исходным кодом выросло более чем в 6 раз, а общее количество строк исходного кода — более чем в 7 раз. В отличие от предыдущего примера, утилита реализует единственный алгоритм сжатия, который не претерпевал существенных изменений за 20 лет (это не относится к реализации алгоритма).

Приведённые примеры показывают, что в процессе длительной эксплуатации программного обеспечения общий объем исходного кода существенно возрастает (более чем в 5 раз для двух примеров). Отметим также, что, учитывая длительную историю развития программного обеспечения, за 18-20 лет состав команды разработчиков существенно изменился.

A.2. Плотность дефектов в программных продуктах с открытым исходным кодом

Компания Coverity ежегодно публикует отчёт о исследовании крупных продуктов с открытым исходным кодом на предмет наличия дефектов, которые способна обнаруживать актуальная версия программного продукта Coverity Scan. В число объектов исследования входили, в частности, следующие распространённые и широко используемые продукты: ядра ОС Linux, FreeBSD и NetBDS; серверы Apache, Samba, ntp, Postfix; СУБД PostgreSQL; библиотеки curl, OpenSSL. настоящем приложении рассматриваются данные, представленные в отчётах за 2008–2012 гг. [31–36]. Прежде чем будет представлена информация о результатах необходимо сделать следующие замечания о методике тестирования:

- ежегодно исследование проводилось на в общем случае не совпадающих множествах программных продуктов;
- ежегодно исследование проводилось с использованием актуальной версии средства статического анализа, при этом множество обнаруживаемых дефектов для различных версий различается.

С учетом представленных выше замечаний, необходимо интерпретировать результаты исследований независимо за различные годы. Оценки на количество дефектов в проектах необходимо рассматривать как оценки снизу (минимальное обнаруженное число дефектов). Определённые классы дефектов могли избежать обнаружения, что подтверждается на практике: в 2014 году в библиотеке OpenSSL была обнаружена критическая уязвимость CVE-2014-0160, связанная с выходом за границы массива, в результате которой оказались скомпрометированы закрытые данные большого количества крупнейших веб-сайтов. Уязвимыми в результате атаки оказывались, в том числе, закрытые ключи, используемые в рамках протокола TLS для идентификации узла, а эксплуатация уязвимости была возможна как на серверных, так и на клиентских узлах. При этом актуальная на момент обнаружения уязвимости версия статического анализатора Coverity не была способна её обнаружить (http://blog.regehr.org/archives/1128).

Основным рассматриваемым в настоящем приложении показателем является плотность дефектов (среднее количество обнаруженных дефектов на тысячу строк кода).

В 2008 г. был проведён анализ 250 проектов с открытым исходным кодом общим объёмом более 50 млн строк. Минимальный размер составил 6493, максимальный — 5050450, средний — 425179 строк кода. Среднее количество дефектов на тысячу строк кода составило 0,30.

В 2009 г. был проведён анализ 280 проектов с открытым исходным кодом общим объёмом более 60 млн строк. Среднее количество дефектов на тысячу строк кода составило 0,25.

В 2010 году был проведён анализ 291 проекта с открытым исходным кодом общим объёмом более 61 млн строк. Среднее количество дефектов на тысячу строк кода составило 0,81.

В 2011 году был проведён анализ 45 крупных проектов с открытым исходным кодом общим объёмом более 37 млн строк. Среднее количество дефектов на тысячу строк кода составило 0,45.

В 2012 году был проведён анализ 118 проектов с открытым исходным кодом общим объёмом более 68 млн строк. Среднее количество дефектов на тысячу строк кода составило 0,69.

В 2013 году был проведён анализ 741 проекта с открытым исходным кодом общим объёмом более 250 млн строк. Среднее количество дефектов на тысячу строк кода составило 0,59.

Сводная статистика по плотности дефектов за все годы представлена в последнем на настоящее время отчёте [36] и в таблице A.1.

Таблица A.1: Сводная статистика по плотности дефектов, обнаруженных в рамках исследования Coverity Scan в 2008–2012 гг.

Год	2008	2009	2010	2011	2012	2013
Дефектов на 1 тыс. строк кода	0,30	0,25	0,81	0,45	0,69	0,59

A.3. Количество языков программирования, используемых при разработке программных продуктов с открытым исходным кодом

На настоящее время крупнейшими по количеству пользователей поставщиками услуг репозиториев систем контроля версий для программных продуктов с открытым исходным кодом являются: GitHub (`https://www.github.com`), SourceForge

(http://sourceforge.net) и Launchpad (http://launchpad.net). При этом для репозиториев GitHub предоставляется автоматически собранная на основе анализа файлов информация об использованных в составе исходного кода языках программирования. Для двух других поставщиков подобная информация вводится вручную владельцем репозитория, поэтому является менее достоверной и неструктурированной.

В настоящем приложении приведены результаты анализа состава языков программирования, используемых в 1256 репозиториях GitHub, наиболее популярных по количеству голосов пользователей (stars в терминологии GitHub) и по количеству производных репозиториев (forks в терминологии GitHub). Для получения данных использовался веб-сервис, предоставляемый компанией GitHub бесплатно для зарегистрированных пользователей. Анализ проводился автором настоящей работы 13 октября 2013 г. Поскольку веб-сервис не предоставляет исторических данных, результаты анализа, проведённого по аналогичной методике в другое время могут отличаться от представленных в настоящем приложении. Далее приведены результаты, методика анализа и ссылка на сохранённые исходные данные, на основе которых возможно повторение полученных результатов.

Результаты анализа представлены в таблице А.2. В группе столбцов «с JavaScript» представлены данные по всем проанализированным репозиториям, в группе столбцов «без JavaScript» представлены данные по проанализированным репозиториям без учёта репозиториев без языков программирования и репозиториев, в которых один из языков программирования — JavaScript (подробнее о причинах разделения см. методику анализа). В строке «без ЯП» указаны количество и доля репозиториев, для которых языки программирования не были указаны или были отфильтрованы.

Таблица А.2: Результаты анализа по количеству языков программирования, используемых при разработке программных продуктов с открытым исходным кодом

	С JavaScript		Без Javascript	
	Кол-во, шт.	Доля, %	Кол-во, шт.	Доля, %
Без ЯП	74	5,9	—	—
Один ЯП	509	40,52	239	51,62
Два ЯП	337	26,83	118	25,49
Три и более ЯП	336	26,75	106	22,89
Всего	1256	100	463	100

Методика анализа. С использованием веб-сервиса GitHub были получены исходные данные в формате JSON для двух источников — списков наиболее популярных репозиториев по количеству голосов пользователей и производных репозиториев,

соответственно. Такие данные по состоянию на момент проведения анализа доступны по адресу, указанному далее под заголовком «исходные данные». Затем было выполнено слияние двух источников: списки были объединены, после чего были исключены дубликаты (по идентификатору, поле «id» в исходных данных). Далее, для каждого репозитория в поле «languages_url» был указан адрес, по которому доступна статистика по использующимся в репозитории языкам программирования, который был использован для получения таких данных. Из полученных списков были удалены упоминания языков «Shell» и «CSS». Языки командной оболочки, обозначаемые в исходных данных «Shell», часто используется для задания системы сборки проектов. Учёт таких систем сборки в составе используемых языков не позволил бы получить корректные данные. Язык каскадных таблиц стилей CSS используется для настройки режимов отображения документов, написанных на языке разметки HTML. При учёте таких языков все репозитории, содержащие документацию в формате HTML, попали бы в категорию «Три и более ЯП». После такой фильтрации был проведён подсчёт. Репозитории, для которых не был указан язык программирования, либо был указан единственный язык «HTML» или «Text» были отнесены к категории «без ЯП». Ручная проверка показала, что в состав таких репозиториев входили, в первую очередь, тексты книг и наборы документации. Репозитории, для которых были указаны один, два, три и более языков программирования были отнесены к соответствующей категории.

При ручной проверке было принято решение составить второй набор результатов, для которого из исходных данных исключались репозитории, содержащие код на языке JavaScript. Этот язык часто используется при разработке веб-приложений совместно с языками HTML и CSS. В связи с особенностями этого языка, на настоящее время широкое распространение получают также основанные на нём предметно-ориентированные языки и диалекты (такие как CoffeeScript, Dart и TypeScript). Многие их таких языков учитываются автоматизированной системой GitHub как различные. Таким образом, группа результатов «без JavaScript» отражает статистику по проектам, за исключением клиентских библиотек и оболочек веб-разработки.

Исходные данные по составу репозиториев, наиболее популярных по количеству голосов пользователей и количеству производных репозиториев, доступны по следующему URL: https://drive.google.com/file/d/0BwEa50ni6h2NUnhnN2JKZnplS2c/edit?usp=sharing. В доступном по приведённому URL файле содержится zip-архив, в котором находятся файлы search-stars.json и search-forks.json, которые содержат список наиболее популярных репозиториев по количеству голосов пользователей и количеству производных репозиториев по состоянию на 13 октября 2013 г.

Приложение В

Исходный код для системы Coq

В представленном далее исходном коде для системы Coq выполняется верификация определения коиндуктивного типа потока как высшего индуктивного типа.

Аксиоматически заданы следующие определения, свойственные для предложенной автором разновидности λ-исчисления с зависимыми типами:

HIT — определение высшего индуктивного типа;

elN — введение элемента высшего индуктивного типа;

eqI — фрагмент введения элемента типа идентичности высшего индуктивного типа (рассматривается только введение элемента типа I);

Hel — удаление высшего индуктивного типа;

itoe_pair — удаление типа идентичности для типа пар;

pair_eta — η-эквивалентность для первого элемента типа пар.

Требуется построить коиндуктивный тип потоков, который, в рамках теоретико-категориальной семантики коиндуктивных типов как терминальных коалгебр, должен обладать следующими свойствами для некоторого данного типа A:

S_F X ≡ A × X — эндофунктор на типах, задающий тип потоков;

HStream : Type — конструктор типа;

HSana : Π (S : Type) . (ΠS.S_F S) . S . HStream — анаморфизм (гомоморфизм из любой S_F-коалгебры в HStream), показывает терминальность HStream;

HSout : Π(hs : HStream) . S_F HStream — удаление потока; морфизм, требуемый определением коалгебры.

```
Require Import Utf8_core.
Require Import Utf8.

Notation "x × y" := (x * y)%type (at level 70, right associativity) : type_scope.

Axiom HIT : ∀ (A : Type) (I : A → A → Type), Type.
Axiom elN : ∀ {A : Type} (I : A → A → Type) (a : A), HIT A I.
Axiom eqI : ∀ {A : Type} {I : A → A → Type} (a b : A) (i : I a b), elN I a = elN I b.
Axiom Hel : ∀ {A : Type} {I : A → A → Type}
    (C : HIT A I → Type) (f : ∀ (a : A), C (elN I a))
```

```
  (β : ∀(a b : A) (i : I a b), eq_rect (elN I a) C (f a) (elN I b)
      (eqI a b i) = f b)
  (h : HIT A I), C h.

Axiom itoe_pair : ∀ {A B : Type} {a b : A × B} (e : a = b),
  (fst a = fst b × snd a = snd b).
Axiom pair_eta : ∀ {A B C : Type} (a : A × B) (c : C),
  (let (aa, _) := a in (aa, c)) = (fst a, c).

Section StreamH.
Variable A : Type.
Definition S_F (X : Type) := A × X.
Definition S_T (S : Type) (n : nat) := nat_iter n S_F S.
Definition S_r : ∀ {X : Type} (n : nat), S_T X n → S_T True n.
  intros X n. unfold S_T. induction n ; simpl ; auto.
  unfold S_F at 1 3. intuition.
Defined.



Eval compute in S_r.


Print S_r.


Definition HS_T := { S : Type & { s : S & ∀ (s : S) (n : nat), S_T S n } }.


Definition HS_I (i j : HS_T) : Type.
  unfold HS_T in *. destruct i as [Ti si]. destruct j as [Tj sj].
  destruct si as [si1 si2]. destruct sj as [sj1 sj2].
  exact (∀ (n : nat), S_r n (si2 si1 n) = S_r n (sj2 sj1 n)).
Defined.
Print HS_I.


Definition HStream := HIT HS_T HS_I.


Program Definition HSana (S : Type) (a : S → S_F S) (s : S) : HStream :=
 elN _ (existT _ S (existT _ s _)).
```

```
Next Obligation.
   intros ; induction n ; unfold S_T ; simpl in * ; intuition ;
   unfold S_F in *; intuition.
Defined.

Program Definition HSout (hs : HStream) : S_F HStream := Hel _ (λ (a : HS_T),
let (x, s) := a in
let (x0, s0) := s in
(λ sx : S_T x 1, let (a0, _) := sx in (a0, hs)) (s0 x0 1)) _ hs.

Next Obligation.
   intros.  unfold eq_rect. destruct (eqI a b i). destruct a. destruct b.
   destruct s. destruct s0.
   Print HS_I.
   unfold HS_I in i.
   assert (e := i 1). unfold S_r in e. simpl in e. unfold prod_rect in e.
   unfold S_T in *. Check (let (a, _) := s x1 1 in (a, I)).
   rewrite pair_eta in e. rewrite pair_eta in e. apply itoe_pair in e.
   simpl in e. assert (e_fst := fst e).
   rewrite pair_eta. rewrite pair_eta. rewrite e_fst. reflexivity.
Defined.
Print HSout_obligation_1.
End StreamH.
```

Приложение С

Формальная модель вычислений с плавающей точкой

В настоящем приложении представлена версия статьи С.В. Антонова, М.А. Кривчикова «Формальная модель вычислений с плавающей точкой на основе лямбда-исчисления с зависимыми типами», выход которой запланирован в журнале «Программная инженерия», №9, 2015. Вклад автора настоящей диссертации заключается в обобщении и адаптации модели для широкого класса реализаций лямбда-исчисления с зависимыми типами, в том числе — для разновидности, представленной в главах 2, 3. Список литературы приведён в заключительной части статьи. Исходный код модели для среды Coq доступен по следующему URL: https://github.com/antonovsergey93/floating-point-model

С.1. Введение

На настоящее время актуальны задачи, связанные с формальной верификацией унаследованных крупных программных комплексов математического моделирования физических процессов, происходящих при функционировании сложных объектов критически важных инфраструктур [1]. При реализации таких комплексов используются вычисления с плавающей точкой. В рамках процессов верификации таких комплексов возникает необходимость доказательства корректности реализации используемых численных методов. Такие доказательства осложняются особенностями выполнения вычислений над числами с плавающей точкой. В частности, в рамках доказательств сходимости и устойчивости численных методов, как правило, рассматривается погрешность, которая вносится во входные данные. Погрешность, которая вносится при замене точных арифметических операций над действительными числами на операции над числами с плавающей точкой, при этом не принимается во внимание, хотя зачастую является причиной ошибок [2].

В настоящей статье представлены результаты работы по построению формальной модели чисел с плавающей точкой стандарта IEEE 754 [3] и операций над ними в среде Coq [4]. Формальная модель была построена с целью получения метода анализа вычислительной погрешности операций над числами с плавающей точкой в рамках процессов формальной верификации программ. При этом предполагается, что формальная верификация проводится методами дедуктивной верификации, использующими формальную модель на основе λ-исчисления

с зависимыми типами. Разработанная авторами модель является исполнимой, что позволяет получать с её помощью результаты вычислений в среде Coq.

С целью применения модели в процессе анализа фрагментов исходного кода существующего программного комплекса, для модели чисел с плавающей точкой авторами был предложен интерфейс и язык арифметических выражений. С использованием таких средств могут быть исследованы свойства кода, которые проявляются при использовании различных реализаций арифметики на приближениях действительных чисел. В дополнение к модели чисел с плавающей точкой, такой интерфейс был реализован авторами также для рациональных чисел с операцией приближённого вычисления арифметического квадратного корня с произвольной точностью.

Кроме того, ориентируясь на основные направления дальнейших исследований, отмеченные в заключении настоящей статьи, авторы проанализировали модель на предмет возможности её адаптации к другим разновидностям λ-исчисления с зависимыми типами, отличным от той, которая используется в среде Coq. Результатом этой работы стала адаптация модели к разновидности λ-исчисления с зависимыми типами, представленной в [5].

C.2. Стандарт IEEE-754

Стандарт IEEE 754-2008 [3] является основным стандартом, определяющим вычисления с плавающей точкой. В частности, этот стандарт поддерживается большим количеством аппаратных реализаций вычислений над числами с плавающей точкой. Кроме того, этот стандарт (а также его предыдущие редакции и редакции стандарта ISO/IEC/IEEE 60559, идентичные по содержанию IEEE 754) используется в спецификациях распространённых языков программирования, например, таких как C, C# или Fortran.

Стандарт IEEE 754-2008 определяет следующие аспекты вычислений над числами с плавающей точкой:

- форматы чисел с плавающей точкой в двоичной и десятичной системе счисления для вычислений и обмена данными;
- операции сложения, вычитания, умножения и деления чисел с плавающей точкой, а также вычисления арифметического квадратного корня из числа с плавающей точкой
- операции сравнения, реализующие отношение порядка на числах с плавающей точкой;
- операции прямого и обратного преобразования целого числа в число с плавающей точкой;
- операции преобразования между форматами чисел с плавающей точкой;
- операции преобразования числа с плавающей точкой из двоичной системы счисления в десятичную и обратно.

Следует отметить, что в сферу ответственности стандарта IEEE 754-2008 и настоящей модели не входят форматы представления целых чисел и способы интерпретации знака и мантиссы не-чисел (NaN).

Число с плавающей точкой в формате IEEE 754-2008 представляется как тройка (рис. C.1): *знак* (sign), принимающий значение 1, если число отрицательно и 0 — в противном случае;

порядок (exp) — степень основания системы счисления, используемая при вычислении значения числа, при этом порядок хранится со смещением на константу (bias), выбранную таким образом, чтобы диапазон смещённого порядка был неотрицательным; *мантисса* (significant) — дробная часть числа.

Изображение распространяется под лицензией CC BY-SA 3.0 Fresheneesz at the English Wikipedia project, исходный вариант доступен по URL http://en.wikipedia.org/wiki/File:Float_example.svg

Рис. C.1: Пример числа с плавающей точкой одинарной точности

Мантисса всегда представлена в нормализованном виде — бит, стоящий до точки всегда должен быть равен единице (т.е. число с плавающей точкой представляется в виде `1,**...*`, где знаком `*` обозначен произвольное значение бита). Поэтому такой единичный бит, стоящий до точки не хранится в памяти компьютера. Этот бит называется неявным ведущим битом. Поскольку неявный ведущий бит не занимает места в памяти, это позволяет увеличить максимальную точность чисел.

В стандарте зарезервированы множества значений, обозначающих специальные числа с плавающей точкой — положительная и отрицательная бесконечности и не-числа (NaN). Такие значения возникают в результате операций, результат которых не определён, например, при делении на ноль или извлечении квадратного корня из отрицательного числа. Бесконечность записывается в виде порядка, состоящего из единиц и нулевой мантиссы. Значение NaN записывается в виде порядка состоящего из единиц и ненулевой мантиссы.

По заданной тройке из знака, порядка и мантиссы значение числа находится следующим образом: $(-1)^{\text{sign}} \times 2^{(\text{exp} - \text{bias})} \times 1.\text{significant}$.

C.3. Разработка модели чисел с плавающей точкой

В качестве типа данных для хранения одного бита был выбран тип встроенный в систему Coq тип `bool`. Для хранения последовательности бит фиксированной длины используется тип вектора с элементами типа `bool`. Такой выбор обусловлен наличием стандартного модуля `Bvector` (вектор с элементами типа `bool`) в Coq. В этом модуле реализованы стандартные функции работы с векторами, которые используются в настоящей работе.

Введём базовые параметры модели, а именно: битовую размерность экспоненты и мантиссы (`lenExp` и `lenSign` соответственно); сдвиг и максимальное значение экспоненты (`shiftExp` и `maxExp`, соответственно). С использованием этих параметров модель числа с плавающей точкой записывается в Coq следующим, представленным в листинге 9, образом.

Для модели, согласно стандарту, реализованы следующие операции:

```
Record fp_num : Set := mkFp
{
 sign : bool;
 exp : Bvector lenExp;
 significant : Bvector lenSign
}.
```

Листинг 9: Модель числа с плавающей точкой

- операции сравнения (больше, меньше, равно и пр.; основная реализация содержится в функции `compare`);
- арифметические операции (сложение, вычитание, умножение, деление, арифметический квадратный корень из числа; основная реализация содержится в функциях `sumFp`, `diffFp`, `multFp`, `divideFp`, `sqrtFp`, соответственно);
- операции преобразования с потерей точности между значениями модели чисел и другими представлениями, входящими в стандартную библиотеку, а именно — целыми и рациональными числами.

С целью унификации вычислений на числах с плавающей точкой и на рациональных числах был предложен интерфейс модели арифметики на приближениях действительных чисел. Интерфейс задаёт тип числа, арифметические операции, операции сравнения и операции конвертации над элементами типа числа. Определение интерфейса в системе Coq представлено в листинге 10.

```
Record ApproxArith := mkApproxArith {
 Num : Set;
 sum  : Num -> Num -> Num;
 diff : Num -> Num -> Num;
 mult : Num -> Num -> Num;
 div  : Num -> Num -> Num;
 sqrt : Num -> Num;
 from_rational : Q -> Num;
 to_rational : Num -> Q;
 lt : Num -> Num -> bool;
 le : Num -> Num -> bool;
 eq : Num -> Num -> bool;
 ge : Num -> Num -> bool;
 gt : Num -> Num -> bool
}.
```

Листинг 10: Интерфейс модели арифметики на приближениях действительных чисел

Авторами разработана реализация интерфейса для чисел с плавающей точкой и для рациональных чисел с операцией приближённого вычисления квадратного корня. Поскольку

стандартные модули Coq не поддерживают вычисление арифметического квадратного корня с заданной точностью для рациональных чисел, эта арифметическая операция была реализована авторами. В качестве алгоритма был выбран метод Герона.

Для сравнения вычислительной погрешности работы программы на числах с плавающей точкой и рациональных числах, был построен простейший язык арифметических выражений. Язык поддерживает следующие операции: сложить/вычесть/умножить/разделить два числа; вычислить корень из числа; конвертировать число в/из рационального; условный оператор; получить значение переменной; записать значение в переменную. Программа на таком языке представляется как список записей, состоящих из элементов простейшего языка. Динамическая семантика языка арифметических выражений реализована в терминах интерпретатора. Интерпретатор получает на вход следующие параметры: число переменных в коде; вектор с исходными значениями переменных; код программы, представленный в виде элемента статической семантики языка арифметических выражений; реализацию интерфейса арифметики чисел. Результатом работы интерпретатора является вектор с конечными значениями переменных.

C.4. Тестирование модели

Для тестирования модели был выбран фрагмента кода, входящий в состав программного комплекса моделирования теплогидродинамических процессов в первом и втором контурах атомных реакторов серии ВВЭР [1]. Выбранный фрагмент кода предназначен для определения наличия прямого и обратного потока жидкости в компонентах моделируемой системы. Фрагмент кода принимает на вход следующие параметры: *p_out*, *p_in_out*, *p_in_in* — показатели давления (Па); *d1, d2* — плотность жидкости на входе и выходе компонента (кг/м^3); *dtt* — разрешение модели по времени (с). Выходными данными являются *part1* и *part2* — прямые и обратные потоки, а также *nf* — наличие потока в компоненте.

Исследуемая функция является достаточно компактной (32 строки исходного кода на C), однако содержит несколько различных операций над числами с плавающей точкой, в том числе — операции извлечения арифметического квадратного корня. Для тестирования исходный код функции был переписан на языке среды Coq, а также в форме статической семантики простого языка вычислений с плавающей точкой, определённого в предыдущем разделе.

Для тестирования использовалась ЭВМ с процессором Intel Core i5-2410M (2300MHz), оперативной памятью объёмом 6Gb DDR3 1333MHz CL11. На установлена ОС Ubuntu версии 14.04.3 LTS (64-bit), система интерактивного доказательства теорем Coq версии 8.4pl3 и компилятор GCC версии 4.8.2.

В таблице C.1 представлены результаты сравнения результатов вычисления модели с аппаратной реализацией чисел с плавающей точкой. Входные данные были подобраны таким образом, чтобы покрывать все ветви условных операторов в исследуемой функции. С такими входными данными выполнялся исходный вариант кода на языке C, скомпилированный с

Таблица C.1: Сравнение результатов
вычислений модели с аппаратной реализацией чисел с плавающей точкой

p_out	p_in_out	p_in_in	d1	d2	dtt	*разность part1*	*разность part2*	*nf*
100	150	200	961	1024	0.01	$4.33e^{-19}$	0	1
128	130	130	976	953	0.08	$2.16e^{-19}$	0	1
400	300	200	1056	968	0.1	0	0	1
350	349	350	953	976	0.04	0	0	1
375	375	400	1024	961	0.09	0	0	0
333	300	400	998	1017	0.06	0	0	0
350	400	300	961	1024	0.08	0	$3.47e^{-18}$	1
325	350	310	1024	961	0.04	0	8.673^{-19}	1

Таблица C.2: Сравнение относительной погрешности
вычислений с использованием модели чисел с плавающей точкой и модели
вычислений на рациональных числах с результатами, полученными аналитически

Входные данные			Рациональные числа			Числа с плавающей точкой		
p_out	p_in_out	p_in_in	part1	part2	время	part1	part2	время
100	150	200	$1.1e^{-58}$	0	918	$5e^{-17}$	0	1
400	300	350	0	$6e^{-69}$	512	0	$9e^{-18}$	1
350	300	400	0	0	1.5	0	$8e^{-18}$	0.5
350	400	300	$4e^{-58}$	$3e^{-69}$	1442	$5e^{-17}$	$6e^{-17}$	3

ключом `-mfpmath=sse` и вариант кода на языке среды Coq. Ключ компиляции использовался для того, чтобы предотвратить использование более точных (80 bit) вычислений с плавающей точкой на FPU x87. Столбцы таблицы *разность part1* и *разность part2* показывают модуль разности между полученными результатами по соответствующим выходным переменным. Столбец *nf* содержит выходные данные для соответствующей переменной, которые совпали в точности для всех наборов входных данных.

По представленным данным можно сделать вывод о том, что предлагаемая модель чисел с плавающей точкой в целом адекватно описывает аппаратную реализацию вычислений с плавающей точкой. Тем не менее, причины, которыми обусловлена ненулевая погрешность, требуют дальнейшего рассмотрения.

В таблице C.2 представлены результаты сравнения относительной погрешности моделей чисел с плавающей точкой и рациональных чисел (с вычислением приближённого арифметического квадратного корня) с результатом, полученным аналитически. Сравнивались результаты выполнения интерпретатора, определяющего динамическую семантику языка вычислений с плавающей точкой, с реализациями интерфейса вычислений для чисел с плавающей точкой двойной точности и для рациональных чисел, на одном и том же исходном коде. Исходный код был представлен в форме статической семантики языка арифметических выражений. Точность вычисления арифметического квадратного корня для рациональных чисел была ограничена 10^{-50}.

Полученные результаты позволяют утверждать, что модель может быть использована в задачах оценки погрешности, которая является следствием замены точных арифметических операций над действительными числами операциями над числами с плавающей точкой.

C.5. Адаптация модели для других реализаций лямбда-исчисления с зависимыми типами

Основные определения в составе модели вычислений с плавающей точкой были представлены как термы разновидности λ-исчисления с зависимыми типами, которая лежит в основе среды Coq. Далее рассмотрены основные синтаксические конструкции языка формальной спецификации и функционального программирования Gallina (является основным языком среды Coq), используемые в формальном определении модели, и соответствующие им термы λ-исчисления с зависимыми типами.

`Record` — синтаксическая конструкция, определяющая запись, фактически является определением типа зависимой суммы с набором дополнительных функций. В число таких функций входят следующие: конструктор, позволяющий получить элемент типа записи по значениям компонент; функция удаления, с использованием которой покомпонентно определяются функции (зависимые произведения) из типа записи; функции доступа к компонентам. Все такие функции могут быть определены в рамках λ-исчисления с поддержкой зависимых сумм и терма их удаления. Следует отметить, что, согласно официальной документации конструкция `Record` раскрывается в конструкцию определения индуктивных типов `Inductive` с одним конструктором. Несмотря на то, что с позиций формального представления такое определение отличается от используемого в Coq определения зависимой суммы `sigT`, определяемые для записи вспомогательные функции соответствуют термам введения и удаления элемента типа зависимой суммы.

`Fixpoint` — синтаксическая конструкция, позволяющая заменить удаление терма индуктивного типа оператором сопоставления с образцом `match`. В рамках модели конструкция используется со следующими типами: тип вектора `Vector.t` (`Bvector`); тип натуральных чисел `nat`; конечные типы из двух (`bool`) и трёъ (`comparison`) элементов. Таким образом, для определения модели на основе некоторой разновидности λ-исчисления с зависимыми типами, такая разновидность должна поддерживать термы удаления для перечисленных типов. При этом, как показано в [6] термы введения и удаления элемента типа вектора могут быть сведены к таким термам для натуральных чисел и конечных типов.

Таким образом, было установлено, что разработанная авторами модель существенным образом использует следующие типы в рамках λ-исчисления с зависимыми типами: зависимые суммы; конечные типы из двух и трёх элементов; натуральные числа. Кроме того, разновидность λ-исчисления, используемая для реализации, должна поддерживать термы удаления для таких типов, эквивалентные тем, которые используются в Coq. При этом следует отметить, что стандартные правила вывода для зависимых сумм, конечных типов и натуральных чисел в рамках теории типов (даны, например, в [7,8]) удовлетворяют таким требованиям.

С использованием результатов анализа, представленных в настоящем разделе, модель вычислений с плавающей точкой была адаптирована авторами к разновидности λ-исчисления с зависимыми типами, предложенной в [5].

C.6. Существующие результаты

Формальные модели вычислений с плавающей точкой достаточно давно являются предметом интереса исследователей. Такие модели были представлены в статье [9] и в отчёте [10]. При этом, в отличие от настоящей работы, исходный код таких моделей не был опубликован, а в качестве базовой формальной модели использовались модели, отличные от λ-исчисления с зависимыми типами. В числе современных работ следует отметить [11], в которой используется система Coq и модель λ-исчисления с зависимыми типами.

При этом в настоящей работе, в отличие от [11], операции деления и извлечения арифметического квадратного корня не требуют дополнительного указания точности вычислений. В настоящей работе используются реализации операций, соответствующие стандарту IEEE 754. В частности, для тестирования используется формат, соответствующий числам с плавающей точкой двойной точности. В числе других отличий от результатов указанной работы следует выделить компактность предлагаемой авторами модели. Код предлагаемой авторами модели в системе Coq занимает менее одной тыс. строк, тогда как модель [11] вместе с реализацией операций содержит большое количество свойств и их доказательств и занимает 22 тыс. строк кода. Выделить код реализации операций в этой модели для точного сравнения не представляется возможным в связи со сложной структурой зависимостей кода.

Результаты тестирования модели, представленные в настоящей статье, включая интерфейс для модели вычислений с плавающей точкой и результаты, полученные при его применении, ближе к тем, которые опубликованы в [12]. Результаты этой работы показывают также, что потенциальной областью приложения модели является верифицируемая компиляция программ. Отметим, что в библиографии статьи [11] приведён ряд дополнительных ссылок на формальные модели вычислений с плавающей точкой для разных формальных систем.

C.7. Заключение

В рамках работы, результаты которой представлены в настоящей статье, построена формальная модель чисел с плавающей точкой в соответствие с положениями стандарта IEEE 754-2008. Для модели реализованы арифметические операции сложения, вычитания, умножения, деления и извлечения арифметического квадратного корня, а также операции сравнения, которые требуют положения этого стандарта. Модель была верифицирована на примере фрагмента кода программного комплекса математического моделирования физических процессов в атомных электростанциях.

Для применения модели к задачам анализа вычислительной погрешности в рамках дедуктивной формальной верификации предложен абстрактный интерфейс вычислений

над приближениями действительных чисел и язык арифметических выражений для такого интерфейса. Интерфейс реализован как для разработанной модели, так и для реализации приближённых вычислений над рациональными числами. Для языка разработаны статическая и динамическая формальная семантика в форме интерпретатора. Применимость интерфейса и модели к задачам анализа вычислительной погрешности продемонстрирована на примере фрагмента кода, использовавшегося для верификации модели.

Кроме того, с целью использования модели при верификации с использованием реализаций λ-исчисления с зависимыми типами, отличных от Coq, для основных компонентов модели было получено представление модели для λ-исчисления с зависимыми типами с поддержкой зависимых сумм и натуральных чисел со стандартной схемой удаления.

Дальнейшие исследования в рассматриваемой области будут проводиться по трём перечисленным далее направлениям.

1. Реализация интерфейса вычислений над приближениями действительных чисел над моделью точных вычислений над действительными числами. Такие модели известны для реализаций лямбда-исчисления, поддерживающих коиндуктивные типы [13], в частности, для Coq [14] и Agda [15]. С использованием таких моделей может быть установлено соответствие между используемыми численными методами и их реализацией для чисел с плавающей точкой.
2. Адаптация языка арифметических выражений к общему подходу описания формальной семантики языков программирования с использованием монад и преобразователей монад. Решение этой задачи позволит использовать модель в качестве отдельного модуля при описании формальной семантики языков программирования.
3. Реализация процедур частичного разрешения и DSL на основе модели в рамках реализации методов дедуктивной верификации для языково-ориентированного программирования [5,16].

1. Васенин В.А. и др. Распараллеливание теплогидравлического расчетного кода CMS в составе полномасштабной суперкомпьютерной модели «Виртуальная АЭС» // Ломоносовские чтения. Тезисы научной конференции. Изд-во МГУ. 2013.

2. Goldberg D. What every computer scientist should know about floating point arithmetic // ACM Computing Surveys. 1991. Т. 23, № 1. С. 5–48.

3. Microprocessor Standards, Computer Society. IEEE std 754TM-2008 (revision of IEEE std 754-1985), IEEE standard for floating-point arithmetic. 2008. Т. 2008.

4. Coq Development Team. The coq proof assistant reference manual. INRIA-Rocquencourt, 2012.

5. Васенин В., Кривчиков М. Предметно-ориентированные языки с заданной формальной семантикой на основе лямбда-исчисления с зависимыми типами // Международная конференция «мальцевские чтения». Новосибирск: Изд-во НГУ, 2014. С. 25–25.

6. Chlipala A. Certified programming with dependent types: A pragmatic introduction to the coq proof assistant. The MIT Press, 2013.

7. Thompson S. Type theory and functional programming. Redwood City, CA: Addison Wesley Longman Publishing Co., Inc, 1991.

8. The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Princeton, NJ: Institute for Advanced Study, 2013.

9. Barrett G. Formal methods applied to a floating-point number system // IEEE Transactions on Software Engineering. 1989. T. 15, № 5. C. 611–621.

10. Carreno Victor A. Interpretation of IEEE-854 floating-point standard and definition in the HOL system. NASA Langley Technical Report Server, 1995.

11. Boldo S., Melquiond G. Flocq: A unified library for proving floating-point algorithms in coq. IEEE, 2011. C. 243–252.

12. Boldo S. и др. Verified compilation of floating-point computations // Journal of Automated Reasoning. 2015. T. 54, № 2. C. 135–163.

13. Berger U., Hou T. Coinduction for exact real number computation // Theory of Computing Systems. 2007. T. 43, № 3-4. C. 394–409.

14. Krebbers R., Spitters B. Type classes for efficient exact real arithmetic in coq // Logical Methods in Computer Science / под ред. Escardó M. 2013. T. 9, № 1.

15. Chuang C.M. Extraction of programs for exact real number computation using agda: PhD thesis. Swansea: Dept. of Computer Science, Swansea University, 2011.

16. Васенин В., Кривчиков М. Языково-ориентированное программирование для формальной верификации программного обеспечения // Четвёртая научно-практическая конференция. МИЭМ НИУ ВШЭ, Москва: Изд-во НИУ ВШЭ, Москва, 2015. С. 30–31.

Приложение D

Статическая формальная семантика стандарта ECMA-335

В настоящем приложении представлен фрагмент статической формальной семантики стандарта ECMA-335, описанный в главе 4, в терминах базового языка.

(λ {Ide Ide_{dot} ℕ F String Culture Integer **Type**} (**tuple**

[$\text{Module}_{\text{ref}}$ (Σ {name Ide_{dot}} {hash ℕ})]

[Version (Σ {major minor build revision ℕ})]

[$\text{Assembly}_{\text{ref}}$ (Σ {name Ide_{dot}} {version Version} {culture Culture}
{mdhash ℕ} {public-key (List #256)})]

[scr (Ind (Σ **Type Type**)
(λ (Σ **Type Type**) (**split** (Σ **Type Type**) (**var** 0)
(λ {Scope $\text{Class}_{\text{ref}}$ **Type**} (**pair** _ (adt
{Top (Σ {assembly $\text{Assembly}_{\text{ref}}$} {module $\text{Module}_{\text{ref}}$})}
{Nested $\text{Class}_{\text{ref}}$})
(Σ {scope Scope} {name Ide_{dot}} {nparams ℕ}))))))]

[Scope (**fst** scr)]

[$\text{Class}_{\text{ref}}$ (**snd** scr)]

[$\text{Method}_{\text{call-conv}}$ (adt {Static #1} {Instance #1} {Explicit #1})]

[$\text{Method}_{\text{call-kind}}$ (adt {Default #1} {Vararg #1} {Unmanaged #1})]

[$\text{FMethod}_{\text{sig}}$ (λ { type **Type** } (Σ
{ call-conv $\text{Method}_{\text{call-conv}}$ } { call-kind $\text{Method}_{\text{call-kind}}$ }
{ **return** type} {args (List type)} {nparams ℕ}))]

[Ftype (λ {$\text{Method}_{\text{sig}}$ type **Type**} (adt
{ Gen_{t} ℕ } { Gen_{m} ℕ } { Ref $\text{Class}_{\text{ref}}$ } { Ptr_{m} type } { Ptr_{u} type }
{ Val $\text{Class}_{\text{ref}}$ } { vector type } { Boxed $\text{Class}_{\text{ref}}$ }
{ Array (Σ { t type } { dim ℕ } { range (Vector ℕ (Σ {min max ℕ})) }) }
{ Modifier (Σ { required #2 } { ty mod type })}
{ Instance (Σ $\text{Class}_{\text{ref}}$ (List type)) }))]

[mst (Ind (Σ **Type Type**)
(λ (Σ **Type Type**) (**split** (Σ **Type Type**) (**var** 0)
(λ {$\text{Method}_{\text{sig}}$ type **Type**} (**pair** _
($\text{FMethod}_{\text{sig}}$ type) (Ftype $\text{Method}_{\text{sig}}$ type))))))]

[$\text{Method}_{\text{sig}}$ (**fst** mst)]

[type (**snd** mst)]
[Access (adt { Public Private Internal Protected InternalAndProtected
InternalOrProtected CompilerControlled #1 })]
[Special (Σ {Specialname RTSpecialname #2})]

[Class$_{layout}$ (adt {Auto Sequential Explicit #1})]
[Class$_{attr}$ (Σ { Visibility Access } { Layout Class$_{layout}$ } { Special$_{attrs}$ Special }
{ Abstract Beforefieldinit Interface Sealed Serializable #2 })]
[Class$_{def}$ (Σ { Ref Class$_{ref}$ } { Attr Class$_{attr}$ }
{ Parent type }{ Interfaces (List type) })]

[Field$_{ref}$ (Σ { Field-scope Scope } { Field-type **Type** } { Name Ide })]
[Field$_{attr}$ (Σ { Visibility Access } { Special$_{attrs}$ Special }
{ Literal Initonly Serializable Static #2 })]
[Field$_{init}$ (adt { Bool #2 } { Bytes (List #256) } { Char (Σ #256 #256) }
{ Float$_{32}$ Float$_{64}$ F } { Int$_{8}$ UInt$_{8}$ #256 }
{ Int$_{16}$ UInt$_{16}$ (Σ #256 #256) } { Int$_{32}$ UInt$_{32}$ (Σ #256 #256 #256 #256) }
{ Int$_{64}$ UInt$_{64}$ (Σ #256 #256 #256 #256 #256 #256 #256 #256) }
{ String$_{init}$ String } { Nullerf None #1 })]
[Field$_{def}$ (Σ {Ref Field$_{ref}$ } { Attr Field$_{attr}$ } { Init Field$_{init}$ })]

[Method$_{ref}$ (Σ { Method$_{scope}$ Scope } { Name Ide } { Signature Method$_{signature}$ })]
[Param (Σ { Param$_{type}$ type } { In Out Opt #2 } { Name Ide })]
[Method$_{code}$ (adt { Cil Native Runtime #1 })]
[Method$_{attr}$ (Σ { Visibility Access } { Code Method$_{code}$ } { Special$_{attrs}$ Special }
{ Abstract Final Hidebysig Newslot Stati Virtual
Strict Internalcall Noinlining Synchronized #2 })]
[Method$_{def}$ (Σ { Ref Method$_{ref}$ } { Attr Method$_{attr}$ })]

[Generic$_{variance}$ (adt { Invariant Covariant Contravariant #1 })]
[Generic$_{constraint}$ (adt { Unconstr Value Reference Ctor ReferenceCtor #1 })]
[Generic$_{def}$ (Σ { Name Ide } { Number N } { types (List type) }
{ Variance Generic$_{variance}$ } { Constraint Generic$_{constraint}$ })]

[Property$_{ref}$ (Σ { Scope Scope } { Name Ide } { type Method$_{signature}$ })]
[Property$_{attr}$ (Σ { Special Special } { Getter Setter Method$_{ref}$ } { Other (List Method$_{ref}$) })]
[Property$_{def}$ (Σ { Ref Property$_{ref}$ } { Attr Property$_{attr}$ } { Init Field$_{init}$ })]

[Event$_{ref}$ (Σ { Scope Scope } { Name Ide } { type **Type** })]

[Event_{attr} (Σ { Special Special } { Addon Removeon Fire Method_{ref} }
{ Other (List Method_{ref}) })]
[Event_{def} (Σ { Ref Event_{ref} } { Attr Event_{attr} })]

[Custom_{fields} (List (Σ { Ref Field_{ref} } { Init Field_{init} }))]
[$\text{Custom}_{properties}$ (List (Σ { Ref Property_{ref} } { Init Field_{init} }))]
[Custom_{def} (Σ { Ctor Method_{ref} } { Args (List Field_{init}) }
{ Fields Custom_{fields} } { Properties $\text{Custom}_{properties}$ })]

[Security_{action} (adt { Assert Demand Inheritcheck Linkcheck Permitonly
Reqopt Reqrefuse Request #1 })]
[Security_{def} (adt { Action Security_{action} } { type type } { Properties $\text{Custom}_{properties}$ })]

[Member_{ref} (adt { Field Field_{ref} } { Method Method_{ref} })]
[Generic_{item} (adt { Class Class_{ref} } { Method Method_{ref} })]
[Security_{item} (adt { Assembly Assembly_{ref} } { Method Method_{ref} } { Class Class_{ref} })]
[Custom_{item} (adt { Assembly Assembly_{ref} } { Class Class_{ref} }
{ Field Field_{ref} } { Method Method_{ref} }
{ Property Property_{ref} } { Event Event_{ref} })]

[Class_{env} (Map Class_{ref} Class_{def})]
[Field_{env} (Map Field_{ref} Field_{def})]
[Property_{env} (Map Property_{ref} Property_{def})]
[Security_{env} (Map Security_{item} Security_{def})]
[Custom_{env} (Map Custom_{item} Custom_{def})]
[Generic_{env} (Map Generic_{item} Generic_{def})]
[Compare (adt { Equals GreaterThan LessThan #1 })]
[Token (adt { Field Field_{ref} } { Method (Σ Method_{ref} (List type)) }
{ **Type**-token type })]
[Check (adt { Typecheck Rangecheck Nullcheck #1 })]
[Unsigned (adt { Signed Unsigned #1 })]
[Overflow (adt { Normal Overflow $\text{Overflow}_{unsigned}$ #1 })]
[Prim (adt {i_1 u_1 i_2 u_2 i_4 u_4 i_8 u_8 i u r_4 r_8 })]

[Instr_{prefix} (adt { Readonly Tail Volatile #1 } { No Check }
{ Unaligned (adt { Byte Word Dword #1 }) })]
[Instr (adt
{ And Arglist Break Ckfinite $\text{Conv}_{r\text{-}unsigned}$ Cpblk Dup Endfilter Endfault
Initblk Ldnull Ldind_{ref} Ldlen Localloc Neg Nop Not Or Pop Refanytype

Ret Rethrow Shl Stind$_{ref}$ Throw Xor #1}
{ Div Rem Shr Unsigned }
{ Ldind Stind Ldelem Prim }
{ Add Mul Sub Overflow }
{ Conv (Σ Prim Overflow) }
{ Comp (Σ Compare Unsigned) }
{ BranchCond (Σ Compare Unsigned $\mathbb{N}$) }
{ Branch Brfalse Brtrue Leave
Ldarg Ldarga Ldloc Ldloca Starg Stloc $\mathbb{N}$ }
{ Call Callvirt Ldvirtftn (Σ Method$_{ref}$ List type) }
{ CallI Method$_{sig}$}
{ Jmp Ldftn Newobj Method$_{ref}$ }
{ Ldc$_{i_4}$ Ldc$_{i_8}$ Integer }
{ Ldc$_{r_4}$ Ldc$_{r_8}$ F }
{ Switch (Σ { nTargets $\mathbb{N}$ } { targets (Vector $\mathbb{N}$ nTargets) }) }
{ Box Castclass Cpobj Initobj Isinst Ldelem Ldelema Ldobj Mkrefany
Newarr Sizeof Stelem Stobj Unbox UnboxAny type }
{ Ldfld Ldflda Ldsfld Ldsflda Stfld Stsfld Field$_{ref}$ }
{ Ldstr String }
{ Ldtoken Token })]

[Instr$_{item}$ (Σ { Instruction Instr } { Prefixes (List Prefix) })]

[Seh$_{handler}$ (adt { Finaly Fault #1 } { Catch Class$_{ref}$ } { Filter $\mathbb{N}$ })]
[Seh (Σ { Start End HStart HEnd $\mathbb{N}$ } { Handler Seh$_{handler}$ })]
[Local (Σ { Number $\mathbb{N}$ } { Local-type type } { Name Ide })]
[Method$_{body\text{-}attr}$ (Σ { Maxstack $\mathbb{N}$ } { Seh (List Seh) })]
[Method$_{item}$ (Σ { Method Method$_{def}$ } { Params (List Param) } { Locals (List Local) }
{ Attr Method$_{body\text{-}attr}$ } { Instructions (List Instr$_{item}$) })]
[Method$_{env}$ (Map Method$_{ref}$ Method$_{item}$)]
))

Приложение Е

Фрагмент кода CMS для демонстрации

В настоящем приложении представлены листинги минимального фрагмента кода CMS. Исследуемый фрагмент входит в состав расчётного кода CMS (Compressible Media Solver), который предназначен для вычисления в режиме реального времени теплогидравлических параметров в двухфазной сжимаемой неравновесной парогазокапельной среде с любым числом неконденсируемых газов. Код разработан специалистами ОАО «ВНИИАЭС» и в настоящее время широко применяется в программном обеспечении тренажёров по обучению управлением и отработке действий при чрезвычайных ситуациях персонала атомных электростанций [77].

Код на языке С:

```
#include <math.h>

double kf0=0.0,kf1=1.0,kfp5=0.5,visc=3.16;

void ysparg(double p_out,double p_in_out,
  double p_in_in,double d1,double d2,
  double dtt, double *part1,double *part2,
  double *nf)
{
 /*********************************************/
 //   p_out    - pressure out          Pa  i
 //   p_in_out - pressure in (line in)   Pa  i
 //   p_in_in  - pressure in (line out)  Pa  i
 //   part_in  - part flow in             o
 //   part_out - part flow out            o
 //   block_f  - (1 flow, 0 not flow)      o
 //   dobj_1   - density in               i
 //   dobj_2   - density out              i
 //   dtt      - time step                i
 /*********************************************/
 double f_8,f_9,f_tin,f_tout;

 *nf = kf1;
 f_8=visc*sqrt(d1);
 f_9=visc*sqrt(d2);
 if((p_in_out > p_out) && (p_in_in >= p_out)) {
```

```
  f_tin = (p_in_out-p_out) /
      (p_in_out-p_out+f_8);
  f_tout = kf0;
 }
 else {
  if((p_in_out < p_out) && (p_in_in <= p_out)) {
   f_tin = kf0;
   f_tout = (p_out-p_in_in) /
      (p_out-p_in_in+f_9);
  }
  else {
   if((p_in_out - p_in_in) < kf0) {
    *nf = kf0;
    f_tin = kfp5;
    f_tout = kfp5;
   }
   else {
    f_tin = (p_in_out-p_out) /
       (p_in_out-p_in_in+f_8);
    f_tout = (p_out-p_in_in) /
       (p_in_out-p_in_in+f_9);
   }
  }
 }
 *part1+=dtt*(f_tin-*part1);
 *part2+=dtt*(f_tout-*part2);
}
```

Код на языке CIL, полученный с использованием компилятора lcc (записан в три столбца с переносом в рамках одной страницы):

```
.field public static float64 kf0
.data $kf0 = { float64 (0.0) }

.field public static float64 kf1
.data $kf1 = {
  float64 (1.000000e+00)
}

.field public static float64 kfp5
.data $kfp5 {
  float64 (0.500000e+00)
}

.field public static float64 visc
.data $visc = {
  float64 (3.160000e+00)
}

.method public hidebysig static
void 'ysparg'(float64 p_out,
  float64 p_in_out, float64 p_in_in,
  float64 d1,float64 d2, float64 dtt,
  float64& part1,float64& part2,
  [out] float64& nf)
  cil managed {
      .locals ([0] float64 'f_8')
      .locals ([1] float64 'f_9')
      .locals ([2] float64 'f_tin')
      .locals ([3] float64 'f_tout')
      .maxstack 5

    ldarg.s 8
    ldsfld float64 kf1
    stind.r8
    ldsfld float64 visc
    ldarg.3
    call float64 class [mscorlib]\
      System.Math::Sqrt(float64)
    mul
    stloc.0
    ldsfld float64 visc
```

```
  ldarg.s 4
  call float64 class [mscorlib]\
    System.Math::Sqrt(float64)
  mul
  stloc.1
  ldarg.1
  ldarg.0
  ble.un $L15

  ldarg.2
  ldarg.0
  blt.un $L15

  ldarg.1
  ldarg.0
  sub
  dup
  ldloc.0
  add
  div
  stloc.2
  ldsfld float64 kf0
  stloc.3
  br $L20

$L15:
  ldarg.1
  ldarg.0
  bge.un $L17

  ldarg.2
  ldarg.0
  bgt.un $L17

  ldsfld float64 kf0
  stloc.2
  ldarg.0
  ldarg.2
  sub
  dup
  ldloc.1
```

```
  add
  div
  stloc.3
  br $L20

$L17:
  ldarg.1
  ldarg.2
  sub
  ldsfld float64 kf0
  bge $L19

  ldarg.s 8
  ldsfld float64 kf0
  stind.r8
  ldsfld float64 kfp5
  dup
  stloc.2
  stloc.3
  br $L20

$L19:
       ldarg.1
       ldarg.0
       sub
       ldarg.1
       ldarg.2
       sub
       ldloc.0
       add
       div
       stloc.2
       ldarg.0
       ldarg.2
       sub
       ldarg.1
       ldarg.2
       sub
       ldloc.1
       add
       div
```

```
        stloc.3

$L20:
        ldarg.s 6
        ldarg.s 6
        ldind.r8
        ldarg.s 5
        ldloc.2
        ldarg.s 6
        ldind.r8
        sub
        mul
        add
        stind.r8
        ldarg.s 7
        ldarg.s 7
        ldind.r8
        ldarg.s 5
        ldloc.3
        ldarg.s 7
        ldind.r8
        sub
        mul
        add
        stind.r8
        ret
}
```

Приложение F

Реализация модели динамического параллельного исполнения программ

В настоящем приложении представлено два варианта реализации модели динамического параллельного исполнения программ. Первый вариант основан на описании, соответствующем главе 5 настоящей работы и содержит некоторые фрагменты, опущенные в этой главе. Для его записи используется синтаксис, аналогичный применяемому в главах 2, 3 и 5. Второй вариант на языке программирования Haskell реализует разновидность модели на базе теории доменов.

F.1. Фрагменты кода модели

$$
\begin{array}{lll}
\mathrm{Bf}_{\mathrm{semantics}} & \equiv & \lambda\ (\mathrm{b} : \mathrm{Bf})\ .\ (\mathrm{sem}_{\mathrm{true}}\ \mathrm{sem}_{\mathrm{false}} : \mathrm{CFL}_{\mathrm{dynamic}} \cdot \mathrm{S})\ .\ (\mathrm{s} : \mathrm{S})\ . \\
 & & \mathbf{bind}\ (\mathrm{asks} \cdot (\lambda\ \mathrm{Prog}\ .\ (\mathrm{B} \cdot \mathbf{var}\ 0 \cdot \mathrm{b})))\ (\lambda\ (\mathrm{b} : \Pi \mathrm{S}.\#2). \\
 & & \mathbf{case}_2\ (\mathrm{CFL}_{\mathrm{dynamic}} \cdot \mathrm{S}, \mathrm{b} \cdot \mathrm{s}, \mathrm{sem}_{\mathrm{true}}\ , \mathrm{sem}_{\mathrm{false}}\ ))) \\
\mathrm{CFL}_{\mathrm{semantics}} & \equiv & \lambda\ (\mathrm{st} : \mathrm{St})\ .\ \mathrm{ind}_{\mathrm{St}} \cdot (\mathrm{CFL}_{\mathrm{dynamic}} \cdot \mathrm{S}) \cdot \\
 & & [\mathrm{skip} \qquad \lambda\ \#1.\ \mathrm{get}_{\mathrm{S}}\ ] \\
 & & [\mathrm{comp} \qquad \lambda\ (\mathrm{c} : \mathrm{Cf})\ .\ \mathbf{bind}\ (\mathrm{asks} \cdot (\lambda\ \mathrm{Prog}\ .\ (\mathrm{C} \cdot \mathbf{var}\ 0 \cdot \mathrm{c})))\ \mathrm{modify}_{\mathrm{S}}] \\
 & & [\text{if-then-else} \qquad \lambda\ (\Sigma(\mathrm{b} : \mathrm{Bf})\ .\ (\mathrm{sem}_{\mathrm{t}}\ \mathrm{sem}_{\mathrm{f}} : \mathrm{CFL}_{\mathrm{dynamic}} \cdot \mathrm{S})\ . \\
 & & \qquad \mathbf{bind}\ \mathrm{get}_{\mathrm{S}}\ (\mathrm{Bf}_{\mathrm{semantics}} \cdot \mathrm{b} \cdot \mathrm{sem}_{\mathrm{t}} \cdot \mathrm{sem}_{\mathrm{f}}))] \\
 & & [\text{while-do} \qquad \lambda\ (\Sigma(\mathrm{b} : \mathrm{Bf})\ (\mathrm{sem}_{\mathrm{l}} : \mathrm{CFL}_{\mathrm{dynamic}} \cdot \mathrm{S}))\ . \\
 & & \qquad \text{ana-iter}_{\mathrm{ReactT}} \cdot \mathrm{S} \cdot \\
 & & \qquad\ \ \lambda\ (\mathrm{s} : \mathrm{S})\ .\ (\mathrm{n} : \mathbb{N})\ .\ \mathbf{iter}_{\mathbb{N}}(\mathrm{n}, \\
 & & \qquad\qquad \mathrm{FReactT}_{\mathbf{iter}} \cdot \mathrm{S}, \\
 & & \qquad\qquad \lambda\ \mathrm{S}\ .\ \mathbf{return}\ (\mathbf{var}\ 0) \\
 & & \qquad\qquad (\mathrm{k}, \mathrm{prev}_{\mathrm{red}}) \mapsto \lambda\ \mathrm{S}\ .\ \mathrm{Bf}_{\mathrm{semantics}} \cdot \mathrm{b} \cdot \\
 & & \qquad\qquad\ (\mathrm{Cont} \cdot (\mathbf{bind}\ (\mathrm{set}_{\mathrm{S}} \cdot (\mathbf{var}\ 0))\ \mathrm{sem}_{\mathrm{l}}) \cdot (\lambda\ \#1\ .\ \mathrm{prev}_{\mathrm{red}})) \cdot \\
 & & \qquad\qquad\ (\mathbf{bind}\ \mathrm{get}_{\mathrm{S}}\ \mathbf{return}) \cdot (\mathbf{var}\ 0)) \\
 & & \qquad\ \ ] \\
 & & [\mathrm{seq} \qquad \lambda(\Sigma(\mathrm{s}_1\ \mathrm{s}_2 : \mathrm{CFL}_{\mathrm{dynamic}} \cdot \mathrm{S}))\ .\ \mathbf{bind}\ \mathrm{s}_1\ (\lambda\ \mathrm{x}.\ \mathrm{s}_2)] \\
 & & [\mathrm{spawn} \qquad \lambda(\Sigma(\mathrm{t} : \mathrm{Tf}).(\mathrm{c} : \mathrm{Cf}).(\mathrm{v} : \mathrm{Var}))\ .\ \mathbf{bind}\ (\\
 & & \qquad \mathbf{bind}\ \mathrm{get}_{\mathrm{S}}\ (\lambda\ (\mathrm{s} : \mathrm{S}). \\
 & & \qquad \mathbf{bind}\ (\mathrm{asks} \cdot (\lambda\ \mathrm{Prog}\ .\ (\mathrm{C} \cdot \mathbf{var}\ 0 \cdot \mathrm{c})))\ (\lambda(\mathrm{c}_{\mathrm{sem}} : \Pi \mathrm{S}.\mathrm{S}). \\
 & & \qquad \mathrm{SpawnReq} \cdot \mathbf{pair}(\mathrm{t}, \mathrm{c}_{\mathrm{sem}} \cdot \mathrm{s}) \cdot (\lambda\ (\mathrm{i} : \mathrm{TId})\ .\ \mathrm{modify}_{\mathrm{V}} \cdot (\text{set-key} \cdot \mathrm{v} \cdot \mathrm{i})))))
\end{array}
$$

```
                get_S]
                [wait        λ(Σ(v : Var).(m : Mf)) . bind (
                             bind get_S (λ (s_0 : S).
                             bind (asks · (λ Prog . (M · var 0 · m))) (λ (m_sem : ΠS.S.S)
                             bind (gets_V · (get-key · v)) (λ TId .
                             WaitReq · (var 0) · (λ S . set_S (m_sem · s_0 · (var 0)))))))
                get_S]

apply         :  Π (c : CFL · CFL_dynamic · S) . CFL_system · (CFL_dynamic · S)
loadContext   :  Π (thread : Thread) . CFL_system · #1
saveContext   :  Π (thread : Thread) . CFL_system · Thread
handle        :  Π (thread : Thread) . (system : System) . CFL_system · System
handle        ≡  λ (thread system) . match fst(semantics · thread)
                 [Done (λ S . return system)]
                 [Request (λ (Σ(req : Req).(next : Π(Rsp · fst(req)).CFL · CFL_dynamic · S)).
                  match req
                  [Cont (λ #1 . bind (loadContext · thread) (λ _ .
                   bind (apply · (next · (Ack · 0#1))) (λ _ .
                   bind (saveContext · thread(state · thread))
                     (λ thread′ . return · setKey · system ·
                     (tid · thread) · thread′))))]
                  [SpawnReq (λ Σ(name : Tf).(state : S) .
                   let ns ≡ nextId · system
                     newTid ≡ fst ns
                     system′ ≡ snd ns
                   in bind (asks · (λ Prog . (T · (var 0) · name))) (λ (tSem : CFL_dynamic · S).
                     bind get_V (λ (v : _) .
                     bind (setThread · newTid ·
                      tuple(newTid, tid · thread, name, v, state, 𝕆, v, state, tSem))
                     return next · (SpawnRsp · newTid))))]
                  [WaitReq (λ (id : TId).
                   bind (getThread · id) (λ (thread : _).
                   match fst(semantics · thread)
                     [Done next]
                     [Request (λ _. return system) ])))]])
can-handle    :  Π (thread : Thread) . CFL_system · #2
can-handle    ≡  λ (thread : _) . match fst(semantics · thread)
                 [Done (λ _. return 0#2)]
                 [Request (λ (Σ(req : Req).(next : )) . match req
```

$$
\begin{aligned}
&\left[\text{Cont } \left(\lambda\ _.\ \textbf{return } 1\#2\right)\right]\\
&\left[\text{SpawnReq } \left(\lambda\ _.\ \textbf{return } 1\#2\right)\ \right]\\
&\left[\text{WaitReq } \left(\lambda\ \left(\text{id} : \text{TId}\right).\right.\right.\\
&\quad \textbf{bind } \left(\text{getThread} \cdot \text{id}\right)\ \left(\lambda\ \left(\text{thread} : _\right)\ .\right.\\
&\qquad \textbf{match fst}\left(\text{semantics} \cdot \text{thread}\right)\\
&\qquad\ \left[\text{Done } \left(\lambda\ _.\ \textbf{return } 1\#2\right)\right]\\
&\qquad\ \left.\left.\left.\left[\text{Request } \left(\lambda\ _.\ \textbf{return } 0\#2\right)\right]\right)\right)\right]\Big]
\end{aligned}
$$

F.2. Исходный код на языке Haskell

В следующем далее листинге представлен пример реализации модели динамического параллельного исполнения программ на языке Haskell.

```
module TSemantics where


import Control.Monad
import Control.Monad.State
import Control.Monad.Reader
import Control.Monad.Trans
import Control.Monad.Identity
import List (partition, sortBy, delete, deleteBy)
import Data.Map (Map, (!), mapKeys, keys, insert, toDescList, toAscList, empty, fromList)

-- * Reactive concurrency monad by Harrison W., "Cheap (But Functional) Threads"
data (Monad m) => ReactT req rsp m a = D a | P (req, rsp -> (m (ReactT req rsp m a)))
instance (Monad m) => Monad (ReactT req rsp m) where
 return v = D v
 (D v) >>= f = f v
 P (q,r) >>= f = P (q, \c -> (r c) >>= (\k -> return (k >>= f)))

data (Monad m) => ResT m a = Done a | Pause (m (ResT m a))
instance (Monad m) => Monad (ResT m) where
 return = Done
 (Done v) >>= f = f v
 (Pause r) >>= f = Pause (r >>= \k -> return (k >>= f))

rstep :: (Monad m) => m a -> ResT m a
rstep x = Pause (x >>= (return . Done))
```

```
-- * Basic Definitions
-- Functions containing T-statements
type Tf = Int
-- Computational functions - updating state
type Cf = Int
-- Mf :: returned state -> current state -> merged state
type Mf = Int
-- State to boolean functions
type Bf = Int

type Id = Int
type Var = Int

-- a is local state type parameter

-- Variable mapping (to pass variables to functions)
type Vm = Map Var Var

-- * Statements
data St =
   Skip
 | Comp  Cf           -- Compute (update state by calling Cf)
 | Cond  Bf St St      -- Conditional operator (if Bf then St else St)
 | While Bf St         -- Loop (while Bf do St)
 | Seq   St St         -- Sequence (St ; St)
 | Call  Tf Cf Vm Mf    -- Call Tf with
                     --  arguments produced by Cf
                     --  variables passed by Vm
                     --  result modified by Mf
 | Spawn Var Tf Cf Vm   -- Spawn Tf (and put process id in Var) with
                     --  arguments produced by Cf
                     --  variables passed by Vm
 | Wait Var Mf         -- Wait computation of Var and merge state with Mf
 | Return              -- Finish computations

-- * Environments
type Ce a = Map Cf (a -> a)
type Me a = Map Mf (a -> a -> a)
type Be a = Map Bf (a -> Bool)
```

```
type Te a = Map Tf St

-- Complete external environment (= program definition)
data Env a = Env {
 ce :: Ce a,
 me :: Me a,
 be :: Be a,
 te :: Te a
}

type StateVar = Map Var Id



data Req a = Cont | SpawnReq Tf a StateVar | WaitReq Id
data Rsp a = Ack  | SpawnRsp Id           | WaitRsp a

type Inner a = ReaderT (Env a) (
   StateT (StateVar) (
    StateT a Identity))

type Re a =
 ReactT (Req a) (Rsp a) (Inner a)

type R a =
 ResT (Inner a)

step :: (Monad m) => m b -> ReactT (Req a) (Rsp a) m b
step x = P (Cont, \Ack -> x »= (return . D))

signal :: Req a -> Re a (Rsp a)
signal q = P (q, return . return)

getA :: Re a a
getA = step $ lift $ lift $ get

getSV :: Re a (Map Var Id)
getSV = step $ lift $ get

getV :: Var -> Re a Id
```

```
getV v = step $ lift $ gets $ flip (!) v

askCf :: Cf -> Re a (a->a)
askCf c = step $ asks $ flip (!) c . ce

askMf :: Mf -> Re a (a->a->a)
askMf m = step $ asks $ flip (!) m . me

askBf :: Bf -> Re a (a -> Bool)
askBf b = step $ asks $ flip (!) b . be

askTf :: Tf -> R a St
askTf t = rstep $ asks $ flip (!) t . te

update :: (a -> a) -> Re a a
update c = do
 step $ lift $ lift $ modify c
 getA

putV :: Var -> Id -> Re a a
putV v i = do
 step $ lift $ modify $ insert v i
 getA

-- * Language semantics
sem :: St -> Re a a
sem (Skip)             = getA
sem (Comp c)           = askCf c »= update
sem (Cond b s1 s2)      = do
                         s <- getA
                         bf <- askBf b
                         sem $ if bf s then s1 else s2
sem (While b s)        = askBf b »= flip mwhile (sem s)
                        where
                        mwhile c s1 = do
                         s <- getA
                         if (c s) then s1 » (mwhile c s1) else getA
sem (Seq s1 s2)        = sem s1 » sem s2
sem (Call t c v m)      = do
```

```
                       tst  <- step $ asks $ flip (!) t . te
                       cf   <- askCf c
                       mf   <- askMf m
                       sv   <- getSV
                       sl   <- getA
                       let sv1 = mapKeys ((!) v)
                       step $ lift $ modify (sv1)
                       step $ lift $ lift $ modify (cf)
                       sem tst
                       sl2 <- getA
                       step $ lift $ put sv
                       step $ lift $ lift $ put (mf sl2 sl)
                       getA

sem (Spawn v t c vm)   = do
                       sl <- getA
                       sv <- getSV
                       cf <- askCf c
                       (SpawnRsp id) <- signal $ SpawnReq t (cf sl)
                           (if (vm == empty) then empty else mapKeys ((!) vm) sv)
                       putV v id
sem (Wait v m)        = do
                       (WaitRsp s) <- getV v »= signal . WaitReq
                       mf <- askMf m
                       update (mf s)
sem (Return )         = do getA »= D


-- * TSystem semantics
-- System: ready processes, waiting processes, done processes, entry point, max id

saveState :: R a (StateVar, a)
saveState = do
 v <- rstep $ lift $ get
 l <- rstep $ lift $ lift $ get
 return (v, l)

loadState :: (StateVar, a) -> R a ()
loadState (v, l) = do
```

```
 rstep $ lift $ put v
 rstep $ lift $ lift $ put l

type Process a = (Id, Re a a, StateVar, a)
type System a = ([Process a], [(Id, Process a)], [(Id, a)], Id, Id)

-- Scheduler prototype
type Sched a = System a -> R a a

-- Now, as in source publication, sched should remove process from ready list
handler :: System a -> Process a -> Sched a -> R a a

-- Done -> move (id, result) to done list and move any dependent processes to ready list
handler (ready, waiting, done, entry, maxId) (id, D d, _, _) sched =
 let done_m = (id, d) : done in
  let (waiting_r, waiting_m) = partition (\(wid,proc) -> wid == id) waiting in
   let ready_m = map snd waiting_r ++ ready in
    sched (ready_m, waiting_m, done_m, entry, maxId)

-- Continue -> step
handler (ready, waiting, done, entry, maxId) (id, P (Cont, r), sv, sl) sched = do
 --(svo, slo) <- saveState
 loadState (sv, sl)
 Pause ((r Ack) »= \k -> do
   svm <- lift $ get
   slm <- lift $ lift $ get
   --lift $ put svo
   --lift $ lift $ put slo
   return $ next id k svm slm)
 where
  next id k svm slm = sched (ready ++ [(id, k, svm, slm)], waiting, done, entry, maxId)

-- Spawn -> add spawned process to ready, return its id, increment maxId
handler (ready, waiting, done, entry, maxId) (id, P (SpawnReq tf sls svs, r), sv, sl) sched = do
 tst <- askTf tf
 --(svo, slo) <- saveState
 loadState (sv, sl)
 Pause ((r (SpawnRsp maxId)) »= \k -> do
  svm <- lift $ get
```

```
  slm <- lift $ lift $ get
  --lift $ put svo
  --lift $ lift $ put slo
  return $ sched (ready ++ [(id, k, svm, slm), (maxId, sem tst, svs, sls)],
   waiting, done, entry, maxId + 1))

handler (ready, waiting, done, entry, maxId) (id, P (WaitReq wid, r), sv, sl) sched =
 case filter (\(rid, d) -> rid == wid) done of
  (rid, d) : tail -> do
    --(svo, slo) <- saveState
    loadState (sv, sl)
    Pause ((r (WaitRsp d)) »= \k -> do
     svm <- lift $ get
     slm <- lift $ lift $ get
     --lift $ put svo
     --lift $ lift $ put slo
     return $ sched (ready ++ [(id, k, svm, slm)],
      waiting, done, entry, maxId))
  [] ->
    sched (ready, (wid, (id, P (WaitReq wid, r), sv, sl)) : waiting , done, entry, maxId)

-- Default function for scheduling empty queue
-- If there's nothing to run, so either we've finished or we're in deadlock
-- (last case is impossible in this model)
sched_empty :: Id -> [(Id, a)] -> R a a
sched_empty entry done =
 case filter (\(rid, d) -> rid == entry) done of
  (rid, d) : tail -> Done d

-- Sequential scheduler: run process with max id
sched_seq :: Sched a
sched_seq ([], waiting, done, entry, maxId) = sched_empty entry done

-- Sort ready processes by id descending and then select top
sched_seq (ready, waiting, done, entry, maxId) =
 let process : ready_m = sortBy desc_id ready
  in handler (ready_m, waiting, done, entry, maxId) process sched_seq
  where
   desc_id (id1, _, _, _) (id2, _, _, _) = compare id2 id1
```

```
-- Lazy scheduler : run process
sched_lazy :: Sched a
sched_lazy ([], _, done, entry, _) = sched_empty entry done

-- We're making use of property of handler _ (D _) _:
--  when process P1 is done, all processes waiting P1 are
--  automatically moved from waiting list to ready list.
sched_lazy (ready, waiting, done, entry, maxId) =
-- if entry point finished, stop computation
-- otherwise, explore dependency "graph"
-- (in our model it should be called dependency string :)
 case filter (\(id, d) -> id == entry) done of
  (id, d) : tail -> Done d
  [] -> let process = find_lazy entry ready waiting in
   handler (deleteBy procEq process ready, waiting, done, entry, maxId) process sched_lazy
 where
  find_lazy entry ready waiting =
   case filter (\(id, _, _, _) -> id == entry) ready of
    process : _ -> process
    [] -> case filter (\(_, (id, _,_,_)) -> id == entry) waiting of
     (waiting_for, _) : _ -> find_lazy waiting_for ready waiting
  procEq (idx, _, _, _) (idy, _, _, _) = idx == idy

-- * Program semantics

type Program a = (Env a, Tf)

program_sem :: Program a -> a -> Sched a -> Int -> (a, StateVar, R a a)
approx_nth  :: Env a -> a -> StateVar ->  R a a -> Int -> (a, StateVar, R a a)
approx_step :: Env a -> a -> StateVar -> R a a -> (a, StateVar, R a a)

program_sem prog args sched n =
 let (env, entry) = prog in
 let ef = (te env) ! entry in
 let system_sem = sched ([(1, sem ef, empty, args)], [], [], 1, 2) in
 let (sl, sv, act) = approx_nth env args empty system_sem n in
 (sl, sv, act)
```

```
approx_nth _ args sv act 0 = (args, sv, act)
approx_nth env args sv act n =
 let (args1, sv1, act1) = approx_step env args sv act in
 approx_nth env args1 sv1 act1 (n-1)

approx_step env args sv act =
 case act of
  Done a -> (args, sv, Done a)
  Pause mf ->
   let rdr = runReaderT mf env in
   let sv0 = execStateT rdr sv in
   let st1 = evalStateT rdr sv in
   let sl = execStateT st1 args in
   let sv = evalStateT sv0 args in
   let act = evalStateT st1 args in
   (runIdentity sl, runIdentity sv, runIdentity act)


-- * Three example programs
-- "Language" semantics
true :: Bf
true = 0
ab_nonzero :: Bf
ab_nonzero = 1
a_ge_b :: Bf
a_ge_b = 2

ide :: Cf
ide = 0
summ :: Cf
summ = 1
mod_fst :: Cf
mod_fst = 2
mod_snd :: Cf
mod_snd = 3
half_sum_fst :: Cf
half_sum_fst = 4
half_sum_snd :: Cf
half_sum_snd = 5
```

```
zero :: Cf
zero = 6
back_sub :: Cf
back_sub = 7

merge_add :: Mf
merge_add = 0

main :: Tf
main = 0
fun1 :: Tf
fun1 = 1
fun2 :: Tf
fun2 = 2

benv :: Be (Int, Int)
benv =  fromList [
 (true, \x -> True),
 (ab_nonzero, \(a,b) -> a /= 0 && b /= 0),
 (a_ge_b, \(a,b) -> a >= b)
 ]

cenv :: Ce (Int, Int)
cenv = fromList [
 (ide, id),
 (summ, \(a,b) -> (a + b, a+b)),
 (mod_fst, \(a,b) -> (a ‘mod‘ b, b)),
 (mod_snd, \(a,b) -> (a, b ‘mod‘ a)),
 (half_sum_fst, \(a,b) -> ((a+b) ‘div‘ 2, b)),
 (half_sum_snd, \(a,b) -> (a, (a+b) ‘div‘ 2)),
 (zero, \(a,b) -> (0,0)),
 (back_sub, \(a,b) -> (b-a, 0))
 ]

menv :: Me (Int, Int)
menv = fromList [
 (merge_add, \(a1, b1) (a2, b2) -> (a1 + a2, b1 + b2))
 ]
```

```
--0: Endless loop; should never halt
-- main:
-- while (true) do id
endless :: Program (Int, Int)
endless = (Env cenv empty benv $
 fromList [(main, While true $ Comp ide)], main)

--1: Simple sequential program; should halt always
-- (GCD (fst, snd)) - halt after 95 steps
-- main a b:
-- while (a != 0 && b != 0)
--  if (a >= b)
--   then a = a % b
--   else b = b % a
-- (a+b, a+b)
gcdd :: Program (Int, Int)
gcdd = (Env cenv empty benv $
 fromList [(main, (Seq (Seq (While ab_nonzero (Cond a_ge_b (Comp mod_fst) (Comp mod_snd)))
    (Comp summ)) Skip))], main)

--2: Integrating; should halt always and give same answers
-- integral of 1 from a to b (in 2 threads, so b - a % 2 == 0)
-- main a b:
-- v1 = spawn integ (a, (a+b)/2)
-- v2 = spawn integ ((a+b)/2, b)
-- (0,0)
-- wait v2 (+, id)
-- wait v1 (+, id)
-- v1 a b:
-- (b-a, 0)
integ :: Program (Int, Int)
integ = (Env cenv menv benv $
 fromList [
  (main,
   Seq
    (Spawn 1 fun1 half_sum_snd empty)
    (Seq
      (Spawn 2 fun1 half_sum_fst empty)
      (Seq
```

```
        (Comp zero)
        (Seq
          (Wait 2 merge_add)
          (Wait 1 merge_add)
    )))),
  (fun1, Comp back_sub)], main)

--3: Seq < Lazy: one of threads doesn't halt
seq_endless :: Program (Int, Int)
seq_endless = (Env cenv menv benv $
 fromList [
  (main, Seq
    (Spawn 1 fun1 half_sum_snd empty)
    (Seq
      (Spawn 2 fun2 half_sum_fst empty)
      (Seq
        (Comp zero)
        (Seq
          (Wait 1 merge_add)
          Skip
    )))),
  (fun1, Comp back_sub),
  (fun2, While true (Comp ide))
  ], main)


run_program :: Program a -> a -> Sched a -> Int -> Maybe a
run_program prog args sched n =
 case (program_sem prog args sched n) of
  (_, _, Done v) -> Just v
  _ -> Nothing
```