САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

На правах рукописи

Лозов Петр Алексеевич

Автоматизированное построение и эффективное исполнение реляционных программ

Научная специальность
2.3.5. Математическое и программное обеспечение
вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание учёной степени кандидата физико-математических наук

Научный руководитель:
доктор технических наук, доцент
Markdown Кознов Дмитрий Владимирович

Санкт-Петербург
2022

Оглавление

Стр.

Введение 4

Глава 1. Обзор предметной области 13
1.1 Реляционное программирование и miniKanren 13
1.2 Методы реляционного исполнения функциональных программ . . 17
1.3 Операционные семантики 23

Глава 2. Реляционное преобразование типизированных функциональных программ 30
2.1 Исходный функциональный язык 31
2.2 Реляционное расширение 35
2.3 Реляционное преобразование 41

Глава 3. Статическая и динамическая корректность реляционного преобразования 47
3.1 Статическая корректность реляционного преобразования 47
3.2 Частичная динамическая корректность реляционного преобразования 50

Глава 4. Семантика языка miniKanren с процедурой динамического управления порядком конъюнктов . . . 59
4.1 Классическая направленная конъюнкция в языке miniKanren . . 60
4.2 Ангелическая семантика и справедливость 64
4.3 Обобщенная семантика языка miniKanren, оснащенная предикатом выбора 70
4.4 Семантика языка miniKanren, оснащенная вполне квазиупорядочивающим предикатом выбора 74

Глава 5. Справедливость конъюнкции в семантике языка miniKanren, оснащенной вполне квазиупорядочивающим предикатом выбора 77
5.1 Сходимость листьев состояния ангелической семантики и семантики, оснащенной квазиупорядочивающим предикатом выбора 77

Стр.

5.2 Сохранение сходимости ангелической семантики 79

5.3 Справедливость конъюнкции в семантике, оснащенной вполне квазиупорядочивающим предикатом выбора 82

Глава 6. Реализация и эксперименты 84

6.1 Реализация 84

6.2 Эксперименты 90

Заключение 95

Список литературы 97

Список рисунков 108

Список таблиц 109

Введение

Актуальность темы. Логическое программирование появилось в конце 60-х годов благодаря исследованиям в области искусственного интеллекта и автоматического доказательства теорем [1—4]. Именно подходы автоматического логического вывода, изначально являющиеся компонентом искусственного интеллекта и систем автоматического доказательства теорем, стали основой логического программирования, благодаря работам R. Kowalski [5] и A. Colmerauer [6]. Современные языки логического программирования, среди которых самым известным является Prolog [7—9], следуют идеям R. Kowalski [10] о разделении любого алгоритма на данные, необходимые для решения проблемы, и стратегию поиска решения, позволяют разработчику сфокусироваться на описании задачи, оставив поиск её решения компьютеру. Это обеспечивает высокую степень абстракции логических языков как на уровне концепта решения задачи, так и непосредственно в коде.

Основным применением языка Prolog является разработка систем символических правил, в которых декларативные знания закодированы в терминах логики первого порядка [11—13]. Язык оптимизирован для выразительности и эффективности такого рода задач, иногда за счет отступлений от базовых идей, основанных на математической логике. В частности, классический Prolog содержит производительную, но некорректную с математической точки зрения унификацию без проверки вхождения (“Occurs Check” [14]). Также для более эффективного и гибкого поиска Prolog использует поиск в глубину [15; 16] и оператор отсечения (“Cut” [17]), вследствие чего поиск решений в классическом языке Prolog не является полным.

Многие другие языки логического программирования стремятся к б*о*льшей “логической чистоте”. Можно упомянуть предметно-ориентированный язык Datalog [18], предназначенный для создания запросов к дедуктивным базам данных. Язык Datalog, в отличие от Prolog, не содержит некорректных с математической точки зрения компонентов, однако, как и многие другие предметно-ориентированные языки, является неполным по Тьюрингу. Следует также указать на язык αProlog [19; 20], позволяющий разрабатывать компиляторы, интерпретаторы и средства доказательства теорем с помощью системы номинальной унификации, и на язык λProlog [21; 22], который создан для эф-

фективного анализа программ, имеет частичную поддержку функций высшего порядка и унификацию высшего порядка. Также отметим реализованный на основе языка Haskell функциональный логический язык Curry [23], обладающий возможностью выбора стратегии поиска [24]. Язык Curry применяется для анализа и синтеза программ, разработки декларативных пользовательских интерфейсов и систем управления базами данных [25—27]. Упомянем типизированный функциональный логический язык Mercury [28], который оснащён системой модов ("Mode System" [29]), позволяющей программисту явно описать входные и выходные параметры, а также определять категорию детерминизма разрабатываемой программы. Эта информация необходима для специализации логической программы с целью оптимизации. Язык Mercury является языком общего назначения и используется для разработки прикладных приложений [30]. Также укажем на логический язык общего назначения Gödel [31], который ориентирован на создание метапрограмм для анализа, трансформации, компиляции, валидации и отладки программ на других языках. Этот язык имеет строгую типизацию, декларативный оператор отсечения и декларативную унификацию. Наконец, отметим язык Twelf [32] с зависимой типизацией, используемый для формализации математических теорий [33] и метатеорий языков программирования [34].

Однако следует отметить, что один из основных компонентов "логической чистоты" — симметричность дизъюнкции и конъюнкции (базовых операторов в языках логического программирования), оказывается нереализованным в этих языках. Следует отметить, что в математической логике эти операции независимы от порядка вычисления аргументов, а в существующих языках логического программирования порядок вычисления конъюнкций и дизъюнкций влияет на сходимость и эффективность процедуры поиска. Операторы с подобным асимметричным поведением будем называть *несправедливыми операторами*.

Для устранения несправедливости дизъюнкции в логическом программировании была выделена отдельная область — реляционное программирование [35]. На данный момент здесь ведутся исследования группой под руководством William E. Byrd (США), а также кафедрой системного программирования СПбГУ под руководством Дмитрия Булычева. Реляционному программированию посвящен ежегодный международный семинар "The miniKanren and Relational Programming Workshop" при известной конференции по языкам программирования ICFP.

Главной отличительной особенностью реляционного программирования является справедливая дизъюнкция, базирующаяся на методе чередования (“Interleaving Search” [36; 37]): процесс поиска решений для каждого дизъюнкта программы разбивается на гарантировано конечные шаги, что позволяет производить полный поиск для произвольного количества дизъюнктов. Справедливость реляционной дизъюнкции позволяет описывать решаемую задачу в виде программы, состоящей из набора декларативных ненаправленных отношений [38; 39]. Основным языком реляционного программирования является miniKanren [35; 40]. Изначально этот язык был минималистичным расширением языков Scheme и Racket [41] и содержал всего пять операторов, а его простейшая реализация составляла менее чем сто строк кода. На текущий момент miniKanren является семейством языков, представители которого содержат различные расширения, призванные увеличить его выразительность и декларативность [42—45].

Разработка реляционных программ является сложной задачей, поскольку от разработчика требуются специфические знания и навыки, а также глубокое понимание семантики каждого реляционного оператора. Было замечено, что на практике разработка реляционной программы происходит в два этапа: создание программы на функциональном языке и последующее “ручное” преобразование этой программы в реляционную. Причем второй этап оказывается механистическим и единообразным, следовательно, он может быть автоматизирован. Таким образом возникает задача создания автоматизированного метода преобразования функциональных программ в реляционные.

Следует отметить, что в общем случае исходная функциональная программа не содержит достаточной информации для однозначного построения её реляционного образа. В частности, при построении такого образа не может быть определён оптимальный порядок конъюнктов, являющийся ключевым фактором эффективности и сходимости процесса исполнения реляционной программы. Важность порядка конъюнктов обусловлена несправедливостью конъюнкции, унаследованной от логического программирования. Поэтому автоматически построенная реляционная программа требует “ручного” редактирования для каждого конкретного реляционного запроса. Однако оптимальный порядок конъюнктов может быть установлен во время исполнения, исходя из текущего состояния реляционной программы. Подобное динамическое управление порядком конъюнктов позволяет повысить эффективность исполнения

автоматически построенных реляционных программ, а также устранить влияние порядка конъюнктов на сходимость процесса исполнения. Исходя из этого, возникает задача создания справедливого метода управления порядком конъюнктов при исполнении реляционных программ.

Степень разработанности темы. W. E. Byrd предложил метод преобразования функциональных программ в реляционные (“Unnesting” [40]). Данный метод основан на преобразовании каждой функции в реляционное отношение путём добавления дополнительного аргумента, устранения вложенных вызовов и замены сопоставления с образцом на дизъюнкцию унификаций с каждым образцом сопоставления. Данный метод позволяет по функциональной программе построить реляционную, однако поддерживает только функции первого порядка, не имеет формального описания и не был реализован. W. Bird, M. Ballantyne, G. Rosenblatt и M. Might рассматривали близкую задачу — исполнение функциональной программы посредством реляционного интерпретатора. Данный подход позволяет реляционно исполнять функциональные программы, посредством внедрения свободных логических переменных в исходную программу. Также предложенный подход позволяет синтезировать функциональные программы по набору тестовых данных [46]. Недостатком данного подхода является низкая эффективность вследствие дополнительного уровня интерпретации.

Проблема эффективного управления порядком конъюнктов в области реляционного программирования многократно рассматривалась [40; 46; 47]. Одним из аспектов несправедливого поведения конъюнкции является приоритизация вычисления независимых дизъюнктов, порождаемых конъюнкцией. В частности, в классической реализации miniKanren б*о*льший приоритет выделяется более ранним дизъюнктам, порождаемым конъюнкцией. Существует, однако, подход, предложенный K.-C. Lu, W. Ma и D. P. Friedman, который балансирует время вычисления различных дизъюнктов, порождаемых конъюнкцией [48]. Это делает конъюнкцию более справедливой, но порядок конъюнктов по-прежнему влияет как на производительность, так и на сходимость вычисления реляционной программы. Кроме того, поведение конъюнкции можно сделать более справедливым, если обнаруживать расходящиеся конъюнкты и откладывать их вычисление. Д. Розплохас и Д. Булычев обнаруживают расхождение конъюнктов во время выполнения реляционной программы и предлагают их динамически переупорядочивать [49]. В этом

случае данные, полученные во время исполнения расходящегося конъюнкта, стираются. Этот подход оказался эффективным на практике, однако консервативная перестановка конъюнктов не использует информацию, полученную при исполнении конъюнкта до перегруппировки. Имеются также примеры, когда в рамках этого подхода порядок конъюнктов все еще влияет на сходимость.

В более широком контексте подобные задачи также рассматривались. Стоит отметить, что многие результаты, посвященные проблемам управления порядком вычисления в Prolog, связаны с преодолением присущей ему неполноты поиска в глубину. Например, для лучшего контроля структуры дерева поиска T. Schrijvers, M. Triska и B. Demoen предложили более справедливую дизъюнкцию [50]. Для обеспечения полноты поиска в Prolog в ряде случаев используется табулирование (Tabling) [51—53], которое, однако, не является универсальным решением ввиду больших накладных расходов этого подхода. Отметим, что многие проблемы логического программирования по управлению порядком вычисления не актуальны в случае реляционного программирования, поскольку для последнего имеется полная процедура поиска [54; 55].

Существует определенное сходство между проблемами, которые мы решаем, и проблемами, возникающими в многопоточных реализациях Prolog. G. Gupta, E. Pontelli с коллегами предложили "зависимый И-параллелизм" ("Dependent And-parallelism"), который позволяет исполнять конъюнкты параллельно и имеет дело с эффектами, вызванными изменчивостью исполнения конъюнктов в зависимости от порядка [56]. Однако, в отличие от нашего случая, основные усилия здесь направлены на повышение производительности при сохранении семантики обхода в глубину слева направо.

Таким образом, исследуемая нами задача эффективного реляционного преобразования и исполнения функциональных программ специфична именно для реляционного программирования, а её решение позволит значительно снизить сложность разработки реляционных программ.

Целью данной работы является создание подхода к построению и исполнению реляционных программ посредством реляционного преобразования функциональных программ и динамического управления порядком исполнения реляционных конъюнктов.

Для достижения поставленной цели были поставлены следующие **задачи**.

1. Разработка метода реляционного преобразования типизированных функциональных программ общего вида в реляционные.

2. Доказательство статической и динамической корректности реляционного преобразования.
3. Разработка семантики языка miniKanren с процедурой динамического управления порядком вычисления конъюнктов.
4. Доказательство справедливости конъюнкции для предложенной семантики.

Основные положения, выносимые на защиту.

1. Предложен новый метод реляционного преобразования функциональных программ общего вида, доказана его статическая и динамическая корректность.
2. Введена формальная ангелическая семантика реляционного языка miniKanren, доказана эквивалентность этой семантики декларативной семантике языка miniKanren. Введено понятие справедливости конъюнкции как свойства ангелической семантики.
3. Определена формальная семантика реляционного языка miniKanren с процедурой динамического управления порядком вычисления конъюнктов, доказана справедливость конъюнкции в данной семантике.
4. Выполнена реализация на языке OCaml реляционного преобразования для подмножества функционального языка OCaml, а также проведено экспериментальное исследование, показавшее высокую эффективность автоматически полученных реляционных программ при использовании справедливой конъюнкции.
5. Выполнено встраивание языка miniKanren с процедурой динамического управления порядком конъюнктов в функциональный язык OCaml и проведено экспериментальное исследование, демонстрирующее высокую эффективность процедуры в сравнении с классической реализацией miniKanren при неоптимальном порядке конъюнктов и незначительность накладных при оптимальном порядке конъюнктов.

Научная новизна результатов, полученных в рамках исследования, заключается в следующем.

1. Впервые представлен метод реляционного преобразования функциональных программ общего вида, для которого была доказана статическая и динамическая корректность.

2. Впервые формально описана ангелическая семантика реляционного языка miniKanren, позволившая впервые ввести понятие справедливости конъюнкции для реляционного программирования.
3. Впервые описана формальная семантика реляционного языка miniKanren с процедурой динамического управления порядком конъюнктов.

Практическая значимость работы заключается в создании и реализации метода реляционного преобразования функциональных программ общего вида. Этот метод может быть использовано для упрощения создания реляционных программ вплоть до полного исключения "ручной" разработки реляционного кода. Также практической значимостью обладает представленный в данном диссертационном исследовании метод исполнения реляционных программ с процедурой динамического управления порядком конъюнктов, позволяющий эффективно исполнять реляционные программы и исключить необходимость редактирования программ для различных реляционных запросов.

Методология и методы исследования. Методология исследования базируется на парадигме формального подхода к описанию семантики языков программирования. В работе используется классические методы описания семантик большого и малого шагов в виде набора правил вывода. В работе также используются базовые концепции логического программирования — метод унифицирования логических выражений, автоматический поиск решений и недетерминированное исполнение. Также в работе использован аппарат реляционного программирования: операторы унификации и ограничения неравенством в качестве основного инструмента построения решений и полный поиск решений методом чередования. Программная реализация теоретических результатов выполнена на функциональных языках программирования OCaml, Haskell и Scheme.

Достоверность полученных результатов исследования обеспечивается формальными доказательствами, а также компьютерными экспериментами. Полученные результаты согласуются с результатами, установленными другими авторами.

Апробация работы. Основные результаты работы докладывались на следующих научных мероприятиях: на конференции PLC 2017 (3–5 апреля 2017 г., Ростов-на-Дону, Россия), на симпозиуме TFP 2017 (19-21 июня 2017 г.,

Кентербери, Великобритания), на семинаре ML 2017, совмещенном с конференцией ICFP 2017 (3-9 сентября 2017 г., Оксфорд, Великобритания), на семинаре miniKanren 2019, совмещенном с конференцией ICFP 2019 (18-23 сентября 2019 г., Берлин, Германия), на семинаре miniKanren 2020, совмещенном с конференцией ICFP 2020 (20-28 августа 2020 г., Нью-Джерси, США), на симпозиуме APLAS 2020 (30 ноября - 2 декабря 2020 г., Фукуока, Япония), на семинаре PEPM 2021, совмещенном с симпозиумом POPL 2021 (17-22 января 2021 г., Копенгаген, Дания).

Личный вклад автора в публикациях, выполненных с соавторами, распределён следующим образом. В работах [57; 58] автор разработал метод реляционного преобразования функциональных программ, выполнил доказательство статической и динамической корректности преобразования, а также выполнил реализацию на языке OCaml [59] и постановил эксперименты. Соавторы участвовали в обсуждении идей, формализации метода, а также улучшили текст статьи. В работе [60] автор участвовал в создании основного подхода, представленного в статье, а также разработал функциональный интерпретатор для подмножества языка OCaml, реляционный образ которого позволил решать задачу поиска. Соавторы предложили идею использования реляционных интерпретаторов для решения задач поиска и адаптировали метод конъюнктивной частичной дедукции для случая реляционных программ. В работе [61] автор предложил и формализовал семантику miniKanren с направленной конъюнкцией на основе развёртки вызовов, семантику miniKanren с процедурой динамического управления порядком конъюнктов на основе свободных переменных в аргументах вызовов, а также выполнил реализацию на языке Haskell и провёл эксперименты. Соавтор участвовал в обсуждении основных идей, в работах по формализации семантики, а также улучшил текст статьи. В работе [62] вклад автора заключается в формализации ангелической семантики для miniKanren, формализации семантики miniKanren с направленной конъюнкцией, доказательстве справедливости конъюнкции в предложенной семантике, выполнении реализации на языке Haskell и проведении экспериментов. Соавтор предложил использовать ангелическую семантику для определения справедливости конъюнкции, участвовал в доказательстве справедливости конъюнкции и улучшил текст статьи. В работе [63] автор предложил использовать реляционное преобразование с целью создания реляционного интерпретатора для

сопоставления с образцом, участвовал в обсуждении остальных идей статьи. Соавторы разработали метод и провели эксперименты.

Публикации. Основные результаты по теме диссертации изложены в 6 научных работах, 1 из которых издана в журнале, рекомендованном ВАК, 3 — в периодических научных журналах, индексируемых Web of Science и Scopus.

Объем и структура работы. Диссертация состоит из введения, 6 глав, заключения.

Полный объём диссертации составляет 109 страниц, включая 16 рисунков и 2 таблицы. Список литературы содержит 105 наименований.

Благодарности. Прежде всего я бы хотел поблагодарить Дмитрия Юрьевича Булычева за возможность познать красоту реляционного программирования, руководство на ранних этапах данного исследования, неоценимый вклад в мою работу, готовность поддержать и поделиться своим опытом. Я благодарю Уильяма Бёрда за само создание реляционного программирования, многочисленные беседы о моей диссертации и помощь в исследованиях.

С большой теплотой хочу сказать слова благодарности своему научному руководителю, Дмитрию Владимировичу Кознову, за его неиссякаемую энергию, проницательность и дальновидность, которыми он делился на протяжении всей нашей работы.

Я выражаю благодарность Андрею Николаевичу Терехову и кафедре системного программирования СПбГУ, а также компаниям JetBrains и Huawei за уникальную возможность заниматься наукой как основной деятельностью.

Я благодарен Дмитрию Сергеевичу Косареву за его бесценную поддержку и помощь в практической составляющей данной работы. Отдельную благодарность хочется выразить моей невесте, Анастасии Андреевной Садыковой, ведь она дарует мне вдохновение и энергию для всего, что я делаю. Наконец, я благодарен родителям, которые являются моей опорой на протяжении моей жизни и проявляют неиссякаемый интерес к моей научной деятельности.

Глава 1. Обзор предметной области

В данной главе приведен обзор реляционного языка miniKanren, приведены примеры его использования, а также рассмотрены существующие подходы к реляционному исполнению функциональных программ. Кроме того, введены основные понятия, используемые в данной диссертационной работе — операционная семантика большого и малого шага, различные виды операционных семантик в нотации Матиаса Феллейсена [64].

1.1 Реляционное программирование и miniKanren

Главной отличительной особенностью языка miniKanren [35; 40] является ненаправленные вычисления реляционных программ: аргументы и результат реляционных отношений не различаются на уровне синтаксиса, что позволяет описывать ненаправленные программы и исполнять их в различных "направлениях". Подобный подход оказывается полезен на практике, ведь задачи формулируются гораздо проще в виде запросов к ненаправленным реляционным программам. Это наблюдение подтверждает целый ряд примеров. В качестве иллюстрации приведём задачу вывода типов для просто типизированного лямбда-исчисления (Simply Typed Lambda Calculus [65]), а также задачу о выявлении населенности какого-либо типа (Type Inhabitation [66]). Эти задачи выразимы в форме запросов к более простой в реализации реляционной программе, проверяющей корректность типизации лямбда-выражения.

Еще одной иллюстрацией является задача построения "квайнов" [38] — программ, исполнение которых возвращает точную копию их исходного текста. Эта задача представима в виде запроса к реляционному интерпретатору языка, на котором необходимо построить квайн. Причем разработка реляционного интерпретатора является более простой задачей по сравнению с исходной задачей построения квайнов. Наконец, решение задачи построения всех перестановок элементов заданного списка может быть выражено в виде запроса к более простой в реализации реляционной сортировке списка.

В контексте данной работы мы будем рассматривать конкретную реализацию языка miniKanren, которая называется OCanren [67] и является расширением функционального языка OCaml [59; 68]. Язык OCanren соответствует классической реализации языка miniKanren [41; 69] с ограничением неравенством [70]. От классической реализации языка miniKanren данная версия отличается наличием строгой типизации, позволяющей на этапе компиляции исключить ряд ошибок, допускаемых при разработке реляционных программ.

Далее мы изучим язык OCanren с точки зрения пользователя, рассматривая только интуитивное определение его конструкций; формальное описание OCanren будет представлено в главе 2. Также в этой и последующих главах мы будем использовать упрощенный синтаксис, который незначительно отличается от актуального синтаксиса реализации OCanren с целью упрощения его восприятия читателем. Мы оставим без рассмотрения операторы исполнения реляционного запроса `run N` и `run`*, двухпараметрический тип представления логических выражений, полностью полиморфные типы данных (Fully-Polimorfic Datatypes), необходимые для одновременного использования выражений в функциональном и логическом доменах и др. Более подробно и полно синтаксис языка OCanren представлен в работе [67].

Основное понятие языка miniKanren является *цель*. В языке OCanren цель может быть произвольным выражением зарезервированного типа цели $\mathfrak{G}$. Результатом вычисления цели является поток данных (возможно бесконечный), содержащий ответы, удовлетворяющие условиям цели. Существует всего пять синтаксических форм целей, обозначаемых ниже как g, g_1, g_2 и т. д.

- Унификация — первый базовый оператор для создания целей, который определяется как $t_1 \equiv t_2$, где t_1 и t_2 являются некоторыми термами, состоящими из конструкторов и переменных [71]. Если термы t_1 и t_2 могут быть унифицированы, тогда цель считается успешной, и в этом случае результатом унификации является одноэлементный поток, содержащий наибольший общий унификатор термов t_1 и t_2. В противном случае цель считается неуспешной, и её результатом является пустой поток.
- Оператор ограничения неравенством необходим для построения целей и имеет обратное унификации поведение. Он определяется как $t_1 \not\equiv t_2$, где t_1 и t_2 являются некоторыми термами.

- Дизъюнкция — это оператор вида $g_1 \vee g_2$, где g_1 и g_2 являются целями, и обе цели выполняются независимо (так называемая справедливая дизъюнкция). Результатом исполнения дизъюнкции является объединение ответов целей g_1 и g_2.
- Конъюнкция — это оператор вида $g_1 \wedge g_2$, где g_1 и g_2 являются целями. При выполнении конъюнкции в первую очередь исполняется цель g_1, затем цель g_2 исполняется в контексте каждого из ответов цели g_1 (таким образом, конъюнкция не является справедливой). В результате получаем поток ответов, удовлетворяющих как цели g_1, так и цели g_2.
- Оператор для введения свежей переменной определяется как `fresh` (x) g, где x является некоторой переменной, а g — некоторой целью. Данный оператор необходим для введения переменной x, отсутствующей в текущем контексте, которая используется в цели g (далее в данной работе будем называть такие переменные *свежими*).

Термы, используемые в операторах унификации и ограничениях неравенством, являются произвольными выражениями полиморфного логического типа α^o. Постфикс $\Box^o$ является традиционным способом обозначения реляционных сущностей, и мы будем использовать его и для имен отношений, и для типов[1].

Простейшее выражение логического типа — это переменная, введенная оператором `fresh`. Другим примером является примитивное выражение, введённое в логический домен с помощью встроенного примитива "$\uparrow$", такое как $\uparrow 3$ (его тип $\texttt{int}^o$) или $\uparrow$`true` (его тип $\texttt{bool}^o$). Прочие типы (кортежи, списки, пользовательские алгебраические типы данных и др.) также могут быть использованы в реляционных программах, если их внести в логический домен с помощью того же примитива. К примеру, выражение $\uparrow$`(1, "abc")` имеет тип `(int * string)`^{o}, а выражение $\uparrow$`[1; 2; 3]` имеет тип `(int list)`^{o}. Однако, так как унификация и ограничение неравенством рекурсивны и работают только с выражениями логического типа, логический тип "^{o}" должен быть применен ко всем элементам типа. Действительно, логическая переменная может быть расположена только на позиции, где ожидается логический тип. Таким образом,

[1] В актуальной реализации языка OCanren термы имеют более сложный двухпараметрический тип, который кодирует тегирование, необходимое для выполнения унификаций, и преобразования результатов реляционной программы в функциональную форму для дальнейшего использования в функциональном контексте; эти детали, однако, не имеют отношения к целям данной работы, и мы будем придерживаться упрощенной версии.

при унификации можно использовать значение типа $(\texttt{int * int})^o$ как *целое* значение, но для реляционного управления *содержимым* пары необходим тип $(\texttt{int}^o,\ \texttt{int}^o)^o$. Это делает невозможным использование некоторых встроенных и стандартных типов в реляционном коде — например, предопределенный тип списка недостаточно гибок, так как в стандартном определении хвост списка не дополнен типом логического домена. Вместо этого необходимо ввести тип логического списка:

```
type α llist = [] | (::) of αᵒ * (α llist)ᵒ
```

С помощью данного определения мы можем реализовать различные отношения над списками, например:

```
val append : (α llist)ᵒ → (α llist)ᵒ → (α llist)ᵒ → 𝔊
let rec appendᵒ = λ x y xy .
  (x ≡ ↑[]  ∧  xy ≡ y)  ∨
  fresh (h t ty)
     x  ≡ ↑(h :: t)  ∧
     xy ≡ ↑(h :: ty)  ∧
     appendᵒ t y ty
```

В этом примере мы определили тернарное отношение конкатенации реляционных списков $\texttt{append}^o$, канонический пример в области реляционного программирования. Это отношение построено с использованием разбора случаев и рекурсии.

1. Если первый список пуст, то второй и третий списки должны быть равны.
2. В противном случае первый список можно разбить на голову и хвост, и для их обозначения нужны две свежие переменные h и t. Нам также нужна новая переменная ty для обозначения списка, который будет равен конкатенации списков y и t. Чтобы обеспечить это, мы используем рекурсивный вызов $\texttt{append}^o$. Наконец, мы получаем окончательный результат, объединяя h и ty.

Определение $\texttt{append}^o$ имеет в качестве аргументов три логических списка `x`, `y` и `xy` и описывает цель, которая может быть исполнена или скомбинирована с другими целями. Результатом вычисления будет поток ответов, при этом каждый элемент потока содержит описание ограничений для логических

переменных, которые необходимо соблюсти для принадлежности аргументов отношению.

Обозначим примитив исполнения реляционного запроса символом “$\rightsquigarrow$”. Тогда запрос

$$\underline{\texttt{fresh}}\ (q)\ \texttt{append}^o\ \uparrow(\uparrow\texttt{1}\ \texttt{::}\ \uparrow\texttt{[]})\ q\ \uparrow\texttt{[]}\ \rightsquigarrow\ \texttt{[]}$$

вернёт пустой поток, так как не существует списка q, присоединение к которому списка (1 :: []) приведет к пустому списку. В то же время вычисление запроса

$$\underline{\texttt{fresh}}\ (q)\ \texttt{append}^o\ q\ \uparrow\texttt{[]}\ \uparrow(\uparrow\texttt{1}\ \texttt{::}\ \uparrow\texttt{[]]})\ \rightsquigarrow\ \texttt{[}q \mapsto\ \texttt{1 :: []]}$$

вернёт ожидаемое ограничение для переменной q.

Как видно из типа отношения, реляционная конкатенация полиморфна, также как и ее функциональный аналог. Однако запрос

$$\underline{\texttt{fresh}}\ (q)\ \texttt{append}^o\ \uparrow(\uparrow\lambda x.x\ \texttt{::}\ \uparrow\texttt{[]]})\ q\ \uparrow(\uparrow\lambda y.y\ \texttt{::}\ \uparrow\texttt{[]]})$$

заканчивается ошибкой времени выполнения из-за невозможности унифицировать выражения высшего порядка. Это фундаментальное ограничение, имеющееся и в оригинальном miniKanren, где используется синтаксическая унификация первого порядка [71]. Данный пример демонстрирует, что, в отличие от чистого OCaml, типизация в OCanren слабее. Чтобы восстановить строгую типизацию, некоторые переменные типа должны быть ограничены только значениями первого порядка. Отсутствие прямой поддержки ограниченного полиморфизма [72] в OCaml делает проверку этого ограничения проблематичным. Однако наш опыт показывает, что на практике этот недостаток редко приводит к ошибкам при разработке реляционных программ. В дальнейшем мы предполагаем, что в полиморфных типах некоторые переменные типа могут быть неявно ограничены множеством типов первого порядка, и эти ограничения соблюдаются во всех экземплярах этих переменных типа.

1.2 Методы реляционного исполнения функциональных программ

Как было сказано в предыдущем разделе, реляционное программирование позволяет эффективно решать многие задачи посредством ненаправленных реляционных вычислений. Однако написание подобных программ является

нетривиальной задачей, которая включает в себя оптимизацию отношений для всевозможных вариантов исполнения, построение эффективных недетерминированных вычислений и т.д.

В связи с этим актуальной является возможность реляционного исполнения программ, написанных на других языках. Наличие такой возможности позволит использовать для разработки программ привычные языки программирования (прежде всего — функциональные в силу их схожести с реляционными), которые затем будут исполнены реляционно. Данный подход позволяет, к примеру, моделировать вычисление обратных функций без прямого использования реляционного программирования. Это облегчает разработку в случае, когда решение обратной задачи значительно проще в сравнении с решением исходной задачи.

На текущий момент существует два подхода к реляционному исполнению функциональных программ: использование реляционного преобразования [40; 57], создающего реляционную программу, эквивалентную исходной функциональной, а также реляционное исполнение функциональных программ с помощью реляционных интерпретаторов [46].

1.2.1 Синтаксическое реляционное преобразование функциональных программ первого порядка

Для построения реляционной программы по функциональной был предложен метод по названием “Unnesting” [35; 40]. Данный метод не был реализован, поэтому в данной работе мы рассмотрим его в качестве преобразования, проводимого вручную.

Данное преобразование основано на введении новых переменных для каждого вложенного применения функции. После того, как каждое подвыражение будет связано с переменной, все сопоставления с образцом необходимо преобразовать в дизъюнкцию. Далее, все введенные ранее переменные, а также переменные, используемые в сопоставлениях с образцом, следует объявить посредством оператора `fresh`. Наконец, каждому применению функции необходимо добавить в качестве дополнительного аргумента соответствующую ему переменную, а каждую преобразуемую функцию дополнить аргументом, кото-

```
let rec append = λ x y.
  match x with
  | []  →  y
  | h :: t  →
     h :: append t y
```

а)

```
let rec append = λ x y.
  match x with
  | []  →  y
  | h :: t  →
     let ty = append t y in
     h :: ty
```

б)

```
let rec appendᵒ = λ x y xy.
  (x ≡ ↑[]  ∧  xy ≡ y)  ∨
  (fresh (h t ty)
    x  ≡ ↑(h :: t)  ∧
    xy ≡ ↑(h :: ty)  ∧
    appendᵒ t y ty)
```

в)

Рис. 1.1 — Пример выполнения ручного преобразования

рый необходимо унифицировать с результатом вычисления. Также в случае языка OCanren каждый конструктор в исходной программе необходимо дополнить вспомогательным конструктором $\uparrow$ для переноса всех конструкторов в логический домен.

В качестве иллюстрации преобразования вновь рассмотрим функцию конкатенации двух списков (рис. 1.1, а). В этом примере представлена классическая реализация поэлементно перемещает элементы из первого списка во второй и в итоге возвращает список, содержащий все элементы исходных списков. После добавления имен для всех вложенных применений функций — в данном примере, это переменная ty для единственного применения `append` t y — получим функцию, эквивалентную исходной (рис. 1.1, б). Далее выполним основной шаг преобразования: заменим сопоставление с образцом на дизъюнкцию; добавим дополнительный аргумент xy и добавим унификацию этой переменной с результатом в каждом дизъюнкте; все переменные сопоставления с образцом и введенную для вложенного применения переменную ty объявим с помощью оператора fresh; применим вложенный вызов к соответствующей ему переменной ty и получим итоговую реляционную программу (рис. 1.1, в).

```
let bar = λ x.
  let f = λ x. x in
  let g = λ a. f in
  g A y
```

а)

```
let barᵒ = λ y r.
  let f = λ x r. x ≡ r in
  let g = λ a r = f ≡ r in
  g ↑A y r
```

б)

Рис. 1.2 — Некорректный случай для ручного преобразования

Однако вследствие синтаксической природы преобразования не каждое определение можно преобразовать в реляционную форму с помощью данного метода. Это преобразование корректно работает лишь с функциями первого порядка. В случае использования функций высшего порядка данный метод может создать некорректную реляционную программу. Рассмотрим, например, определение на рис. 1.2, а (эта программа необходима лишь для иллюстрации работы преобразования и носит искусственный характер). Рассматриваемое преобразование выдаст программу, представленную на рис. 1.2, б. Очевидно, что этот результат является некорректным, так как полученное отношение содержит унификацию функции f и логической переменной r. Чтобы в данном случае этот метод построил корректную реляционную программу, необходимо в качестве предварительного шага использовать η-расширение к определению g, синтаксически явно отображая его функциональную природу.

1.2.2 Реляционное исполнение функциональных программ с помощью реляционных интерпретаторов

Реляционные интерпретаторы — мощный и гибкий инструмент для частичного или полного синтеза программ благодаря возможности внедрения в интерпретируемую программу логических переменных. Это позволяет синтезировать как части программ, так и небольшие программы целиком.

Пусть $\texttt{eval}^o$ является некоторым реляционным интерпретатором, который принимает в качестве аргументов интерпретируемую программу, входные данные и ожидаемый результат. Тогда мы можем непосредственно исполнить

некоторую программу $PROG$ при входных данных IN с помощью следующего запроса:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o\ PROG\ IN\ q.$$

Этот запрос свяжет переменную q с результатом интерпретации. Однако, помимо интерпретации программ, доступной в любом другом интерпретаторе, мы можем использовать реляционный интерпретатор для различных задач синтеза и проверки программ. К примеры, мы можем синтезировать программу по набору тестовых данных $(IN_1, OUT_1), \ldots (IN_n, OUT_n)$ посредством следующего запроса:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o\ q\ IN_1\ OUT_1\ \land\ \ldots\ \land\ \texttt{eval}^o\ q\ IN_n\ OUT_n.$$

Результатом данного запроса будут все программы, которые на входах IN_i возвращают ответы OUT_i. Если же на место переменной q подставить частичную программы, мы получим синтез подвыражений программы. В случае полной программы мы получим проверку выполнения тестовых данных.

Помимо этого, посредством реляционного интерпретатора мы можем решить нетривиальную задачу синтеза "квайнов". Для этого достаточно описать следующую программу, которая в качестве ответа возвращает саму себя:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o\ q\ \texttt{()}\ q.$$

Данный запрос вернет возможно бесконечную коллекцию всех квайнов для интерпретируемого языка.

В контексте данной работы наиболее важным применением реляционных интерпретаторов является реляционное исполнение функциональных программ. Действительно, в зависимости от расположения логических переменных в аргументах входных данных и результата интерпретации мы можем исполнить функциональную программу в различных направлениях. К примеру рассмотренную ранее функцию конкатенации двух списков `append`, можно исполнить в прямом направлении (для упрощения чтения в данном запросе

программа `append` представлена в синтаксисе языка OCaml; на практике необходимо представить эту функцию в виде синтаксического дерева):

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o \left(\begin{array}{l} \underline{\texttt{let}}\ \underline{\texttt{rec}}\ \texttt{append}\ \texttt{=}\ \lambda\ x\ y. \\ \quad \underline{\texttt{match}}\ x\ \underline{\texttt{with}} \\ \quad \texttt{[]}\ \rightarrow\ y \\ \quad h\ \texttt{::}\ t \rightarrow \\ \quad\quad h\ \texttt{::}\ \texttt{append}\ t\ y \end{array} \right) (\texttt{[1]},\ \texttt{[2]})\ q$$

Ответом на этот запрос будет [1; 2] — результат конкатенации списков [1] и [2]. Также мы можем исполнить функцию `append` в обратном направлении:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o \left(\begin{array}{l} \underline{\texttt{let}}\ \underline{\texttt{rec}}\ \texttt{append}\ \texttt{=}\ \lambda\ x\ y. \\ \quad \underline{\texttt{match}}\ x\ \underline{\texttt{with}} \\ \quad \texttt{[]}\ \rightarrow\ y \\ \quad h\ \texttt{::}\ t \rightarrow \\ \quad\quad h\ \texttt{::}\ \texttt{append}\ t\ y \end{array} \right) (\texttt{[1]},\ q)\ \texttt{[1; 2]}$$

В результате мы получим список [2], который необходимо добавить к списку [1], чтобы получить список [1; 2].

В предыдущих примерах мы описали детерминированные запросы. Однако возможно и недетерминированное исполнение интерпретируемой программы. В качестве иллюстрации рассмотрим запрос:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o \left(\begin{array}{l} \underline{\texttt{let}}\ \underline{\texttt{rec}}\ \texttt{append}\ \texttt{=}\ \lambda\ x\ y. \\ \quad \underline{\texttt{match}}\ x\ \underline{\texttt{with}} \\ \quad \texttt{[]}\ \rightarrow\ y \\ \quad h\ \texttt{::}\ t \rightarrow \\ \quad\quad h\ \texttt{::}\ \texttt{append}\ t\ y \end{array} \right) (p,\ q)\ \texttt{[1; 2]}$$

Данный запрос описывает все пары списков, конкатенация которых равна списку [1; 2]. Результатом его исполнения будет коллекция всех таких пар [([], [1; 2]); ([1], [2]); ([1; 2], [])].

Как можно видеть, этот подход имеет множество применений и, среди прочего, позволяет реляционно исполнить функциональную программу. Однако вследствие дополнительного уровня интерпретации подобное исполнение обладает низкой производительностью по сравнению с исполнением реляционных

программ, полученных с помощью реляционного преобразования функциональных программ.

1.3 Операционные семантики

В теории формальных языков формальные семантики являются классическим методом описания “смысла” программ. Существуют различные способы представления формальных семантик. *Денотационные семантики* сопоставляют синтаксическому представлению программы математические объекты, при этом абстрагируясь от процесса исполнения программ [73]. *Аксиоматические семантики* состоят из набора аксиом, посредством которых осуществляется вывод результатов выполнения программы [74]. Для представления процесса исполнения программы в виде набора правил перехода применяются *операционные семантики* [75]. В контексте данной работы для описания семантик функционального и реляционного языков использованы операционные семантики, поскольку данное представление наиболее подробно описывает процесс выполнения программы. Это позволяет сохранить соответствие между теоретическим описанием языка в виде операционной семантики и практической реализацией в виде интерпретатора.

Выделяют два класса операционных семантик: семантики большого шага (также называемые естественными семантиками) и семантики малого шага (также называемые структурными операционными семантиками). Кроме того важными подклассом семантик малого шага являются семантики в нотации Матиаса Феллейсена [64], другое название которых — редукционные семантики.

В данной работе мы используем операционные семантики всех трёх классов. Поэтому далее мы рассмотрим каждый класс подробнее. В качестве примера языка, для которого мы опишем различные виды операционных семантик, будет использован язык бинарных арифметических выражений, которой достаточно лаконичен, но позволит нам продемонстрировать основные особенности различных классов операционных семантик. Для упрощения примеров мы будем использовать только одну бинарную операцию — сложение, так как семантика остальных операторов определяется сходным образом. Синтаксис

языка арифметических выражений включает натруральные числа, переменные и бинарный оператор сложения $L = \mathbb{N} \mid \mathbb{X} \mid L + L$.

Также для вычисления значений переменных в каждой из семантик необходимо в качестве окружения определить отображение из имен переменных в натуральные числа $\sigma : \mathbb{X} \Rightarrow \mathbb{N}$.

Наконец, операция сложения обозначается “+”. Семантическая функция сложения обозначается “$\oplus$” и принимает два натуральных числа и возвращает также натуральное число, являющееся результатом сложения.

Теперь у нас есть всё необходимое для определения операционных семантик. В следующих разделах мы введём семантику большого шага, семантику малого шага и семантику в нотации Феллейсена и рассмотрим их отличительные особенности.

1.3.1 Семантики большого шага

Операционная семантика большого шага была введена Жилем Каном для представления языка Mini-ML, являющегося диалектом функционального языка ML [76]. Главной отличительной особенностью семантики большого шага является прямое отображение программы в результат. Поэтому подобные семантики определяются отношением на множестве программ и в определенном *семантическом домене*, элементы которого определяют всевозможные результаты выполнения программы. В случае рассматриваемого примера семантическим доменом является множество натуральных чисел.

Отношение семантики большого шага, как правило, представляется в виде рекурсивного набора правил вывода. Причем каждое правило сопоставляет окружению и программе, соответствующей заданному шаблону, результат вычисления этой программы. Также часть правил содержит ряд условий, которым должны удовлетворять подвыражения программы. Правила без условий называются аксиомами.

Семантика большого шага (обозн. “$\Rightarrow$”) для языка арифметических выражений представлена на рис. 1.3. Данная семантика состоит из трёх правил вывода. Аксиома для констант $\textsc{Num}_B$ сопоставляет константе её же значение. Вторая аксиома для переменных $\textsc{Var}_B$ с помощью окружения σ заменяет пе-

$$\frac{}{(\sigma, n) \Rightarrow n} \quad [\textsc{Num}_B]$$

$$\frac{}{(\sigma, x) \Rightarrow \sigma(x)} \quad [\textsc{Var}_B]$$

$$\frac{(\sigma, e_1) \Rightarrow n_1 \qquad (\sigma, e_2) \Rightarrow n_2}{(\sigma, e_1 + e_2) \Rightarrow n_1 \oplus n_2} \quad [\textsc{Add}_B]$$

Рис. 1.3 — Семантика большого шага для языка арифметических выражений

ременную на её значение. Последнее правило для сложения $\textsc{Add}_B$ требует вычисления значений каждого слагаемого и определяет в качестве результата сумму этих значений с помощью функции “$\oplus$”.

Как можно видеть, преимуществом семантики большого шага является простота представления: для её описания требуется меньшее количество правил вывода в сравнении с другими операционными семантиками. В силу этого более простыми становятся и доказательства различных свойств семантики, например доказательства корректности и полноты.

Однако в семантике большого шага для расходящихся вычислений не существует деревьев вывода, что делает невозможным определение и доказательство свойств таких вычислений. Также семантика большого шага не дает полного контроля над порядком вычисления. К примеру, в правиле вывода AB порядок вычисления слагаемых не задан. В тоже время управление порядком вычисления необходимо при определении семантики языка: например, в случае λ-исчисления различные стратегии вычисления аргументов функций, а именно вызов по значению (call by value) и вызов по имени (call by name), выражаются именно через упорядочивание вычисления аргументов и тел функций [77]. Для более детального определения семантики языка используют операционные семантики малого шага.

1.3.2 Семантики малого шага

Семантика малого шага была введена Гордоном Плоткиным [78]. В данной нотации вместо сопоставления программе результата её вычисления мы итеративно упрощаем программу, вычисляя только некоторое её подвыражение.

$$\frac{}{(\sigma, x) \to (\sigma, \sigma(x))} \quad [\textsc{Var}_S]$$

$$\frac{}{(\sigma, n_1 + n_2) \to (\sigma, n_1 \oplus n_2)} \quad [\textsc{AddV}_S]$$

$$\frac{(\sigma, e_1) \to (\sigma, \widetilde{e_1})}{(\sigma, e_1 + e_2) \to (\sigma, \widetilde{e_1} + e_2)} \quad [\textsc{AddL}_S]$$

$$\frac{(\sigma, e_2) \to (\sigma, \widetilde{e_2})}{(\sigma, n_1 + e_2) \to (\sigma, n_1 + \widetilde{e_2})} \quad [\textsc{AddR}_S]$$

Рис. 1.4 — Семантика малого шага для языка арифметических выражений

Семантика малого шага также представляется в виде набора правил вывода, в каждом из которых исходной программе сопоставляется упрощенная версия этой программы. Для полного вычисления программы необходимо итеративно применять правила семантики, пока хотя бы одно из них применимо. Данный процесс определяется рефлексивно-транзитивным замыканием отношения, определяемого правилами вывода семантики.

Семантика малого шага (обозн. “$\to$”) для языка арифметических выражений представлена на рис. 1.4 и состоит из четырех правил вывода. Первое правило для переменных $\textsc{Var}_S$ является аксиомой и, как в случае семантики большого шага, сопоставляет переменной её значение. Отметим, что для итеративного применения семантики окружение σ необходимо перенести и в результат выполнения правила вывода. Второе правило, применяемое для сложения вычисленных слагаемых $\textsc{AddV}_S$, также является аксиомой и выполняет сложение. Правило упрощения левого слагаемого $\textsc{AddL}_S$ требует выполнить шаг семантики для левого слагаемого, если оно не является константой. После этого упрощенная версия выражения заменяет исходное левое слагаемое. Последнее правило упрощения правого слагаемого $\textsc{AddR}_S$ требует, чтобы левое слагаемое было константой, а правое слагаемое — нет. В этом случае будет упрощено правое слагаемое. Отметим, что в данной семантике нет правила, соответствующего натуральному числу — так как число является программой, которая не требует никаких действий для вычисления, в отдельном правиле нет необходимости.

Выразим итоговую семантику с помощью рефлексивно-транзитивного замыкания. Пусть p — некоторое арифметическое выражение, σ — некоторое

окружение и n — некоторое натуральное число. Тогда выражение p в окружении σ вычислимо в семантике "$\rightarrow$" и результатом вычисления является значение n, если верно, что $(\sigma, p) \rightarrow^* (\sigma, n)$, где "$\rightarrow^*$" — рефлексивно-транзитивное замыкание отношения "$\rightarrow$".

Прежде всего отметим, что в данной семантике больше правил в сравнении с семантикой большого шага для одного и того же языка. В следствие этого доказательство большинства свойств семантики малого шага будет более объёмным и сложным в сравнении с аналогичным доказательством для семантики большого шага. С другой стороны, данная семантика за счет трёх правил для сложения строго определяет порядок вычисления подвыражений: сначала левое слагаемое, затем правое. В случае λ-исчисления именно эта особенность позволяет описать различные стратегии вычисления. Также заметим, что расходящиеся вычисления в данной семантике имеют хоть и бесконечную, но структурированную цепочку вывода, что позволяет описывать и доказывать свойства расходящихся программ.

Однако, в данной семантике один шаг не является атомарным. Действительно, в общем случае один шаг включает в себя также шаги для подвыражений. Неформально говоря, один шаг семантики малого шага включает в себя как поиск выражения, которое необходимо упростить, так и само упрощение. Это усложняет анализ и доказательство свойств языка, для которого разработана семантика. Для описания, в котором каждый шаг атомарен, были разработаны семантики в нотации Феллейсена.

1.3.3 Семантики в нотации Феллейсена

Семантики в нотации Феллейсена являются альтернативным способом описания семантики малого шага. Основные идеи были предложены Гордоном Плоткиным [77] и обобщены Матиасом Феллейсеном в его PhD диссертации [79]. Семантика в нотации Феллейсена представлена в виде набора правил редукции, каждое из которых определяет один потенциальный шаг редукции. Главной отличительной особенностью данной семантики является атомарность каждого шага. Это достигается благодаря разделению поиска подвыражения, которое необходимо упростить, на набор атомарных шагов.

$$(C : S, \sigma, n) \rightsquigarrow (S, \sigma, C[n]) \quad [\textsc{Num}_F]$$

$$(S, \sigma, x) \rightsquigarrow (S, \sigma, \sigma(x)) \quad [\textsc{Var}_F]$$

$$(S, \sigma, n_1 + n_2) \rightsquigarrow (S, \sigma, n_1 \oplus n_2) \quad [\textsc{AddV}_F]$$

$$(S, \sigma, e_1 + e_2) \rightsquigarrow (\Box + e_2 : S, \sigma, e_1) \quad [\textsc{AddL}_F]$$

$$(S, \sigma, n_1 + e_2) \rightsquigarrow (n_1 + \Box : S, \sigma, e_2) \quad [\textsc{AddR}_F]$$

Рис. 1.5 — Семантика в нотации Феллейсена для языка арифметических выражений

Для описания семантики в нотации Феллейсена необходимо ввести понятие *контекста* — выражения, содержащего “дыру”:

$$K = \Box + L \mid L + \Box.$$

Окружение семантики по-прежнему будет содержать функцию для овеществления переменных, однако оно пополнится *стеком контекстов*, содержащим отложенные выражения вида $S = \varepsilon \mid K : S$.

Контекст возникает при необходимости вычислить подвыражение исполняемой программы. В этом случае в исполняемой программе подвыражение заменяется на “дыру”, а само подвыражение занимает место исполняемого выражения.

Семантика в нотации Феллейсена (обозн. “$\rightsquigarrow$”) для языка арифметических выражений представлена на рис. 1.5 и содержит пять правил редукции. В случае, если вычисляемым выражением является натуральное число (правило $\textsc{Num}_F$), а стек контекстов непуст, извлечем головной элемент из контекста и подставим это число на место “дыры”, так как вычисляемое выражение не может быть упрощено. При вычислении переменной (правило $\textsc{Var}_F$), как и в уже рассмотренных семантиках, овеществим переменную с помощью функции σ. Контекст в этом случае остаётся неизменным. Если необходимо исполнить сложение двух натуральных чисел (правило $\textsc{AddV}_F$), то произведем сложение посредством функции “$\oplus$”, также оставив контекст без изменения. Оставшиеся два правила $\textsc{AddL}_F$ и $\textsc{AddR}_F$ определяют порядок вычисления слагаемых: если левое слагаемое не является числом, вычислим его в первую очередь, а в противном случае вычислим второе слагаемое. В обоих правилах стек пополняется контекстом, соответствующим рассматриваемому оператору сложения, в

котором заменили оно из слагаемых “дырой”. Отметим, что в данной семантике все правила являются аксиомами, поскольку они не содержат условий. Также не соответствует ни одного правила случаю, когда исполняемое выражение является натуральным числом, и стек контекстов пуст. Именно этот случай является сигналом для завершения исполнения.

Итоговая семантика определяется с помощью рефлексивно-транзитивного замыкания, так же как и семантика малого шага. Пусть p — некоторое арифметическое выражение, σ — окружение и n — некоторое натуральное число. Тогда выражение p в окружении функции σ и пустого стека контекстов ε вычислимо И принимает значение n, если верно, что $(\varepsilon, \sigma, p) \rightsquigarrow^* (\varepsilon, \sigma, n)$, где “$\rightsquigarrow^*$” — рефлексивно-транзитивное замыкание отношения “$\rightsquigarrow$”.

Данная семантика позволяет подробно описать процесс вычисления программы, т.е. задать порядок вычисления подвыражений и запомнить путь к исполняемому подвыражению посредством стека контекстов. Также стек контекстов позволяет выразить такие особенности языка, как обработка исключений (exceptions) и продолжений (continuations). Наконец, подобное представление семантики упрощает анализ свойств вычисляемой программы за счёт атомарности каждого шага исполнения.

Глава 2. Реляционное преобразование типизированных функциональных программ

Предлагаемый подход подразумевает преобразование функциональных программ в реляционные, причём предполагается, что у исходного функционального языка имеется *реляционное расширение*. Именно с помощью этого расширения и записываются преобразованные программы. Подход с реляционным расширением выглядит естественным, поскольку основной реляционный язык — miniKanren — является встраиваемым предметно-ориентированным языком: в данный момент имеются реализации miniKanren, встроенные в Scheme [40], OCaml [67], Haskell[1] и другие языки. Таким образом в реляционном языке удаётся повторно использовать много конструкций исходного языка и не определять их заново, что пришлось бы делать при создании отдельного реляционного языка. Повторно используются, например, определение функций, λ-абстракция, конструкторы и т.д. Кроме того, подход с реляционным расширением оправдан с практической точки зрения, поскольку предполагает, что программист работает в одной языковой инфраструктуре — сначала он пишет программу на обычном функциональном языке, а потом он автоматически получает реляционную программу для реляционного расширения этого же языка.

В этой главе представлено формальное описание реляционного преобразования типизированных функциональных программ. Данное преобразование состоит из следующих компонент:

- синтаксис, система правил типизации и семантика исходного функционального языка,
- синтаксис, система правил типизации и семантика реляционного расширения исходного языка,
- система правил преобразования типов и программ в реляционную форму.

В качестве исходного функционального языка было взято простейшее подмножество, включающее λ-исчисление, let-связывания и сопоставление м образцом. Все эти компоненты содержатся в большинстве реальных функциональных языков, таких как ML, Haskell, Racket, F# и др. Благодаря этому

[1] На официальном сайте языка miniKanren [80] представлены встраивания как в перечисленные языки программирования, так и во многие другие.

предлагаемый подход по преобразованию функциональных программ в реляционные не привязан к конкретному языку, но может быть легко реализован для требуемого функционального языка. Более того, несмотря на минималистичность, данный язык содержит всё необходимое для встраивания реляционного расширения. В результате мы можем переиспользовать синтаксис, систему правил типизации и семантику исходного функционального языка при описании семантики реляционного расширения.

2.1 Исходный функциональный язык

Синтаксис исходного функционального языка представлен на рис. 2.1. В первую очередь, язык включает λ-исчисление, состоящее из функциональных переменных, применения и абстракции. Данное исчисление необходимо для описания анонимных функций — основополагающего инструмента описания нерекурсивнх вычислений любого функционального языка программирования. Помимо анонимных функций язык дополнен возможностью описания именованных функций с помощью let-связывания, позволяющих ссылаться на функции по их именам и являющихся классическим способом описания рекурсивных вычислений.

Помимо этих конструкций данный язык содержит конструкторы с фиксированным количеством аргументов C^n, необходимые для конструирования значений языка. Отметим, что среди конструкторов определены два нульместных конструктора `true` и `false`, а среди функциональных переменных определен оператор равенства “=”. В совокупности данные конструкторы и оператор дополняют язык полиморфным сравнением, поведение которого сходно с реляционной унификацией. Поэтому полиморфное сравнение имеет эффективный реляционный образ.

Наконец, исходный язык включает в себя сопоставление с образцом — основополагающий оператор функционального программирования, используемый для ветвления потока исполнения программы и деконструкции значений. Отметим, что в контексте данной работы при сопоставлении с образом допустимы только примитивные образцы, состоящие из конструктора и набора переменных вида $C^n(x_1 \ldots x_n)$. Это ограничение необходимо для упрощения семантики

конструкции сопоставления с образцом. Оно не является существенным, ведь образцы общего вида могут быть выражены с помощью примитивных образцов. Также среди образцов отсутствует специальный образец “wildcard” (обозн. “_”), поскольку его наличие требует существенно иного сопоставления с образцом.

В данном языке использована система типов Хиндли-Милнера [81; 82], поддерживающая полиморфизм и являющаяся де-факто основой систем типов для реальных функциональных языков. Итоговая система типов языка представлена в виде набора правил — см. рис. 2.2. Кроме переменных типа и функциональных типов эта система содержит множество неявно определенных алгебраических типов T^k, причём каждый конструктор принадлежит только к одному типу. В правиле $\textsc{Constr}_T$ предполагается, что тип t^C имеет форму $T^k(t_1, \ldots, t_k)$, где каждый из типов t_i может быть восстановлен из t_i^C как тип соответствующего аргумента конструктора типа T^k. Помимо этого в правиле $\textsc{Match}_T$ типы всех образцов $C_i^{k_i}(x_1^i, \ldots, x_{k_i}^i)$ должны быть равны t^C, а $t_j^{C_i}$ является типом j-ого аргумента конструктора C_i, используемого в образце. Правило $\textsc{Eq}_T$ определяет, что оба аргумента оператора равенства должны иметь одинаковый, хоть и произвольный, тип. Таким образом, этот оператор является оператором “полиморфного равенства”.

$$
\begin{array}{rcl}
\mathcal{E} & = & x \\
& | & \lambda x.e \\
& | & e_1\ e_2 \\
& | & C^n(e_1, \ldots, e_n) \\
& | & \underline{\texttt{true}} \\
& | & \underline{\texttt{false}} \\
& | & \underline{\texttt{let}}\ x\ \texttt{=}\ e_1\ \underline{\texttt{in}}\ e_2 \\
& | & \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ \texttt{=}\ \lambda x.e_1\ \underline{\texttt{in}}\ e_2 \\
& | & e_1 = e_2 \\
& | & \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \to e_i\} \\
\\
\mathcal{P} & = & C^n(x_1, \ldots, x_n)
\end{array}
$$

Рис. 2.1 — Синтаксис входного языка

Семантика исходного языка (рис. 2.3) представлена в виде системы переходов над множеством *состояний*, состоящих из исполнения выражения e со стеком контекстов $\mathcal{S}$. Отношение перехода определяет один шаг:

$$\langle \mathcal{S}, e \rangle \to \langle \mathcal{S}', e' \rangle.$$

Результатом этого шага является новый стек контекстов $\mathcal{S}'$ и новое выражение e'. Контекст является выражением, содержащим уникальный символ, называемый “дырой” (обозн. $\square$); неформально, стек контекстов можно определить как путь вычисления выражения от внешнего уровня до позиции, где в текущий

Типы:

$$\begin{array}{lcll}
\mathcal{X} & = & \alpha, \beta, \ldots & \text{(переменные типа)} \\
\mathcal{D} & = & \texttt{bool}, T^n, \ldots & \text{(типы конструкторов)} \\
\mathcal{T} & = & \alpha \mid T^k(t_1, \ldots, t_k) \mid t_1 \to t_2 & \text{(типы)} \\
\mathcal{S} & = & \forall \bar{\alpha}.t & \text{(замкнутые типы)}
\end{array}$$

Правила типизации:

$$\Gamma \vdash \underline{\texttt{true}}, \underline{\texttt{false}} : \texttt{bool} \quad [\textsc{Bool}_T] \qquad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \texttt{bool}} \quad [\textsc{Eq}_T]$$

$$\frac{\Gamma \vdash e_i : t_i^C}{\Gamma \vdash C^n(e_1, \ldots, e_n) : t^C} \quad [\textsc{Constr}_T] \qquad \Gamma, x : \forall \bar{\alpha}.t \vdash x : t[\bar{\alpha} \leftarrow \bar{t'}] \quad [\textsc{Var}_T]$$

$$\frac{\Gamma \vdash f : t_1 \to t_2 \quad \Gamma \vdash e : t_1}{\Gamma \vdash f\ e : t_2} \quad [\textsc{App}_T] \qquad \frac{\Gamma, x : t_1 \vdash f : t_2}{\Gamma \vdash \lambda x.f : t_1 \to t_2} \quad [\textsc{Abs}_T]$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : \forall \bar{\alpha}.t_1 \vdash e_2 : t}{\Gamma \vdash \underline{\texttt{let}}\ x\ = e_1\ \underline{\texttt{in}}\ e_2 : t}, \ \bar{\alpha} = FV(t_1) \setminus FV(\Gamma) \quad [\textsc{Let}_T]$$

$$\frac{\Gamma, f : t_1 \vdash \lambda x.e_1 : t_1 \quad \Gamma, f : \forall \bar{\alpha}.t_1 \vdash e_2 : t}{\Gamma \vdash \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ = \lambda x.e_1\ \underline{\texttt{in}}\ e_2 : t}, \ \bar{\alpha} = FV(t_1) \setminus FV(\Gamma) \quad [\textsc{LetRec}_T]$$

$$\frac{\Gamma \vdash e : t^C \quad \Gamma, x_1^i : t_1^{C_i}, \ldots, x_{k_i}^i : t_{k_i}^{C_i} \vdash e_i : t}{\Gamma \vdash \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{C_i^{k_i}(x_1^i \ldots x_{k_i}^i)\ \to e_i\} : t} \quad [\textsc{Match}_T]$$

Рис. 2.2 — Правила типизации для исходного языка

момент происходит вычисление. По контексту C и выражению e может быть построено полное выражение $C[e]$ методом подстановки выражения e на место уникальной “дыры” в контексте C. Для любого состояния $\langle C_1 : C_2 : \ldots : C_k, e \rangle$ может быть построено выражение $C_k[\ldots[C_2[C_1[e]]]\ldots]$, которое является промежуточным результатом вычисления в соответствии с семантикой малого шага. Подобная форма описания семантики, как было сказано в главе 1, называется нотацией Матиаса Феллейсена [64] для семантик малого шага. Она была выбрана из-за того, что имеется возможность распространить её на случай реляционного расширения.

Значение:

$$\mathcal{V} = C^n(v_1, \ldots, v_n) \mid \lambda x.e \mid \mu f \lambda x.e \mid \underline{\texttt{true}} \mid \underline{\texttt{false}}$$

Контекст:

$$\begin{array}{lcl} \mathcal{C} & = & \Box\, e \mid v\, \Box \mid C^n(\bar{v},\Box,\bar{e}) \mid \Box\texttt{=e} \mid \texttt{v=}\Box \\ & \mid & \underline{\texttt{let}}\ x\ \texttt{=}\ \Box\ \underline{\texttt{in}}\ e \mid \underline{\texttt{match}}\ \Box\ \underline{\texttt{with}}\ \{p_i \to e_i\} \end{array}$$

Стек контекстов:

$$\mathcal{S} = \varepsilon \mid \mathcal{C} : \mathcal{S}$$

Состояние:

$\langle \mathcal{S}, e \rangle$ (стек контекстов, выражение); $\langle \varepsilon, e \rangle$ (начальное состояние); $\langle \varepsilon, v \rangle$ (финальное состояние)

Правила переходов:

$$\langle C : \mathcal{S}, v \rangle \to \langle \mathcal{S}, C[v] \rangle \qquad [\textsc{Value}]$$

$$\langle \mathcal{S}, f\ e \rangle \to \langle \Box\, e : \mathcal{S}, f \rangle \quad [\textsc{AppL}] \qquad \langle \mathcal{S}, v\ e_2 \rangle \to \langle v\, \Box : \mathcal{S}, e_2 \rangle \quad [\textsc{AppR}]$$

$$\langle \mathcal{S}, e_1 = e_2 \rangle \to \langle \Box = e_2 : \mathcal{S}, e_1 \rangle \quad [\textsc{EqL}] \qquad \langle \mathcal{S}, v = e \rangle \to \langle v = \Box : \mathcal{S}, e \rangle \quad [\textsc{EqR}]$$

$$\langle \mathcal{S}, v = v \rangle \to \langle \mathcal{S}, \underline{\texttt{true}} \rangle \qquad [\textsc{EqTrue}]$$

$$\langle \mathcal{S}, v_1 = v_2 \rangle \to \langle \mathcal{S}, \underline{\texttt{false}} \rangle,\ v_1 \neq v_2 \qquad [\textsc{EqFalse}]$$

$$\langle \mathcal{S}, (\lambda x.e)\ v \rangle \to \langle \mathcal{S}, e[x \leftarrow v] \rangle \qquad [\textsc{Beta}]$$

$$\langle \mathcal{S}, (\mu f \lambda x.e)\ v \rangle \to \langle \mathcal{S}, e[f \leftarrow \mu f \lambda x.e, x \leftarrow v] \rangle \qquad [\textsc{Mu}]$$

$$\langle \mathcal{S}, C^n(v_1, \ldots, v_{k-1}, e_k, \ldots, e_n) \rangle \to \langle C^n(v_1, \ldots, v_{k-1}, \Box, \ldots, e_n) : \mathcal{S}, e_k \rangle \qquad [\textsc{Constr}]$$

$$\langle \mathcal{S}, \underline{\texttt{let}}\ x\ \texttt{=}\ e_1\ \underline{\texttt{in}}\ e_2 \rangle \to \langle \underline{\texttt{let}}\ x\ \texttt{=}\ \Box\ \underline{\texttt{in}}\ e_2 : \mathcal{S}, e_1 \rangle \qquad [\textsc{Let}]$$

$$\langle \mathcal{S}, \underline{\texttt{let}}\ x\ \texttt{=}\ v\ \underline{\texttt{in}}\ e \rangle \to \langle \mathcal{S}, e[x \leftarrow v] \rangle \qquad [\textsc{LetVal}]$$

$$\langle \mathcal{S}, \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ \texttt{=}\ \lambda x.e_1\ \underline{\texttt{in}}\ e_2 \rangle \to \langle \mathcal{S}, e_2[f \leftarrow \mu f \lambda x.e_1] \rangle \qquad [\textsc{LetRec}]$$

$$\langle \mathcal{S}, \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \to e_i\} \rangle \to \langle \underline{\texttt{match}}\ \Box\ \underline{\texttt{with}}\ \{p_i \to e_i\} : \mathcal{S}, e \rangle \qquad [\textsc{Match}]$$

$$\left\langle \mathcal{S}, \underline{\texttt{match}}\ C_k^{n_k}(v_1 \ldots v_{n_k})\ \underline{\texttt{with}}\ \{C_i^{n_i}(x_1^i \ldots x_{n_i}^i) \to e_i\} \right\rangle \to \left\langle \mathcal{S}, e_k[x_j^k \leftarrow v_j] \right\rangle \qquad [\textsc{MatchVal}]$$

Рис. 2.3 — Семантика исходного языка

Данная семантика описывает вычисление аргументов функций слева-направо со стратегией вычисления по значению. Правила Beta, Mu, LetVal,

LetRec and MatchVal отвечают за подстановку в соответствии с именами переменных в абстракциях и let-связываниях. В правиле MatchVal предполагается, что ровно один образец соответствует сопоставляемому выражению — это важное отличие от классической семантики сопоставления с образцом, когда образцы проверяются последовательно сверху-вниз до первого успешного сопоставления. Правила EqTrue и EqFalse предполагают, что значения v, v_1, v_2 не имеют форму $\lambda x \ldots$ или $\mu f \ldots$.

Наконец, для замкнутого выражения e и значения v определено $e \rightsquigarrow^f v$ тогда и только тогда, когда выполнено следующее:

$$\langle \varepsilon, e \rangle \rightarrow^* \langle \varepsilon, v \rangle ,$$

где ε — это пустой стек контекстов а "$\rightarrow^*$" является рефлексивно-транзитивным замыканием для "$\rightarrow$".

2.2 Реляционное расширение

Реляционное расширение добавляет в исходный функциональный язык пять классических операторов языка miniKanren для описания реляционных целей: унификацию, ограничение неравенством, дизъюнкцию, конъюнкцию и оператор введения свежей переменной. Расширенный синтаксис реляционного расширения представлен на рис. 2.4.

Отметим, что добавление реляционных операторов позволяет конструировать некорректные выражения, например `λ x.(x ∧ λ y.y)`. Для устранения подобных случаев было разработано специальное расширение системы типизации исходного языка. Данный подход соответствует актуальной реализации OCanren — реляционного расширения языка OCaml [83]. Аккуратный выбор типов для представления выражений и целей в OCaml реализации позволяет отклонять большинство некорректных программ на этапе компиляции.

$$\begin{array}{rcl} \mathcal{E} & += & \underline{\texttt{fresh}}\ (x)\ \ e \\ & | & e_1 \equiv e_2 \\ & | & e_1 \not\equiv e_2 \\ & | & e_1 \vee e_2 \\ & | & e_1 \wedge e_2 \end{array}$$

Рис. 2.4 — Синтаксис реляционного расширения

Типы:

$$\mathcal{L} \;=\; \alpha^o \mid (T^n(l_1,\ldots,l_n))^o \quad \text{(логические типы)}$$
$$\mathcal{T} \;+\!=\; \mathfrak{G}$$

Правила типизации:

$$\frac{\Gamma, x : l \vdash e : \mathfrak{G}}{\Gamma \vdash \underline{\texttt{fresh}}\ (x)\ e : \mathfrak{G}} \; [\textsc{Fresh}_T]$$

$$\frac{\Gamma \vdash e_1 : l \quad \Gamma \vdash e_2 : l}{\Gamma \vdash e_1 \equiv e_2 : \mathfrak{G}} \; [\textsc{Unify}_T] \qquad \frac{\Gamma \vdash e_1 : l \quad \Gamma \vdash e_2 : l}{\Gamma \vdash e_1 \not\equiv e_2 : \mathfrak{G}} \; [\textsc{Disequality}_T]$$

$$\frac{\Gamma \vdash e_1 : \mathfrak{G} \quad \Gamma \vdash e_2 : \mathfrak{G}}{\Gamma \vdash e_1 \wedge e_2 : \mathfrak{G}} \; [\textsc{Conjunction}_T] \qquad \frac{\Gamma \vdash e_1 : \mathfrak{G} \quad \Gamma \vdash e_2 : \mathfrak{G}}{\Gamma \vdash e_1 \vee e_2 : \mathfrak{G}} \; [\textsc{Disjunction}_T]$$

Рис. 2.5 — Правила типизации для реляционного расширения

Для расширения системы типизации был введён полиморфный конструктор типа $\square^o$ с соответствующим ему конструктором логических термов $\uparrow$. Для выражения реляционных целей был введён специальный уникальный тип $\mathfrak{G}$. Правила типизации для реляционного расширения представлены на рис. 2.5.

Данные правила накладывают следующие ограничения: в случае унификации и ограничения неравенством допустимы только термы одинакового логического типа; конъюнкция и дизъюнкция применима только к реляционным целям. Отметим, что в рамках предложенного расширения терм может быть вычислен как результат выполнения произвольного выражения на исходном функциональном языке (при условии, что это выражение имеет ожидаемый логический тип). Однако такие термы, имея “более высокий порядок”, не могут быть получены в результате реляционного преобразования. Следовательно, данное реляционное расширение определяет более богатый язык, чем требуется для реляционного преобразования.

Семантика расширенного языка представлена на рис. 2.6. Прежде всего расширено состояние исходной семантики. Помимо стека контекстов и текущего выражения оно содержит множество *семантических переменных* Σ и *логическое состояние* σ. Семантические переменные выделяются и подставляются вместо логических переменных, когда исполняется выражение <u>`fresh`</u> в правиле переходов Fresh. Логическое состояние обновляется, когда исполняется унифи-

Семантические переменные:

$$\mathfrak{S} = \mathfrak{s}_1, \mathfrak{s}_2, \dots$$

$$\Sigma, \Sigma' \dots \subset 2^{\mathcal{S}} \text{ (множества выделенных семантических переменных)}$$

$$\langle \Sigma', \mathfrak{s} \rangle \leftarrow \underline{\texttt{new}}\ \Sigma,\ \Sigma' = \Sigma \cup \{\mathfrak{s}\},\ \mathfrak{s} \notin \Sigma \text{ (выделение новой семантической переменной)}$$

Значения:

$$\mathcal{V} \mathrel{+}= \underline{\texttt{success}} \mid \mathfrak{s}$$

Контекст:

$$\mathcal{C} \mathrel{+}= \Box \equiv e \mid v \equiv \Box \mid \Box \not\equiv e \mid v \not\equiv \Box \mid \Box \wedge e \mid e \wedge \Box$$

Состояние:

$$\langle \Sigma, \mathcal{S}, e, \sigma \rangle \text{ (множество семантических переменных, стек контекстов, выражение, логическое состояние)}$$

$$\langle \varnothing, \varepsilon, e, \iota \rangle \text{ (начальное состояние)}$$

Правила переходов:

$$\langle \Sigma, \mathcal{S}, \underline{\texttt{fresh}}(x)\ e, \sigma \rangle \rightsquigarrow \langle \Sigma', \mathcal{S}, e[x \leftarrow \mathfrak{s}], \sigma \rangle,\ \langle \Sigma', \mathfrak{s} \rangle \leftarrow \underline{\texttt{new}}\ \Sigma \qquad [\textsc{Fresh}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \equiv e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \Box \equiv e_2 : \mathcal{S}, e_1, \sigma \rangle \qquad [\textsc{UnifyL}]$$

$$\langle \Sigma, \mathcal{S}, v \equiv e, \sigma \rangle \rightsquigarrow \langle \Sigma, v \equiv \Box : \mathcal{S}, e, \sigma \rangle \qquad [\textsc{UnifyR}]$$

$$\langle \Sigma, \mathcal{S}, v_1 \equiv v_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, \underline{\texttt{success}}, \sigma' \rangle,\ \mathbf{unify}\,(\sigma, v_1, v_2) = \sigma' \qquad [\textsc{Unify}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \not\equiv e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \Box \not\equiv e_2 : \mathcal{S}, e_1, \sigma \rangle \qquad [\textsc{DisEqL}]$$

$$\langle \Sigma, \mathcal{S}, v \not\equiv e, \sigma \rangle \rightsquigarrow \langle \Sigma, v \not\equiv \Box : \mathcal{S}, e, \sigma \rangle \qquad [\textsc{DisEqR}]$$

$$\langle \Sigma, \mathcal{S}, v_1 \not\equiv v_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, \underline{\texttt{success}}, \sigma' \rangle,\ \mathbf{diseq}\,(\sigma, v_1, v_2) = \sigma' \qquad [\textsc{DisEq}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \vee e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, e_1, \sigma \rangle \qquad [\textsc{DisjL}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \vee e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, e_2, \sigma \rangle \qquad [\textsc{DisjR}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \wedge e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \Box \wedge e_2 : \mathcal{S}, e_1, \sigma \rangle \qquad [\textsc{ConjStartL}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \wedge e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, e_1 \wedge \Box : \mathcal{S}, e_2, \sigma \rangle \qquad [\textsc{ConjStartR}]$$

$$\langle \Sigma, \mathcal{S}, \underline{\texttt{success}} \wedge e, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, e, \sigma \rangle \qquad [\textsc{ConjL}]$$

$$\langle \Sigma, \mathcal{S}, e \wedge \underline{\texttt{success}}, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, e, \sigma \rangle \qquad [\textsc{ConjR}]$$

Рис. 2.6 — Семантика реляционного расширения

кация или ограничение неравенством. Все описанные ранее правила переходов для исходного языка сохраняются, однако требуют дополнить состояние множеством семантических переменных и логическим состоянием. Эти добавленные

компоненты состояния остаются без изменений при исполнении правил для исходного языка.

Множество значений дополнено двумя новыми типами значений: семантическими переменными и специальным значением `success`. Семантические переменные являются результатом исполнения свободных логических переменных, значение `success` является результатом исполнения успешной цели.

Определение контекста также расширено четырьмя новыми типами выражений с “дырой”. Новые контексты определяют детерминированное исполнение унификации и ограничения неравенством слева-направо. Исполнение дизъюнкции и конъюнкции недетерминировано. При исполнении дизъюнкции производится выбор только одной подцели (правила DisjL и DisjR), при исполнении конъюнкции недетерминировано определяется, какая из подцелей будет вычислена первой (правила ConjStartL и ConjStartR). После вычисления выбранной подцели до значения `success`, начнется вычисление второй подцели (правила ConjL и ConjR).

Существующие реализации miniKanren используют различные методы поиска. В данной работе был выбран недетерминированный вариант семантики с тем, чтобы устранить зависимость от деталей конкретной реализации. Обратной стороной этого решения является то, что для конкретной программы и конкретной реализации miniKanren результат вычисления может не совпадать с предписываемым семантикой. Например, в конкретной реализации программа может расходиться, в то время как недетерминированная семантика будет попрежнему завершаться. Однако в этом случае программа или интерпретатор miniKanren могут быть переписаны так, чтобы они сходились в соответствии с этим сценарием.

Теперь рассмотрим структуру логического состояния и семантику унификации и ограничения неравенством. Данная семантика во многом соответствует реализации языка miniKanren, дополненного ограничением неравенством, представленного в [70], а также стандартным подходам к реализации унификации [71; 84]. Далее будут использованы следующие стандартные понятия:

- подстановка (θ), сопоставляющая логические термы семантическим переменным;
- применение подстановки θ к терму с ($t\,\theta$) для замены семантических переменных в терме t в соответствии с подстановкой θ;

– композиция подстановок $(\theta\theta')$, применение которой к терму t соответствует последовательному применению подстановок θ и θ';
– наиболее общий унификатор двух термов $(mgu\,(t_1, t_2))$, определяющий наиболее общую подстановку θ, при которой термы $(t_1\,\theta)$ и $(t_2\,\theta)$ синтаксически равны.

Логическое состояние семантики состоит из двух следующих компонентов:

$$\sigma = (\theta,\Theta^-),$$

где θ является подстановкой, а Θ^- — это набор негативных подстановок, описывающих ограничения неравенством, которые потенциально могут быть нарушены. Исходное состояние семантики содержит неопределённую подстановку и пустое множество:

$$\iota = (\bot,\varnothing).$$

Операция унификации принимает на вход два терма и логическое состояние в качестве аргументов:

$$\mathbf{unify}\,(\sigma, t_1, t_2) = unify\,((\theta,\Theta^-), t_1, t_2).$$

Выполнение унификации осуществляется за несколько шагов. Прежде всего, необходимо вычислить наиболее общий унификатор для термов t_1 и t_2 с учетом текущей подстановки θ:

$$\rho = mgu\,(t_1\,\theta, t_2\,\theta).$$

Если подобного унификатора ρ не существует, унификация завершается провалом. Отметим, что для этого случая в семантике реляционного расширения правил нет, поэтому вычисление реляционной программы также завершается без получения ответа. В случае успешного завершения унификации подстановку ρ необходимо проверить на согласованность с ограничениями неравенством, представленными с помощью Θ^- (если набор Θ^- пуст, проверка немедленно завершается успехом).

Рассмотрим пару термов

$$\begin{array}{rcl} t_l & = & (\mathfrak{s}_1, \dots, \mathfrak{s}_k), \\ t_r & = & (\rho(\mathfrak{s}_1), \dots, \rho(\mathfrak{s}_k)), \end{array}$$

где $\{\mathfrak{s}_i\} = dom\,(\rho)$. Для каждой подстановки $\theta^- \in \Theta^-$ вычислим $mgu\,(t_l\,\theta^-,t_r\,\theta^-)$. При этом возможны следующие варианты.

1. Унификация завершилась провалом. Из этого следует, что ограничение неравенством, представленное с помощью θ^-, больше не может быть нарушено. В данном случае удалим θ^- из Θ^- и продолжим проверку со следующим ограничением неравенством.
2. Унификация завершилось успехом с пустой подстановкой в качестве результата. Это означает, что ограничение неравенством, представленное с помощью θ^-, нарушено. Следовательно, проверка завершается и исходная операция унификации завершается провалом.
3. Унификация завершилась успехом с непустой подстановкой θ'^-. В данном случае, чтобы не нарушить представленное θ^- ограничение неравенством, необходимо при будущих унификациях проверять согласованность с θ'^-. Заменим θ^- на θ'^- в Θ^- и продолжим проверку со следующим ограничением неравенством.

Результатом успешной проверки неравенств является модифицированный набор Θ'^-, что является вторым и последним компонентом итогового логического состояния:

$$\mathbf{unify}\,((\theta,\Theta^-),t_1,t_2) = (\theta\rho,\Theta'^-).$$

Операция ограничения неравенством исполняется схожим образом и также принимает пару термов и логическое состояние в качестве аргументов:

$$\mathbf{diseq}\,(\sigma,t_1,t_2) = \mathbf{diseq}\,((\theta,\Theta^-),t_1,t_2).$$

Как и в случае с унификацией, в первую очередь вычислим $mgu\,(t_1\,\theta,t_2\,\theta)$ и рассмотрим три следующих случая.

1. Унификация завершилась с отрицательным результатом, и это означает, что ограничение неравенством уже выполнено в текущей подстановке.
2. Унификация успешно завершилась с пустой подстановкой в качестве результата. Следовательно, ограничение неравенством нарушено.

3. Унификация успешно завершилась с непустой подстановкой θ'^{-}. Это означает, что данная подстановка описывает ограничение неравенством, которое должно выполняться в будущем, поэтому добавим её в Θ^{-}.

Результатом успешного вычисления ограничения неравенством является (потенциально) модифицированный набор Θ'^{-} поэтому необходимо обновить логическое состояние:

$$\textbf{diseq}\left((\theta,\Theta^{-}), t_1, t_2\right) = (\theta,\Theta'^{-}).$$

Наконец, для замкнутой цели g и логического состояния σ определим $g \rightsquigarrow^{r} \sigma$, если $\langle\varnothing,\varepsilon,g,\iota\rangle \rightsquigarrow^{*} \langle\Sigma,\varepsilon,\underline{\texttt{success}},\sigma\rangle$ для некоторого логического состояния Σ, где "$\rightsquigarrow^{*}$" — это рефлексивно-транзитивное замыкание отношения"$\rightsquigarrow$".

Можно заметить, что правила типизации реляционного расширения добавляют некоторые интерпретируемые типы и символы по отношению к системе типов исходного языка. Таким образом, можно ожидать, что реляционное расширение наследует все свои полезные свойства (такие как **завершаемость** (progress) и **сохранение типа** (type preservation) [85]). Однако это не верно. Действительно, единственным значением для цели является $\underline{\texttt{success}}$, но, очевидно, не каждая цель завершается успехом (к примеру, `A` $\equiv$ `B` всегда завершается провалом). Таким образом реляционное расширение лишено свойства завершаемости — существует не являющаяся значением корректно типизируемая цель, для которой не выполним ни один шаг семантики. Однако данное обстоятельство не препятствует данной работе,поскольку в любом случае значение ошибки для целей может быть добавлено к языку вместе с правилами распространения ошибки.

2.3 Реляционное преобразование

Как правило, функциональные программы работают со значениями высокого порядка, в то время как miniKanren ограничен унификацией первого порядка. Поэтому не каждую функциональную программу можно преобразовать в реляционную с помощью простых преобразований. Сформулируем

несколько ограничений для исходных программ перед тем, как ввести реляционное преобразование.

Ввведем множество **заземлённых типов** (ground types) $\mathcal{G}$ следующим образом:

$$\mathcal{G} = \alpha \mid T^k(g_1, \ldots, g_k).$$

Причем все выражения заземленного типа должны быть дополнены конструктором логических выражений $\uparrow$, что на уровне типов может быть описано с помощью преобразования $[\![\bullet]\!]^o$:

$$\begin{array}{rcl} [\![\alpha]\!]^o & = & \alpha, \\ [\![T^k(g_1, \ldots, g_k)]\!]^o & = & (T^k([\![g_1]\!]^o, \ldots, [\![g_k]\!]^o))^o. \end{array}$$

Неформально говоря, значение заземленного типа не может содержать подвыражений высшего порядка. Поэтому введем следующие три ограничения на программы, предназначенные для преобразования в реляционную форму.

1. Параметры конструкторов типа должны быть переменными типа.
2. Полиморфное равенство и конструкторы могут быть применены только к выражениям заземлённого типа.
3. Любое выражение сопоставления с образцом должно быть заземлённого типа.

Первое ограничение требует, чтобы все алгебраические типы программы были полностью полиморфны. Благодаря этому второе ограничение позволяет ограничить полиморфизм для реляционных программ: все переменные типа, содержащиеся в конструкторах типа, могут быть заменены только на заземлённые типы. Отметим, что это условие является достаточным, но не необходимым.

Третье ограничение введено для упрощения представления реляционного преобразования. Если сопоставление с образцом имеет незаземлённый тип, оно всё же может быть преобразовано в эквивалентное выражение, содержащее только сопоставление с образцом заземлённого типа, с помощью η-расширения:

$$\underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i\ \rightarrow\ e_i\} \rightsquigarrow \lambda\ \bar{x}.\underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i\ \rightarrow\ e_i\,\bar{x}\},$$

где $\bar{x}$ является вектором новых переменных, не содержащихся в выражениях e, e_i, и p_i. Причем реализация предлагаемого реляционного преобразования, описанная в главе 6, выполняет η-расширение для любого сопоставления с образцом незаземлённого типа. Отметим, что это единственный случай

использования типа исходной программы и выполнения η-расширения при реляционном преобразовании.

Основную идею реляционного преобразования можно описать на уровне типов: выражение исходного языка с типом t преобразуется в выражение реляционного расширения с типом $[\![t]\!]^t$, где преобразование типа $[\![\bullet]\!]^t$ определено следующим образом:

$$
\begin{array}{rcl}
[\![g]\!]^t & = & [\![g]\!]^o \to \mathfrak{G}, \\
[\![t_1 \to t_2]\!]^t & = & [\![t_1]\!]^t \to [\![t_2]\!]^t.
\end{array}
$$

Иначе говоря, выражение заземлённого типа преобразуется в одноместную функцию, принимающую выражение соответствующего логического типа и возвращающую цель. Данная функция сопоставляет логическую форму исходного значения с переданным аргументом. Поскольку аргумент может содержать несколько вхождений свободных переменных, данная функция пытается сопоставить этим переменных соответствующие подвыражения исходного выражения. Например, конструктор-константа `Nil` будет преобразован в функцию $\lambda\ q\ .\ q \equiv \uparrow\texttt{Nil}$.

Теперь рассмотрим реляционное преобразование для произвольного выражения t, удовлетворяющего описанным выше ограничениям. Обозначим это преобразование как $[\![t]\!]^c$.

$$
\begin{array}{rcl}
[\![x]\!]^c & = & x \\
[\![\lambda\ x\ .\ e]\!]^c & = & \lambda\ x\ .\ [\![e]\!]^c \\
[\![f\ e]\!]^c & = & [\![f]\!]^c\ [\![e]\!]^c \\
[\![\underline{\texttt{let}}\ x\ \texttt{=}\ e_1\ \underline{\texttt{in}}\ e_2]\!]^c & = & \underline{\texttt{let}}\ x\ \texttt{=}\ [\![e_1]\!]^c\ \underline{\texttt{in}}\ [\![e_2]\!]^c \\
[\![\underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ \texttt{=}\ \lambda x.e_1\ \underline{\texttt{in}}\ e_2]\!]^c & = & \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ \texttt{=}\ [\![\lambda x.e_1]\!]^c\ \underline{\texttt{in}}\ [\![e_2]\!]^c
\end{array}
$$

Пять первых правил полностью сохраняют структуру исходного выражения и применяют преобразования ко всем вложенным выражениям. Содержательная часть преобразования заключается в преобразовании различных применений конструкторов, сопоставления с образцом и полиморфного оператора равенства. Отметим, что оператор свежей переменной <u>`fresh`</u> в последующих правилах вводит сразу набор логических переменных. Данный синтаксический сахар введен для упрощения восприятия правил и аналогичен набору операторов <u>`fresh`</u>, вводящих по одной переменной.

$$
[\![C^k(e_1,\ldots,e_k)]\!]^c \;=\; \begin{array}{l}
\lambda\ q\ .\ \underline{\texttt{fresh}}\ (q_1\ldots q_k)\\
\qquad ([\![e_1]\!]^c\ q_1)\ \wedge\\
\qquad \ldots\\
\qquad ([\![e_k]\!]^c\ q_k)\ \wedge\\
\qquad (q \equiv\ \uparrow C^n(q_1,\ldots,q_k))
\end{array}
$$

При преобразовании применения конструктора известно, что каждое выражение e_i имеет заземлённый тип. Поэтому соответствующие им реляционные образы являются одноместными функциями, возвращающими цель в качестве результата. Для каждого выражения e_i создадим свежую логическую переменную с помощью оператора <u>`fresh`</u> и применим функции $[\![e_i]\!]$ к этим переменным для связывания результатов вычисления с соответствующими переменными. Результатом преобразования применения конструктора также является одноместная функция, которая возвращает цель. Поэтому дополним выражение абстракцией по переменной q и унифицируем эту переменную с конструктором, примененным к соответствующим логическим переменным. Применим также логический конструктор $\uparrow$ для приведения к соответствию правилу типизации унификации.

$$
\left[\!\!\left[\begin{array}{l}\underline{\texttt{match}}\ e\ \underline{\texttt{with}}\\ \quad \{C_i^{n_i}(x_1^i,\ldots,x_{n_i}^i) \to e_i\}\end{array}\right]\!\!\right]^c \;=\; \begin{array}{l}
\lambda\ q\ .\ \underline{\texttt{fresh}}\ (q_e)\\
\quad ([\![e]\!]^c\ q_e)\ \wedge\\
\quad \bigvee_i\ ((\underline{\texttt{fresh}}\ (q_1^i\ldots q_{n_i}^i)\\
\qquad\quad (q_e \equiv\ \uparrow C_i^{n_i}(q_1^i,\ldots,q_{n_i}^i))\ \wedge\\
\qquad\quad (\lambda\ x_1^i\ldots x_{n_i}^i\,.\,[\![e_i]\!]^c)\\
\qquad\qquad (\equiv q_1^i)\ \ldots\ (\equiv q_{n_i}^i)\ q\\
\qquad))
\end{array}
$$

Правило преобразования сопоставления с образом организовано сходным образом. Сопоставляемое выражение имеет заземлённый тип, так как оно сопоставляется с конструкторами. Выделим свежую переменную q_e и свяжем её со значением сопоставляемого выражения как и в случае применения конструктора. Затем для всех ветвей создадим по набору логических переменных (в каждой ветви для каждой переменной образца) и выразим сопоставление с образцом при помощи унификации с использованием соответствующего конструктора и созданных логических переменных. Для завершающего шага преобразования заметим, что e_i является выражением со свободными переменными, соответствующими тем, что содержатся в исходном образце. Поэтому

преобразуем e_i, абстрагируем результат абстракциями по этим переменным и получим функцию, заменяющую данные переменные переданными значениями. Остается связать переменные q^i_j с переменными образца в преобразованном выражении $[\![e_i]\!]^c$. Сделаем это посредством применения полученной функции к одноместным функциям $(\equiv q^i_j)$. В итоге снова получим одноместную функцию, которую применим к внешней результирующей переменной q.

$$
[\![e_1 = e_2]\!]^c \quad = \quad \begin{array}{l}
\lambda\ q\ .\ \underline{\texttt{fresh}}\ (q_1\, q_2) \\
\qquad [\![e_1]\!]^c\, q_1 \ \wedge \\
\qquad [\![e_2]\!]^c\, q_2 \ \wedge \\
\qquad ((q_1 \equiv q_2 \ \wedge\ q \equiv\ \uparrow\underline{\texttt{true}}) \ \vee \\
\qquad\ \ (q_1 \not\equiv q_2 \ \wedge\ q \equiv\ \uparrow\underline{\texttt{false}}) \\
\qquad)
\end{array}
$$

Заключительное правило, которое предназначено для преобразования полиморфного оператора равенства, повторяет тот же шаблон: оба аргумента имеют заземлённый тип, поэтому преобразуем их в одноместные функции, возвращающие цель в качестве результата. Применим эти функции к свежим переменным q_1 и q_2 и рассмотрим два случая. В случае равенства этих переменных сопоставим результирующей переменной q конструктор <u>`true`</u>. В противном случае сопоставим конструктор <u>`false`</u>. Заметим, что это единственный случай использования ограничения неравенством в реляционном преобразовании.

В завершение обсудим несколько примечательных свойств реляционного преобразования, представленного в данной главе. Первым свойством является полное сохранение выражений, состоящих только из переменных, абстракции, применения и <u>`let`</u>-связок. Поэтому многие полезные функции высшего порядка (например, применение, композиция, оператор неподвижной точки) уже являются реляционными и могут быть использованы в реляционных программах без изменения.

Второе свойство является композициональностью [86]: реляционный образ применения является применением реляционных образов. Именно благодаря композициональности оператор применения остается без изменения при реляционном преобразовании. В результате последнее поддерживает раздельное применение — набор исходных программ может быть разделен на несколько частей и преобразован независимо с сохранением возможности совместного

исполнения. На практике это соответствует преобразованию программы, разделенной на несколько файлов.

Наконец примечательным является тот факт, что результат реляционного преобразования исполняется детерминировано в прямом направлении, соответствующем исполнению исходной программы. Поэтому реляционное преобразование приводит к не более чем линейному замедлению при исполнении в прямом направлении. Другими словами, запрос, требующий вычислить последний аргумент отношения при заземленности остальных аргументов в точности соответствует вычислению исходной функции на тех же аргументах. Реляционный запрос с незаземленными аргументами в общем случае будет вычисляться недетерминировано, поэтому время его вычисления зависит экспоненциально от размера аргументов. При этом, сравнение производительности с соответствующей исходной функцией не представляется возможным, поскольку запрос с незаземленными аргументами не может быть сопоставлен ни с каким вызовом исходной функции.

Глава 3. Статическая и динамическая корректность реляционного преобразования

При преобразовании программ важным является свойство сохранения их семантики, называемое корректностью преобразования. Данное свойство гарантирует, что программы до и после преобразования будут одинаково исполняться, т.е. на одних и тех же входных значениях выдавать одинаковые выходные.

В предыдущей главе мы определили две семантики как для исходного языка, так и для реляционного расширения: семантику вывода типов и семантику исполнения. Традиционно, семантику вывода типов называют статической, так как результат её исполнения не зависит от конкретных аргументов. В свою очередь, семантику исполнения называют динамической.

В данной главе представлены доказательства корректности реляционного преобразования для обеих семантик: статической и динамической. Мы рассмотрим доказательство статической корректности реляционного преобразования (сохранения типизации) и доказательство частичной динамической корректности (сохранения семантики реляционного преобразования). Отметим, что динамическая корректность является частичной, так как существуют запросы для реляционного образа, которые не могут быть воспроизведены в исходной программе.

3.1 Статическая корректность реляционного преобразования

Статическая корректность — важное свойство как с теоретической, так и с практической точки зрения. С теоретической точки зрения данное свойство является необходимым требованием для подтверждения динамической корректности, так как рассуждать о сохранении семантики в типизированном языке без гарантии корректности типизации не представляется возможным. С практической точки зрения, статическая корректность необходима для гарантии успешного прохождения проверки типов при компиляции реляционного образа. В этом разделе представлено доказательство теоремы о корректности

типизации преобразованной программы при условии корректности типизации исходной функциональной программы.

Теорема 1 (о статической корректности)**.** Пусть в исходном языке выражение e имеет тип t. Тогда в реляционном расширении, после выполнения реляционного преобразования, образ этого выражения $[\![e]\!]^c$ будет иметь тип $[\![t]\!]^t$.

Прежде чем приступить к доказательству теоремы, необходимо рассмотреть, как соотносятся типы переменных, содержащихся в контекстах семантики исходного языка и семантики реляционного расширения. Для этого сформулируем определение *реляционного образа контекста* семантики исходного языка, определяющего преобразование типов переменных, содержащихся в контексте семантики реляционного расширения, а также лемму, что преобразование типов переменных при реляционном преобразовании соответствует применению реляционного образа контекста семантики. Таким образом мы подтвердим сохранение корректности типов для всех переменных, содержащихся в исходной программе.

Определение 1. Пусть e — выражение исходного языка, x — узел в синтаксическом дереве выражения e, Γ — контекст, возникающий при выводе типа выражения e в узле x. Тогда будем называть $[\![\Gamma]\!] = \{(s : [\![t]\!]^t) \mid (s : t) \in \Gamma\}$ *реляционным образом* контекста Γ.

Лемма 1. Пусть e — выражение исходного языка, x — узел в синтаксическом дереве выражения e, $\Gamma \vdash e : t$ — состояние при выводе типа выражения e в узле x, $\bar{\Gamma} \vdash \bar{e} : \bar{t}$ — состояние при выводе типа преобразованного выражения $[\![e]\!]^c$ в узле $[\![x]\!]^c$, являющемся образом узла x. Тогда $[\![\Gamma]\!] \subset \bar{\Gamma}$.

Доказательство. Докажем данную лемму индукцией по длине пути от корня синтаксического дерева выражения e до узла x.

В случае базы индукции узлом x является корнем синтаксического дерева выражения e. Этому узлу соответствует корень синтаксического дерева преобразованного выражения $[\![e]\!]^c$. На первом шаге вывода типов контекст пуст. Следовательно, $\Gamma = \varnothing$ и $\bar{\Gamma} = \varnothing$. Поэтому верно, что $[\![\Gamma]\!] = [\![\varnothing]\!] = \varnothing$. Так как $\varnothing \subset \varnothing$, в результате получаем $\varnothing = [\![\Gamma]\!] \subset \bar{\Gamma} = \varnothing$, что доказывает базу индукции.

Для индукционного перехода необходимо убедиться, что для каждого правила вывода типов исходного языка утверждение $[\![\Gamma]\!] \subset \bar{\Gamma}$ сохраняется для всех предпосылок при условии истинности утверждения $[\![\Gamma]\!] \subset \bar{\Gamma}$ для следствия.

Рассмотрим случай, когда при выводе типа узла x применимо одно из правил $\textsc{Bool}_T$, $\textsc{Eq}_T$, $\textsc{Constr}_T$ или $\textsc{App}_T$. Тогда применяя эти правила, мы не пополним контекст Γ новыми переменными. При выводе типа узла $[\![x]\!]^c$ контекст $\bar{\Gamma}$ может только пополняться, поэтому соотношение $\Gamma \subset \bar{\Gamma}$ сохранится.

В случае применения правил $\textsc{Var}_T$, $\textsc{Abs}_T$, $\textsc{Let}_T$ и $\textsc{LetRec}_T$ узлы x и $[\![x]\!]^c$ имеют идентичный оператор в качестве корня, поскольку реляционное преобразование не меняет выражение, если оно является переменной, абстракцией или let-связыванием. Поэтому контексты Γ и $\bar{\Gamma}$ пополнятся идентичными переменными и соотношение $\Gamma \subset \bar{\Gamma}$ сохранится.

Остаётся рассмотреть применение правила $\textsc{Match}_T$. В данном случае корень узла x является сопоставлением с образцом и имеет структуру вида $\underline{\texttt{match}}\ \ e_0\ \ \underline{\texttt{with}}\ \ \{C_i^{k_i}(x_1^i, \ldots, x_{k_i}^i)\ \rightarrow\ \ e_i\}$. Соответствующий ему узел $[\![x]\!]^c$ будет реляционным образом, поэтому в соответствии с правилом реляционного преобразования для сопоставления с образцом имеем следующее:

$$
[\![x]\!]^c \quad = \quad
\begin{array}{l}
\lambda\ q\ \ .\ \underline{\texttt{fresh}}\ \ (q_{e_0}) \\
\quad ([\![e_0]\!]^c\ q_{e_0})\ \ \wedge \\
\quad \bigvee_i\ \ ((\underline{\texttt{fresh}}\ \ (q_1^i \ldots q_{n_i}^i) \\
\qquad\quad (q_{e_0}\ \equiv\ \uparrow C_i^{n_i}(q_1^i, \ldots, q_{n_i}^i))\ \ \wedge \\
\qquad\quad (\lambda\ \ x_1^i \ldots x_{n_i}^i\,.\,[\![e_i]\!]^c) \\
\qquad\qquad (\equiv q_1^i)\ \ \ldots\ \ (\equiv q_{n_i}^i)\ \ q \\
\qquad\ )).
\end{array}
$$

При выводе типов узла x выражения e для каждого подвыражения e_i контекст Γ будет пополнен переменными x_j^i, которым будут соответствовать типы $t_j^{C_i}$, являющиеся типами аргументов конструктора $C_i^{n_i}$. Необходимо убедиться, что для узла $[\![x]\!]^c$ преобразованного выражения $[\![e]\!]^c$ для каждого подвыражения $[\![e_i]\!]^c$ контекст $\bar{\Gamma}$ также будет пополнен переменными x_j^i с преобразованными типами $[\![t_j^{C_i}]\!]^t$. Как мы можем видеть, каждое выражение $[\![e_i]\!]^c$ абстрагировано по всем переменным x_j^i, поэтому данные переменные пополнят контекст $\bar{\Gamma}$ при выводе типов. Остается проверить, что им будет соответствовать правильный тип $[\![t_j^{C_i}]\!]^t$.

Прежде всего нам необходимо определить типы всех логических переменных q_j^i. Эти типы определяются выражениями ($q_{e_0} \equiv \uparrow C_i^{n_i}(q_1^i, \ldots, q_{n_i}^i)$), где данные переменные расположены на позициях аргументов конструкторов $C_i^{n_i}$. Поэтому типы переменных q_j^i равняются $[\![t_j^{C_i}]\!]^o$. Из правила преобразования сопоставления с образцом непосредственно следует, что абстракция ($\lambda\ x_1^i \ldots x_{n_i}^i$ $. [\![e_i]\!]^c$) применяется к набору аргументов $(\equiv q_1^i)$, $\ldots$, $(\equiv q_{n_i}^i), q$. Таким образом, каждой переменной x_j^i будет соответствовать тип выражения $(\equiv q_j^i)$. Так как переменной q_j^i соответствует тип $[\![t_j^{C_i}]\!]^o$, то для выражения $(\equiv q_j^i)$ получаем тип $[\![t_j^{C_i}]\!]^o \to \mathfrak{G}$. Это в точности соответствует требуемому типу, поскольку по правилу преобразования заземленного типа имеем, что $[\![t_j^{C_i}]\!]^t = [\![t_j^{C_i}]\!]^o \to \mathfrak{G}$. Таким образом индукционный переход доказан. □

Данная лемма подтверждает тот факт, что при выводе типа реляционного образа все переменные исходной программы будут типизированы корректными типами.

Доказательство самой теоремы нетрудно выполнить по схеме доказательства леммы 1, применив структурную индукцию с использованием леммы 1 при доказательстве базы индукции.

Данная теорема подтверждает, что введенное реляционное преобразование сохраняет корректность типизации, однако теорема обходит стороной вопрос о сохранении семантики преобразованной программы. Следующий раздел как раз посвящен решению этой проблемы.

3.2 Частичная динамическая корректность реляционного преобразования

Вторая теорема, которая доказана в данной главе, подтверждает частичную динамическую корректность или, другими словами, сохранение семантики при реляционном преобразовании корректной программы исходного языка. Важно отметить, что доказывается именно *частичная* динамическая корректность, которая гарантирует сохранение семантики реляционного образа при вычислении в *прямом направлении*. Иначе говоря, если исходная программа содержит применение некоторой функции f к набору аргументов $e_1, \ldots, e_n$, то сохранение семантики гарантируется для применения $[\![f]\!]_c\ [\![e_1]\!]_c\ \ldots\ [\![e_n]\!]_c\ q$, где

q — свежая переменная. Однако в случае реляционного образа возможны более сложные применения: любой аргумент может быть заменен свежей переменной как полностью, так и частично за счет замены подвыражения аргумента свежей переменной. Подобной реляционной программе невозможно сопоставить какую-либо исходную функциональную программу в качестве прообраза в силу строгой направленности вычисления функциональных программ.

Теорема 2 (о частичной динамической корректности)**.** Пусть выражение первого порядка e имеет тип t, а также существует значение первого порядка v такое, что $e \leadsto^f v$. Тогда $\underline{\texttt{fresh}}\ (x)\ (\llbracket e \rrbracket^c\ x) \leadsto^r (\theta, \varnothing)$, и $\theta(\mathfrak{s}) = v$, где $\mathfrak{s}$ является семантической переменной, ассоциированной с x на первом шаге вычисления.

Прежде всего прокомментируем тот факт, что множество отрицательных подстановок является пустым. Пополнение данного множества возможно только при выполнении конструкции ограничения неравенством. Эта конструкция может быть исполнена в преобразованной программе, только если исходная программа содержит синтаксическое сравнение, примененное к своим аргументам. Во время исполнения преобразованной реляционной программы в прямом направлении (последний аргумент является свежей переменной, остальные аргументы полностью определены) при выполнении конструкции ограничения неравенством оба аргумента являются замкнутыми, что приводит к немедленному разрешению ограничения неравенством без пополнения изначально пустого множества отрицательных подстановок.

Также отметим, что данную теорему нельзя доказать индукцией по длине вывода, поскольку каждое *применение*, содержащееся в исходной программе содержит функцию в качестве левого подвыражения. И эта функция очевидно не является выражением первого порядка. Это ограничение можно было бы снять, если бы можно было доказать обобщение $p \leadsto^f f \Rightarrow \llbracket p \rrbracket^c \leadsto^r \llbracket f \rrbracket^c$ для произвольного p любого типа. Это утверждение, однако, оказалось ложным, поскольку выражение `C ((λ x.x) A)` можно предъявить в качестве контрпримера.

Причина данной проблемы заключается в том, что при преобразовании происходит *функционализация* конструкторов, сопоставлений с образцом и синтаксических сравнений, и, следовательно, изменяется порядок вычисления реляционного образа в сравнении с исходной функциональной программой. Таким образом при доказательстве необходимо решить эту проблему.

Для устранения различий в порядке вычисления исходной программы и реляционного образа, была разработана модифицированная семантика функционального языка, порядок вычисления в которой приближен к порядку вычисления реляционного образа. Данная семантика была названа *откладывающей*, так как она откладывает вычисление конструкторов, сопоставления с образом и синтаксического сравнения. Эта семантика может быть получена из исходной функциональной семантики в два шага. Сначала необходимо рассмотреть сокращенную версию оригинальной функциональной семантики, которая обрабатывает конструкторы, сопоставления с образцом и синтаксические сравнения как вычисленные значения. Затем определить отложенную семантику в виде итеративного применения сокращенной семантики к аргументам этих новых значений (аргументы конструкторов или синтаксического сравнения, а также сопоставляемое значение сопоставления с образцом).

Далее, отметим, что если выражение первого порядка при вычислении возвращает некоторое значение в исходной семантике, то оно также вычисляется к тому же значению в откладывающей семантике. Это свойство основано на следующих наблюдениях:

- обе семантики обладают стандартными свойствами **продвижения** (progress) и **сохранения типизации** (type preservation) [85];
- для обеих семантик верно свойство Черча-Россера [85; 87] для лямбда-исчисления;
- откладывающая семантика применяет подмножество правил исходной семантики.

Теперь приступим к доказательству теоремы. Будем использовать метод симуляции [88; 89] исходной программой в откладывающей семантике с помощью реляционного образа в реляционной семантике. Перед этим сформулируем несколько лемм и определений.

Определение 2. Определим два множества контекстов исходного языка — *множество функциональных контекстов* (1) и *множество атомарных контекстов* (2):

$$C_f = \Box\ e \mid v\ \Box \mid \underline{\texttt{let}}\ x\ \texttt{=}\ \Box\ \underline{\texttt{in}}\ e, \tag{1}$$

$$C_g = \underline{\texttt{match}}\ \ \Box\ \ \underline{\texttt{with}}\ \{p_i \rightarrow e_i\} \mid C^n(\bar{v},\Box,\bar{e}) \mid \Box\texttt{=e} \mid \texttt{v=}\Box. \tag{2}$$

Отметим, что множества функциональных и атомарных контекстов дизъюнкты, а их объединение в точности равно множеству всех контекстов исходного языка.

Лемма 2. Пусть $\langle \mathcal{S}, e\rangle$ — произвольное состояние последовательности вычислений в откладывающей семантике. Тогда выполнено следующее: $\mathcal{S} = C_f^* C_g^*$.

Другими словами, в процессе вычисления в откладывающей семантики стек контекстов можно разделить на два (возможно, пустых) сегмента: все атомарные контексты расположены ниже всех функциональных.

Лемма легко доказывается индукцией по длине вывода.

Определение 3. Определим два множества выражений исходного языка — *множество функциональных выражений* (3) и *множество атомарных выражений* (4):

$$E_f = e_1\, e_2 \mid \lambda x.e \mid \mu f.\lambda x.e \mid \underline{\texttt{let}}\ x\ \texttt{=}\ e_1\ \underline{\texttt{in}}\ e_2 \mid \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ \texttt{=}\ \lambda x.e_1\ \underline{\texttt{in}}\ e_2, \quad (3)$$

$$E_g = (e_1 = e_2) \mid \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i\ \rightarrow\ e_i\}. \mid \texttt{C}^k\ \ (e_1 \ldots e_k). \quad (4)$$

Отметим, что множества функциональных и атомарных выражений дизъюнкты, а их объединение в точности равно множеству всех выражений исходного языка.

Определение 4. Для произвольной подстановки θ *расширенное реляционное преобразование* выражения исходного языка $[\![\bullet]\!]_\theta$ определим следующим образом:

$$\begin{array}{rcl} [\![f]\!]_\theta & = & [\![f]\!]^c, \\ [\![v]\!]_\theta & = & (\lambda x.x \equiv \mathfrak{s}), \text{если}\ \ \theta(\mathfrak{s}) = v. \end{array}$$

Здесь θ — подстановка, f — произвольное функциональное выражение, v — произвольное значение первого порядка в исходной семантике (т. е. композиция конструкторов).

Заметим, что разные случаи в этом определении не дизъюнктны, а во втором случае может быть более одной переменной с запрошенным свойством, поэтому расширенное преобразование определяет набор реляционных выражений.

Лемма 3. Пусть f и e являются произвольными выражениями в исходном языке, а θ — это произвольная подстановка. Тогда справедливо следующее:

$$[\![f[x \leftarrow e]]\!]_\theta = [\![f]\!]_\theta[x \leftarrow [\![e]\!]_\theta].$$

В данном случае равенство необходимо трактовать как равенство двух множеств.

Эта лемма легко доказывается структурной индукцией.

Определение 5. Для произвольной подстановки θ определим *преобразование функционального контекста* $[\![\bullet]\!]_\theta$ следующим образом:

$$\begin{array}{rcl} [\![\Box\, e]\!]_\theta & = & \Box\, [\![e]\!]_\theta, \\ [\![v\, \Box]\!]_\theta & = & [\![v]\!]_\theta\, \Box, \\ [\![\underline{\texttt{let}}\ x\ =\ \Box\ \ \underline{\texttt{in}}\ e]\!]_\theta & = & \underline{\texttt{let}}\ \ x\ =\ \Box\ \ \underline{\texttt{in}}\ \ [\![e]\!]_\theta. \end{array}$$

Здесь e — произвольное функциональное выражение, v — λ-абстракция.

Определение 6. Для произвольной семантической переменной $\mathfrak{s}_1$, $\mathfrak{s}_2$ и произвольной подстановки θ определим *преобразование атомарных контекстов* $[\![\bullet]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2}$ следующим образом:

$$\begin{array}{l} [\![C^k(v_1, \dots, v_{i-1}, \Box, e_{i+1}, \dots, e_k)]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} = \\ \quad \Box \wedge \\ \quad ([\![e_{i+1}]\!]_\theta\ \mathfrak{s}'_{i+1}) \wedge \\ \quad \dots \\ \quad ([\![e_k]\!]_\theta\ \mathfrak{s}'_k) \wedge \\ \quad (\mathfrak{s}_2 \equiv \uparrow C^k(\mathfrak{s}'_1, \dots, \mathfrak{s}'_{i-1}, \mathfrak{s}_1, \mathfrak{s}'_{i+1}, \dots, \mathfrak{s}_k)), \text{если } \theta(\mathfrak{s}'_j) = v_j,\ j < i; \end{array}$$

$$\begin{array}{rcl} [\![\Box = e]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} & = & \Box \wedge \\ & & ([\![e]\!]_\theta\ \mathfrak{s}') \wedge \\ & & (((\mathfrak{s}_1 \equiv \mathfrak{s}') \wedge (\mathfrak{s}_2 \equiv \uparrow \underline{\texttt{true}})) \vee \\ & & ((\mathfrak{s}_1 \not\equiv \mathfrak{s}') \wedge (\mathfrak{s}_2 \equiv \uparrow \underline{\texttt{false}}))); \end{array}$$

$$\begin{array}{rcl} [\![v = \Box]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} & = & \Box \wedge \\ & & (((\mathfrak{s}' \equiv \mathfrak{s}_1) \wedge (\mathfrak{s}_2 \equiv \uparrow \underline{\texttt{true}})) \vee \\ & & ((\mathfrak{s}' \not\equiv \mathfrak{s}_1) \wedge (\mathfrak{s}_2 \equiv \uparrow \underline{\texttt{false}}))), \text{если } \theta(\mathfrak{s}) = v; \end{array}$$

$$\begin{array}{l} [\![\underline{\texttt{match}}\, \Box\ \ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i,\ \ \dots, y_{n_i}^i)\ \to e_i\}]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} = \\ \quad \Box\ \wedge \bigvee_i \\ \quad (\underline{\texttt{fresh}}\ \ (s_1^i \dots s_{n_i}^i) \\ \qquad (\mathfrak{s}_1 \equiv \uparrow C_i^{n_i}(s_1^i, \dots, s_{n_i}^i)) \\ \qquad (\lambda y_1^i. \dots \lambda y_{n_i}^i.[\![e_i]\!]_\theta)\ (\equiv s_1^i) \dots (\equiv s_{n_i}^i)\ \mathfrak{s}_2). \end{array}$$

Здесь $\mathfrak{s}'$ и $\mathfrak{s}'_i$ — произвольные семантические переменные, v_i — произвольные значения в исходном языке, e_i — произвольные выражения в исходном языке. Также потребуем, чтобы θ была неопределена для всех указанных семантических переменных, если явно не указано противоположное.

Определение 7. Для произвольной подстановки θ, произвольной семантической переменной $\mathfrak{s}_m$ и функционального выражения e определим *преобразование стека контекстов* $[\![\bullet]\!]_\theta^{e,\mathfrak{s}_m}$ следующим образом:

$$[\![f_n \dots f_1 g_m \dots g_1]\!]_\theta^{e,\mathfrak{s}_m} = \begin{cases} [\![g_m]\!]_\theta^{\mathfrak{s}_m \mathfrak{s}_{m-1}} \dots [\![g_1]\!]_\theta^{\mathfrak{s}_1 \mathfrak{s}_0}, & \text{если } n = 0 \text{ и } e \in E_g \\ [\![f_n]\!]_\theta \dots [\![f_1]\!]_\theta (\Box\, \mathfrak{s}_m)[\![g_m]\!]_\theta^{\mathfrak{s}_m \mathfrak{s}_{m-1}} \dots [\![g_1]\!]_\theta^{\mathfrak{s}_1 \mathfrak{s}_0}, & \text{иначе.} \end{cases}$$

Здесь $\mathfrak{s}_0 \dots \mathfrak{s}_{m-1}$ являются произвольными уникальными семантическими переменными.

Определение 8. Для произвольной подстановки θ и произвольной семантической переменной $\mathfrak{s}_m$ определим *симуляционное преобразование* $[\![\bullet]\!]_\theta^{\mathfrak{s}_m}$ для выражения исходного языка следующим образом:

$$\begin{aligned} [\![e_1 = e_2]\!]_\theta^{\mathfrak{s}_m} \;=\; & ([\![e_1]\!]_\theta\ \mathfrak{s}'_1) \wedge \\ & ([\![e_2]\!]_\theta\ \mathfrak{s}'_2) \wedge \\ & (((\mathfrak{s}'_1 \equiv \mathfrak{s}'_2) \wedge (\mathfrak{s}_m \equiv {\uparrow}\underline{\texttt{true}})) \vee \\ & ((\mathfrak{s}'_1 \not\equiv \mathfrak{s}'_2) \wedge (\mathfrak{s}_m \equiv {\uparrow}\underline{\texttt{false}}))); \end{aligned}$$

$$\begin{aligned} [\![v = e]\!]_\theta^{\mathfrak{s}_m} \;=\; & ([\![e]\!]_\theta\ \mathfrak{s}'_2) \wedge \\ & (((\mathfrak{s}'_1 \equiv \mathfrak{s}'_2) \wedge (\mathfrak{s}_m \equiv {\uparrow}\underline{\texttt{true}})) \vee \\ & ((\mathfrak{s}'_1 \not\equiv \mathfrak{s}'_2) \wedge (\mathfrak{s}_m \equiv {\uparrow}\underline{\texttt{false}}))), \text{если } \theta(\mathfrak{s}'_1) = v; \end{aligned}$$

$$\begin{aligned} [\![v_1 = v_2]\!]_\theta^{\mathfrak{s}_m} \;=\; & (((\mathfrak{s}'_1 \equiv \mathfrak{s}'_2) \wedge (\mathfrak{s}_m \equiv {\uparrow}\underline{\texttt{true}})) \vee \\ & ((\mathfrak{s}'_1 \not\equiv \mathfrak{s}'_2) \wedge (\mathfrak{s}_m \equiv {\uparrow}\underline{\texttt{false}}))), \text{если } \theta(\mathfrak{s}'_j) = v_j; \end{aligned}$$

$$\begin{aligned} & [\![C^k(v_1, \dots, v_{i-1}, e_i, \dots, e_k)]\!]_\theta^{\mathfrak{s}_m} = \\ & \quad ([\![e_i]\!]_\theta\ \mathfrak{s}'_i) \wedge \\ & \quad \dots \\ & \quad ([\![e_k]\!]_\theta\ \mathfrak{s}'_k) \wedge \\ & \quad (\mathfrak{s}_m \equiv {\uparrow} C^k(\mathfrak{s}'_1, \dots, \mathfrak{s}'_k)), \text{если } \theta(\mathfrak{s}'_j) = v_j, j < i; \end{aligned}$$

$$[\![C^k(v_1,\ldots,v_k)]\!]_\theta^{\mathfrak{s}_m} = (\mathfrak{s}_m \equiv\ C^k(\mathfrak{s}'_1,\ldots,\mathfrak{s}'_k)), \text{если } \theta(\mathfrak{s}'_j) = v_j;$$

$$[\![C^k(v_1,\ldots,v_k)]\!]_\theta^{\mathfrak{s}_m} = (\mathfrak{s}_m \equiv\ \mathfrak{s}'), \text{ если } \theta(\mathfrak{s}') = C^k(v_1,\ldots,v_k);$$

$$\begin{array}{l}[\![\underline{\texttt{match}}\, e\ \ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i,\ \ \ldots, y_{n_i}^i)\ \to e_i\}]\!]_\theta^{\mathfrak{s}_m} = \\ \quad [\![e]\!]_\theta\, \mathfrak{s}'\ \wedge\ \bigvee_i \\ \quad (\underline{\texttt{fresh}}\ (s_1^i \ldots s_{n_i}^i) \\ \qquad (\mathfrak{s}' \equiv \uparrow C_i^{n_i}(s_1^i,\ldots,s_{n_i}^i)) \\ \qquad (\lambda y_1^i.\ldots\lambda y_{n_i}^i.[\![e_i]\!]_\theta)\ (\equiv s_1^i)\ldots(\equiv s_{n_i}^i)\ \mathfrak{s}_m);\end{array}$$

$$\begin{array}{l}[\![\underline{\texttt{match}}\, v\ \ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i,\ \ \ldots, y_{n_i}^i)\ \to e_i\}]\!]_\theta^{\mathfrak{s}_m} = \\ \quad \bigvee_i \\ \quad (\underline{\texttt{fresh}}\ (s_1^i \ldots s_{n_i}^i) \\ \qquad (\mathfrak{s}' \equiv \uparrow C_i^{n_i}(s_1^i,\ldots,s_{n_i}^i)) \\ \qquad (\lambda y_1^i.\ldots\lambda y_{n_i}^i.[\![e_i]\!]_\theta)\ (\equiv s_1^i)\ldots(\equiv s_{n_i}^i)\ \mathfrak{s}_m), \text{если } \theta(\mathfrak{s}') = v.\end{array}$$

Здесь все $\mathfrak{s}'$ и $\mathfrak{s}'_i$ — произвольные семантические переменные, e — произвольное выражение исходного языка, v — произвольное значение для семантики исходного языка. Также потребуем, чтобы θ была неопределена для всех указанных семантических переменных, если явно не указано противоположное.

Определение 9. Пусть $\langle \mathcal{S}, e\rangle$ является состоянием в откладывающей семантике, а $\left\langle \Sigma, \hat{\mathcal{S}}, \hat{e}, (\theta, \varnothing)\right\rangle$ — это состояние в реляционной семантике. Назовём эти состояния *связанными*, если существует такая семантическая переменная q_m такая, что верно следующее:

- $\hat{\mathcal{S}} \in [\![\mathcal{S}]\!]_\theta^{e,\mathfrak{s}_m}$;
- $\hat{e} \in \begin{cases} [\![e]\!]_\theta^{\mathfrak{s}_m} & , \text{ если } e \in E_g \text{ и } \mathcal{S} \cap C_f = \varnothing \\ [\![e]\!]_\theta & , \text{ иначе;} \end{cases}$
- Σ содержит все семантические переменные из $\hat{e}$, $\hat{\mathcal{S}}$, и θ.

Лемма 4. Пусть $v = \mathtt{C}^k(v_1,\ldots,v_k)$ – значение в терминах семантики исходного функционального языка. Тогда для произвольных Σ, $\mathcal{S}$, θ, $\hat{v} \in [\![v]\!]_\theta$ и семантической переменной $\mathfrak{s}$ такой, что $\mathfrak{s} \notin dom(\theta)$, верно (5) или (6).

$$\langle \Sigma, \mathcal{S}, (\hat{v}\,\mathfrak{s}), (\theta,\varnothing)\rangle \rightsquigarrow^* \left\langle \Sigma', \mathcal{S}, \mathfrak{s} \equiv \mathtt{C}^k(\mathfrak{s}'_1,\ldots,\mathfrak{s}'_k), (\theta',\varnothing)\right\rangle \text{ и } \theta'(\mathfrak{s}'_i) = v_i, \qquad (5)$$

$$\langle \Sigma, \mathcal{S}, (\hat{v}\,\mathfrak{s}), (\theta, \varnothing) \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, \mathfrak{s} \equiv \mathfrak{s}', (\theta, \varnothing) \rangle \text{ и } \theta(\mathfrak{s}') = v. \quad (6)$$

Эта лемма легко доказывается индукцией по высоте v.

Лемма 5. Пусть $s = \langle \mathcal{S} = g_m \dots g_1, e \rangle$ — это состояние в откладывающей семантике, g_i — атомарные контексты, e — выражение первого порядка, θ — некоторая подстановка, Σ — множество выделенных семантических переменных, $\mathfrak{s}_m$ — некоторая семантическая переменная, а $\hat{\mathcal{S}} \in [\![\mathcal{S}]\!]_{\theta}^{e, \mathfrak{s}_m}$, $\hat{e} \in [\![e]\!]_{\theta}$ и Σ содержит все семантические переменные из $\hat{\mathcal{S}}$ и θ. Тогда существует такая последовательность шагов в семантике реляционного расширения, что верно следующее:

$$\left\langle \Sigma, \hat{\mathcal{S}}, (\hat{e}\,\mathfrak{s}_m), (\theta, \varnothing) \right\rangle \rightsquigarrow^* \hat{s},$$

где s и $\hat{s}$ связаны. Это верно в предположении, что Σ содержит все семантические переменные из $\hat{\mathcal{S}}$ и θ.

Доказательство этой леммы легко выполнить разбором случаев для выражения e с использованием Леммы 4.

Лемма 6. Пусть $s_1 \rightarrow s_2$ — один шаг вычисления в откладывающей семантике, $\hat{s_1}$ — состояние в реляционной семантике такое, что s_1 и $\hat{s_1}$ связаны. Тогда существует последовательность шагов в реляционной семантике $\hat{s_1} \rightsquigarrow^* \hat{s_2}$ такая, что s_2 и $\hat{s_2}$ связаны.

Доказательство этой леммы легко выполнить разбором случаев для s_1 и конструктивным построением отношения симуляции [88; 89] с использованием Лемм 3, 4, 5.

Лемма 7. Пусть $s_0 = \langle \varnothing, \varepsilon, \underline{\texttt{fresh}}\ (x)\ ([\![e]\!]^c\ x), \iota \rangle$ — начальное состояние в реляционной семантике. Тогда существует последовательность шагов $s_0 \rightsquigarrow^* \hat{s}$ такая, что начальное состояние в откладывающей семантике $\langle \varepsilon, e \rangle$ и $\hat{s}$ связаны.

Доказательство этой леммы немедленно следует из Леммы 5.

Все необходимые определения и леммы сформулированы. Теперь докажем теорему о частичной динамической корректности. Пусть e – выражение первого порядка в исходном языке, которое вычисляется в значение $v = \texttt{C}^k(v_1, \dots, v_k)$ в оригинальной семантике. Тогда после вычисления в откладывающей семантике выражения e будет получено это же значение следующим образом: $\langle \varepsilon, e \rangle \rightarrow^* \langle \varepsilon, v \rangle$.

Тогда по Лемме 7 имеем

$$\langle \varnothing, \varepsilon, \underline{\texttt{fresh}}\ (x)\ (\llbracket e \rrbracket^c\, x), \iota\rangle \rightsquigarrow^* \hat{s},$$

где $\langle \varepsilon, e\rangle$ и $\hat{s}$ связаны. По Лемме 6 существует состояние $\hat{s'}$ в реляционной семантике такое, что $\hat{s} \rightsquigarrow^* \hat{s'}$, где $\langle \varepsilon, v\rangle$ и $\hat{s'}$ связаны. По определению отношения симуляции, $\hat{s'}$ представимо в форме (7) или (8):

$$\left\langle \Sigma, \varepsilon, \mathfrak{s}_0 \equiv \uparrow\!\texttt{C}^k(\mathfrak{s}'_1, \dots, \mathfrak{s}'_k), (\theta, \varnothing)\right\rangle, \theta(\mathfrak{s}'_i) = v_i; \tag{7}$$

$$\langle \Sigma, \varepsilon, \mathfrak{s}_0 \equiv \mathfrak{s}', (\theta, \varnothing)\rangle, \theta(\mathfrak{s}') = v. \tag{8}$$

Здесь $\mathfrak{s}_0$ — первая семантическая переменная, введенная в Σ, и $\mathfrak{s}_0 \notin dom(\theta)$. В обоих случаях, остается сделать один последний шаг в реляционной семантике, который и завершает это доказательство.

Глава 4. Семантика языка miniKanren с процедурой динамического управления порядком конъюнктов

Динамическое управление порядком вычисления операторов в программе является классическим подходом к оптимизации программ не только в логическом программировании. Данный подход упрощает разработку программ, устраняя необходимость выбора оптимального порядка операторов вручную. Также данный подход оказывается полезным при отсутствии статического оптимального порядка. К примеру, в случае множественного исполнения одной функции оптимальный порядок операторов в теле этой функции может быть различным. Динамическое управление порядком вычисления позволит избежать создания нескольких копий этой функции, различающихся лишь порядком операторов.

В классическом реляционном программировании порядок вычисления дизъюнктов и конъюнктов определяется семантикой операторов дизъюнкции и конъюнкции. И если дизъюнкция определяет порядок вычисления независимых ветвей, эффективно управляя порядком вычисления дизъюнктов методом чередования, то конъюнкция отвечает за порядок вычисления операторов в каждой отдельной ветке и строго фиксирует порядок вычисления.

В данной главе представлена формальная семантика языка реляционного miniKanren с классической направленной конъюнкцией. Также описана ангелическая семантика, позволяющая определить понятие справедливости для реляционной конъюнкции. Определён важный с практической точки зрения класс детерминированных семантик языка miniKanren, параметризованных предикатом развертывания для выбора оптимального порядка конъюнктов. Наконец, определена конкретная форма этого предиката, основанная на понятии вполне квазиупорядочения, превращающая параметризованную семантику в справедливую.

4.1 Классическая направленная конъюнкция в языке miniKanren

В данном разделе мы разберём особенности классической направленной конъюнкции в языке miniKanren на нескольких примерах реляционных определений и спецификаций. Начнем с рассмотрения определения $\texttt{append}^o$, которое является отношением конкатенации двух списков.

$$\begin{aligned}\texttt{append}^o = \lambda\, x\; y\; xy\;\; .\;\; & (x \equiv \texttt{[]} \;\wedge\; y \equiv xy) \;\vee \\ & (\underline{\texttt{fresh}}\; (e\; xs\; xys) \\ & \quad x \equiv e : xs \;\wedge \\ & \quad xy \equiv e : xys \;\wedge \\ & \quad \texttt{append}^o\; xs\; y\; xys)\end{aligned}$$

Три аргумента x, y и xy соответствуют трем такием спискам, что конкатенация x и y равна xy. Отношение определяется следующим образом: если x пусто (`[]`), то xy должно быть равно y; в противном случае мы разобьем x на голову e и хвост xs, тогда xy будет равен $e : xys$, где xys — это конкатенация xs и y, полученная рекурсивным вызовом того же отношения (здесь мы используем "`[]`" и "`:`" в качестве имён конструкторов).

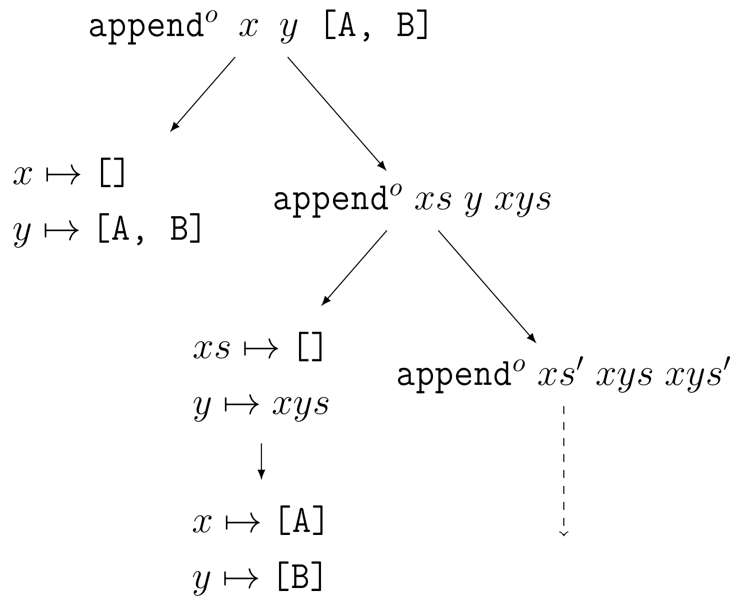
В контексте этого определения цель $\texttt{append}^o$ `[A]` `[B]` x вычислит подстановку $x \mapsto$ `[A, B]`, что является ожидаемым результатом. Однако, поскольку конкатенация фактически работает в обоих направлениях, одно и то же отношение можно использовать и для решения других проблем. Например, вычисление цели $\texttt{append}^o$ `[A]` x `[A, B]` вернёт подстановку $x \mapsto$ `[B]`, что соответствует получению суффикса списка. В то же время, цель $\texttt{append}^o$ x y `[A, B]` вернёт набор из трёх следующих подстановок с точностью до порядка:

$$\begin{array}{ll} x \mapsto \texttt{[]} & y \mapsto \texttt{[A, B]} \\ x \mapsto \texttt{[A]} & y \mapsto \texttt{[B]} \\ x \mapsto \texttt{[A, B]} & y \mapsto \texttt{[]}. \end{array}$$

Другими словами, одно и то же определение отношения может быть использовано для поиска в различных "направлениях" в зависимости от распределения свободных переменных.

Во всех вышеупомянутых примерах при вычислении цели выдается конечный набор ответов за конечное время. Однако это не всегда так. Например,

если мы поместим рекурсивный вызов в определение appendo^o *первым* в конъюнкции, то цель append^o x y [A, B] будет *расходиться* после возврата всех ответов. Иллюстрация этого поведения представлена на следующей диаграмме:



Здесь левая и правая ветви узлов соответствуют вычислению первого и второго дизъюнктов append^o соответственно. Мы начинаем с первого дизъюнкта append^o, что немедленно дает нам первый ответ $x \mapsto$ [], $y \mapsto$ [A, B]. Во втором дизъюнкте мы сталкиваемся с конъюнкцией и начинаем вычислять его левый конъюнкт, который является применением append^o к *свободным* переменным xs, y и xys. После подстановки тела отношения append^o на позицию применения этого отношения мы снова сталкиваемся с дизъюнкцией. Его левый дизъюнкт дает нам подстановку $xs \mapsto$ [], $y \mapsto xys$, которая после возвращения из рекурсивного вызова и вычисления оставшихся конъюнктов, дает второй ответ $x \mapsto$ [A], $y \mapsto$ [B]. Второй дизъюнкт, однако, снова является конъюнкцией, начинающейся с рекурсивного применения append^o к новым свободным переменным. Ясно, что эта ветвь никогда не перестанет расти, и после восстановления третьего ответа вычисление разойдется. Этот пример демонстрирует хорошо известный феномен *неполноты по опровержению* [40] этой конкретной реализации append^o. Можно доказать [90], что первоначальная реализация (с рекурсивным вызовом, помещенным последним в конъюнкции) является полной по опровержению, но только для *линейных* целей, т.е. таких целей, в которых ни одна переменная не встречается в аргументах более одного раза.

Размещение вызовов отношений в конце набора конъюнктов является хорошо известным трюком в реляционном программировании, который часто

помогает улучшить производительность или даже сходимость вычисления цели. Этот трюк, однако, помогает не во всех случаях, например, не помогает, когда в наборе конъюнктов более одного вызова. Рассмотрим следующее отношение $\texttt{revers}^o$, которое ставит в соответствие произвольному списку список, содержащий такие же элементы в обратном порядке.

$$
\begin{array}{l}
\texttt{revers}^o \texttt{ = } \lambda\, x\, y \;\; . \;\; (x \equiv \texttt{[]} \;\wedge\; y \equiv \texttt{[]}) \;\vee \\
\qquad\qquad\qquad\qquad \underline{\texttt{fresh}}\; (e\; xs\; ys) \\
\qquad\qquad\qquad\qquad\quad x \equiv\; e \;\texttt{:}\; xs \;\wedge \\
\qquad\qquad\qquad\qquad\quad \texttt{revers}^o\; xs\; ys \;\wedge \\
\qquad\qquad\qquad\qquad\quad \texttt{append}^o\; ys\; \texttt{[}e\texttt{]}\; y
\end{array}
$$

В этом отношении присутствуют два вызова в одном наборе конъюнктов: $\texttt{revers}^o$ и $\texttt{append}^o$; таким образом, невозможно поставить их обоих последними. С этим конкретным порядком вызовов цель $\texttt{revers}^o\; \texttt{[A, B, C]}\; x$ сходится, но с обратным порядком та же цель расходится после нахождения ответа. Более того, обратный порядок отрицательно влияет на эффективность оценки ответов. При этом цель $\texttt{revers}^o\; x\; \texttt{[A, B, C]}$ демонстрирует симметричное поведение: расходится для данного порядка конъюнктов, и сходится для обратного.

Важной целью разработки языка miniKanren было предоставление чисто декларативного способа определения исполняемых реляционных спецификаций, которые одинаково хорошо работали бы независимо от “направления” поиска. В частности, поскольку мы ожидаем, что конъюнкция и дизъюнкция будут коммутативными и ассоциативными, порядок конъюнктов и дизъюнктов не должен оказывать заметного влияния на поведение спецификации. Как показывают эти примеры, данная цель еще не полностью достигнута. Что еще более интересно, все эти случаи являются проявлениями одного и того же недостатка классической реляционной стратегии поиска — *смещение влево* при исполнении конъюнкции. Продемонстрируем это на следующем искусственном примере. Во-первых, мы определяем два отношения $\texttt{div}^o$ и $\texttt{fail}^o$.

$$
\begin{array}{l}
\texttt{div}^o \;\; \texttt{=} \; \lambda\, x \;\texttt{.}\; \texttt{div}^o \;\; \texttt{x} \\
\texttt{fail}^o \; \texttt{=} \; \lambda\, x \;\texttt{.}\; \texttt{A} \equiv \texttt{B}
\end{array}
$$

В обоих случаях аргумент x является “несущественным” параметром, который никак не влияет на результат вычисления этих отношений. Ясно, что $\texttt{div}^o$ расходится без ответов, а $\texttt{fail}^o$ за один шаг вычисляется как пустой набор ответов. Таким образом, можно было бы ожидать, что конъюнкция этих отношений

потерпит неудачу за один шаг. Действительно, конъюнкция $\texttt{fail}^o$ _ ∧ $\texttt{div}^o$ _ завершается за один шаг с пустым набором ответов. Однако эквивалентная конъюнкция $\texttt{div}^o$_ ∧ $\texttt{fail}^o$_ расходится без ответа (здесь подчеркивание ("_") означает произвольный несущественный аргумент). Объяснение этого факта тривиально — во время вычисления конъюнкции мы сначала вычисляем её левый конъюнкт, пока не будет найден первый ответ. В первом случае мы сразу получаем пустой поток, и вычислить всю конъюнкцию не удается. Во втором случае мы никогда не получаем ответа; в то же время у нас всегда есть оставшиеся шаги для вычисления левого конъюнкта, и это означает, что весь поиск расходится. Такое поведение конъюнкции не только приводит к расхождению в некоторых важных случаях, но также сильно влияет на производительность, засоряя дерево поиска бесконечными ветвями, которые в то же время не позволяют получить никаких результатов. В некоторых случаях этого можно избежать, изменив порядок конъюнктов в определении отношения в соответствии с определенным направлением вычисления. Однако, как показано в работе [49], решение ряда прикладных задач требует, чтобы одно и то же определение выполнялось в различных направлениях одновременно.

Также в ряде случаев не существует статического порядка конъюнктов, при котором вычисление сходится, однако при динамическом изменении порядка конъюнктов сходимость может быть достигнута. Например, рассмотрим запрос ($\texttt{repeat}^o\ A\ q \wedge \texttt{repeat}^o\ B\ q$) по свободной переменной q для отношения $\texttt{repeat}^o$, которая проверяет, что все элементы списка l равны терму e.

$$
\begin{aligned}
\texttt{repeat}^o = \lambda\, e\, l\ .\ & (l \equiv \texttt{[]}) \ \vee \\
& \underline{\texttt{fresh}}\ (ls) \\
& \quad l \equiv e\ :\ ls\ \wedge \\
& \quad \texttt{repeat}^o\ e\ ls
\end{aligned}
$$

Данный запрос требует, чтобы все элементы списка l, с одной стороны, были равны нульместному конструктору A, а с другой — конструктору B, отличному от A. Только пустой список список удовлетворяет данному требованию. Легко видеть, что при вычислении этого запроса в семантике со смещенной влево конъюнкцией процесс вычисления разойдется при любом порядке аргументов. Однако если первый конъюнкт $\texttt{repeat}^o\ A\ q$ вычислить частично, отложив вложенный вызов $\texttt{repeat}^o$, а затем вычислить второй конъюнкт $\texttt{repeat}^o\ B\ q$ в контексте полученных от первого конъюнкта данных, то

процесс вычисления завершится, обнаружив единственный правильный ответ. Другими словами, в этом случае мы не только найдем решение для заданного запроса, но также докажем его единственность.

Описанные выше проблемы смещенной влево конъюнкции актуальны для реляционного преобразования, описанного в главе 2. Это происходит из-за того, что результат данного преобразования является реляционной программой, и эффективность исполнения этой программы непосредственно зависит от выбранной стратегии исполнения конъюнкции. Более того, при реляционном преобразовании не может быть определён оптимальный порядок конъюнктов вследствие недостаточности информации, содержащейся в исходной функциональной программе. Поэтому для эффективного исполнения автоматически полученной результирующей программы в семантике со смещенной влево конъюнкцией требуется "ручное" редактирование каждого реляционного запроса этой программы.

Подводя итог заметим, что в общем случае при смещенной влево стратегии вычисления конъюнкции не существует оптимального статического порядка конъюнктов. Мы предлагаем другую стратегию вычисления, которая динамически откладывает вычисление некоторых ветвей, используя динамический анализ внутренних свойств вычисляемых отношений.

4.2 Ангелическая семантика и справедливость

В этом разделе мы вводим *ангелическую* операционную семантику для языка miniKanren. Вместо фиксированного порядка вычисления конъюнктов данная семантика выбирает следующий для исполнения конъюнкт недетерминировано, тем самым перечисляя все возможные порядки их вычисления. Таким образом, если реляционный запрос расходится в ангелической семантике, то не существует сходящегося порядка вычисления для этого запроса.

Семантика, которую мы предлагаем, основана на *развертке* применения отношений. На каждом шаге в текущем состоянии программы недетерминировано выбирается и развёртывается некоторое применение. Этот процесс продолжается до тех пор, пока в текущем представлении остаются применения. Если на некотором шаге состояние опустело, то вычисление завершается.

Результаты этого (и следующего) раздела опираются на статью [90], в которой представлены две семантики для языка miniKanren: денотационная и операционная. Денотационная семантика является теоретико-множественной и близка к наименьшей модели Эрбранда [55]. Операционная семантика задаёт реляционный поиск решения с чередованием в виде системы переходов с метками [91], и неформально описана в предыдущем разделе. Корректность и полнота поиска с чередованием в операционной семантике доказаны с использованием интерактивного инструмента доказательства теорем Coq [92]. В статье [90] также установлено, что для операционной семантики языка miniKanren некоторые синтаксические преобразования сохраняют семантику преобразуемой реляционной программы и не влияют на множество вычисляемых ответов. Эти преобразования будут играть важную роль в обосновании справедливости представленной нами семантики, чему посвящается следующий раздел.

В качестве первого элемента ангелической семантики мы определим набор следующих *семантических* переменных:

$$\mathcal{A} = \{\alpha_0, \alpha_1, \dots\}.$$

Отметим, что семантические переменные линейно упорядочены, что позволяет использовать их детерминистическим образом при исполнении реляционной программы. Затем естественным образом определим набор *семантических термов* $\mathcal{T}_{\mathcal{A}}$, состоящих из конструкторов и семантических переменных. Мы также будем использовать следующие обозначения для множества свободных переменных в термах и целях: $\mathcal{FV}(\bullet)$. Подстановку *семантического* терма t на место *синтаксической* переменной x в термах и целях будем обозначать так: $\bullet[x \leftarrow t]$.

Далее определим набор *состояний семантики* $\mathfrak{S}$ следующим образом:

$$\begin{array}{rcll}
\mathfrak{S} & = & \times & \text{(конечное состояние)} \\
 & \mid & \mathfrak{S}^o & \text{(промежуточное состояние)} \\
\mathfrak{S}^o & = & \langle \theta, i, r^* \rangle & \text{(листовое состояние)} \\
 & \mid & \mathfrak{S}^o \oplus \mathfrak{S}^o & \text{(дизъюнкция)} \\
r & = & R^k(t_1, \dots, t_k) & \text{(применение отношения)}
\end{array}$$

Промежуточное состояние семантики — это дерево дизъюнкций с листьями вида $\langle \theta, i, r^* \rangle$, где θ — подстановка семантических переменных в семантические термы, i — номер первой незанятой семантической переменной, а r^* — список применений отношений $R^k(t_1, \dots, t_k)$ (возможно пустой),

где R^k является именем отношения, а t_i — это семантические термы. Неформально, состояние семантики представляет собой некоторое представление цели во время её выполнения, когда результаты вычисления конъюнктов собираются в подстановках, а остаточная цель представлена в дизъюнктивной нормальной форме (ДНФ). Если список применений отношений в листовом состоянии пуст, то подстановка представляет собой ответ. Конечное состояние соответствует окончанию выполнения реляционного запроса.

Введем две вспомогательные функции для работы с состояниями. Для объединения двух состояний необходима функция **union**, которую определим так:

$$\begin{array}{rcl}\mathbf{union}\,(\times, S) & = & S, \\ \mathbf{union}\,(S, \times) & = & S, \\ \mathbf{union}\,(S_1, S_2) & = & S_1 \oplus S_2.\end{array}$$

Следующая функция **push** используется для восстановления состояния после выполнения шага одного применения отношения:

$$\begin{array}{rcl}\mathbf{push}\,(_, \times) & = & \times, \\ \mathbf{push}\,(c, S_1 \oplus S_2) & = & \mathbf{push}\,(c, S_1) \oplus \mathbf{push}\,(c, S_2), \\ \mathbf{push}\,(\rho\Box\pi,\ \langle\theta, i, \sigma\rangle) & = & \langle\theta, i, \rho\sigma\pi\rangle\,.\end{array}$$

Первый аргумент функции **push** — это список применений отношений, который содержит *дыру* “$\Box$”. Эта дыра отмечает то место, в котором располагалось развернутое применение отношения. Второй аргумент — это состояние, представляющее результат развертывания. Таким образом, функция **push** распространяет контекст развертывания через результирующее локальное состояние.

Теперь опишем семантику для одношаговой развертки применения отношения. Неформально, мы берем применение символа отношения к некоторым семантическим термам и заменяем эти термы для соответствующих аргументов в теле определения отношения, и далее выполняем все унификации и преобразовываем остальную часть тела в ДНФ. В результате мы получаем некоторое состояние семантики. Отметим, что мы не выполняем развертку применений отношений внутри тела отношения. Дадим формальное определение одношаговой развертке в терминах семантики большого шага “$\Rightarrow$” с помощью следующего правила:

$$\frac{R = \lambda \bar{x}.b \qquad \langle \theta, i, \varepsilon \rangle \vdash b\,[\bar{x} \leftarrow \bar{t}] \rightsquigarrow S}{\langle \theta, i \rangle \vdash R\,(\bar{t}) \Rightarrow S} \quad [\textsc{Unfold}]$$

Здесь и далее $\bar{\bullet}$ обозначает вектор термов или переменных. В заключении этого правила в качестве окружения используется пара, состоящая из подстановки и номера первой незанятой семантической переменной. В посылке правила применяется ещё одна семантика — семантика малого шага, задаваемая в виде отношения "$\rightsquigarrow$", которое использует определенное ранее состояние семантики в качестве окружения. Это отношение описывает локальные вычисления внутри тела развернутого применения. Определение этого отношения дано на рис. 4.1.

Правило [End] распространяет конечное состояние на все остальные вычисления. Правила [UnifyFail] и [UnifySucc] кодируют этапы унификации: если для данных термов в подстановке θ существует наибольший общий унификатор, то обновим эту подстановку. В противном случае результатом вычисления будет конечное состояние. Правило [App] сохраняет применение отношения, добавляя его в список применений в текущем листовом состоянии. Правило [Fresh] соответствует выделению свежей семантической переменной. Берём первую нераспределённую семантическую переменную и заменяем ее свежесвязанной синтаксической; мы также увеличиваем номер первой незанятой

$$\times \vdash g \rightsquigarrow \times \quad [\textsc{End}]$$

$$\frac{\nexists\, mgu\,(t_1, t_2, \theta)}{\langle \theta, i, c \rangle \vdash t_1 \equiv t_2 \rightsquigarrow \times} \quad [\textsc{UnifyFail}]$$

$$\frac{\theta' = mgu\,(t_1, t_2, \theta)}{\langle \theta, i, c \rangle \vdash t_1 \equiv t_2 \rightsquigarrow \langle \theta', i, c \rangle} \quad [\textsc{UnifySucc}]$$

$$\langle \theta, i, c \rangle \vdash R\,(\bar{t}) \rightsquigarrow \langle \theta, i, cR\,(\bar{t}) \rangle \quad [\textsc{App}]$$

$$\frac{\langle \theta, i+1, c \rangle \vdash g\,[x \leftarrow \alpha_i] \rightsquigarrow S}{\langle \theta, i, c \rangle \vdash \underline{\texttt{fresh}}(x)\, g \rightsquigarrow S} \quad [\textsc{Fresh}]$$

$$\frac{\langle \theta, i, c \rangle \vdash g_1 \rightsquigarrow S_1 \qquad \langle \theta, i, c \rangle \vdash g_2 \rightsquigarrow S_2}{\langle \theta, i, c \rangle \vdash g_1 \vee g_2 \rightsquigarrow \mathbf{union}\,(S_1, S_2)} \quad [\textsc{DisjGoal}]$$

$$\frac{S_1 \vdash g \rightsquigarrow S_3 \qquad S_2 \vdash g \rightsquigarrow S_4}{S_1 \oplus S_2 \vdash g \rightsquigarrow \mathbf{union}\,(S_3, S_4)} \quad [\textsc{DisjState}]$$

$$\frac{\langle \theta, i, c \rangle \vdash g_1 \rightsquigarrow S \qquad S \vdash g_2 \rightsquigarrow S'}{\langle \theta, i, c \rangle \vdash g_1 \wedge g_2 \rightsquigarrow S'} \quad [\textsc{Conj}]$$

Рис. 4.1 — Семантика малого шага локальных вычислений

переменной на единицу. Правила [DisjGoal] и [DisjState] описывают вычисление дизъюнкции. Первое правило вычисляет оба дизъюнкта и объединяет результирующие состояния с помощью функции **union**. Второе правило обрабатывает случай, когда текущее окружение является дизъюнкцией. Как и в первом случае, мы проводим независимые вычисления и объединяем результаты. Наконец, правило [Conj] описывает вычисление конъюнкции. В данном случае мы сначала вычисляем левую подцель, получая состояние S, а затем вычисляем правую цель в контексте S, получая окончательный результат S'. Следует отметить, что поскольку мы не делаем никаких развёрток, процесс всегда сходится к некоторому состоянию.

Непосредственно саму ангелическую семантику определяем с использованием семантики развертки — см. рис. 4.2. Эта семантика определяется в терминах системы переходов с метками над множеством состояний. Набор меток семантики определяется как $\mathcal{L} = \circ \mid \theta$, где “$\circ$” означает отсутствие ответа, а θ — ответ в виде подстановки.

$$\langle \theta, i, \varepsilon \rangle \xrightarrow{\theta} \times \qquad [\textsc{Answer}]$$

$$\frac{\langle \theta, i \rangle \vdash c \Rightarrow S}{\langle \theta, i, \rho c \pi \rangle \xrightarrow{\circ} \mathbf{push}\,(\rho \square \pi, S)} \qquad [\textsc{ConjUnfold}]$$

$$\frac{S_1 \xrightarrow{\alpha} \times}{S_1 \oplus S_2 \xrightarrow{\alpha} S_2} \qquad [\textsc{Disj}]$$

$$\frac{S_1 \xrightarrow{\alpha} S_1' \qquad S_1' \neq \times}{S_1 \oplus S_2 \xrightarrow{\alpha} S_2 \oplus S_1'} \qquad [\textsc{DisjStep}]$$

Рис. 4.2 — Ангелическая семантика языка miniKanren

Правило [Answer] соответствует ситуации, когда в рассматриваемом состоянии не осталось ни одного применения отношения. В этом случае мы возвращаем текущую подстановку в качестве ответа. Если текущее состояние представляет собой список с непустым списком применений отношений, то мы недетерминировано выбираем одно для развёртки в соответствии с правилом [ConjUnfold]. После развёртки мы создаем новое состояние, помещая оставшийся список применений в результирующее состояние. Наконец, если текущее состояние — дизъюнкция, то мы выполняем шаг семантики для левого дизъюнкта S_1. Если результатом является конечное состояние, то мы по правилу [Disj] устанавливаем S_2 в качестве остаточного состояния. В противном случае

по правилу [DisjStep] строим новое остаточное состояние путем формирования новой дизъюнкции, в которой правое исходное подсостояние помещается влево, а остаточное состояние после вычисления левого помещается вправо. С помощью данной перестановки реализуется метод чередования, являющийся важной особенностью реляционного поиска. Отметим, что эти два правила дизъюнкции повторяют соответствующие правила в [90].

Чтобы применить ангелическую семантику к спецификации $D_1 D_2 \dots D_k \diamond g$, нам нужно заменить все свободные переменные в g на семантические переменные и преобразовать их в начальное состояние $S^0(g)$:

$$\langle \Lambda, n, \varepsilon \rangle \vdash g\,[x_0 \leftarrow \alpha_0, \dots, x_{n-1} \leftarrow \alpha_{n-1}] \rightsquigarrow S^0(g).$$

Здесь мы предполагаем, что $\mathcal{FV}(g) = \{x_0, \dots, x_{n-1}\}$, Λ является пустой подстановкой. Определения спецификаций используются позже, на этапах развёртки применений в состоянии $S^0(g)$.

Ангелическая семантика сохраняет детерминированное чередование слева направо для классического поиска языка miniKanren и обрабатывает конъюнкцию недетерминировано, описывая все возможные сценарии их выполнения. Например, цель (fail^o _ $\land$ div^o _) из предыдущего раздела сойдётся в соответствии с ангелической семантикой, если мы рассмотрим ветку её вычисления, в которой был развёрнут левый конъюнкт:

$$\cfrac{\cfrac{\langle \Lambda,\, 1, \varepsilon \rangle \vdash A \equiv B \rightsquigarrow \times}{\langle \Lambda, 1 \rangle \vdash \texttt{fail}^o\, \alpha_0 \Rightarrow \times}}{\langle \Lambda, 1,\, \texttt{fail}^o\, \alpha_0 \land \texttt{div}^o\, \alpha_0 \rangle \xrightarrow{\circ} \times}$$

Действительно, развёртка fail^o приведет нас к конечному состоянию за один шаг. В то же время, если мы решим разворачивать только правый конъюнкт, то вычисление разойдётся:

$$\cfrac{\cfrac{\langle \Lambda,\, 1, \varepsilon \rangle \vdash \texttt{div}^o\, \alpha_0 \rightsquigarrow \langle \Lambda,\, 1, \texttt{div}^o\, \alpha_0 \rangle}{\langle \Lambda, 1 \rangle \vdash \texttt{div}^o\, \alpha_0 \Rightarrow \langle \Lambda,\, 1, \texttt{div}^o\, \alpha_0 \rangle}}{\langle \Lambda,\, 1,\, \texttt{fail}^o\, \alpha_0 \land \texttt{div}^o\, \alpha_0 \rangle \xrightarrow{\circ} \langle \Lambda,\, 1,\, \texttt{fail}^o\, \alpha_0 \land \texttt{div}^o\, \alpha_0 \rangle}$$

Поскольку состояния до и после развёртки div^o совпадают, этот шаг будет повторяться бесконечное число раз.

Следующая лемма подтверждает соответствие между денотационной семантикой языка miniKanren из работы [90] и ангелической семантикой.

Лемма 8 (Корректность и полнота)**.** Ангелическая семантика обладает свойствами корректности и полноты.

Полнота непосредственно следует из полноты детерминированной семантики, доказанной в работе [90]. В самом деле, любой вывод в детерминированной семантике может быть воспроизведен в ангелической. Доказательство корректности повторяет доказательство корректности операционной семантики в работе [90].

Ангелическая семантика позволяет формально определить то, что мы можем назвать *справедливостью* стратегии вычисления конъюнкции. Сначала определим понятие *сходимости*.

Определение 10 (Сходимость)**.** Цель g *сходится* (обозначение: $g \Downarrow$), если существует последовательность вывода из начального состояния g в конечное состояние, т.е. $\left\langle S^0\left(g\right), \times\right\rangle \in \rightarrow^*$.

Теперь у нас есть всё необходимое, чтобы формально определить понятие *справедливости*.

Определение 11 (Справедливость)**.** Корректная и полная семантика в виде системы переходов с метками является *справедливой*, если цель g сходится к конечному состоянию всякий раз, когда $g \Downarrow$.

В следующем разделе мы рассмотрим справедливую и детерминированную семантику для языка miniKanren.

4.3 Обобщенная семантика языка miniKanren, оснащенная предикатом выбора

В этом разделе мы представляем детерминированную справедливую семантику для языка miniKanren. С теоретической точки зрения в этой задаче нет ничего сложного: поскольку метод чередования при вычислении дизъюнкции обеспечивает полноту поиска, чередование при вычислении конъюнкции обеспечит его справедливость. Другими словами, достаточно перемещать “фокус” вычисления с одного конъюнкта на другой после выполнения одного шага

(или любого конечного числа шагов), чтобы достичь цели. Увы, на практике такой подход приводит к критическому снижению производительности. Проблема заключается в том, чтобы угадать подходящий момент для приостановки вычисления конъюнкта: если конъюнкт “собирается” дать ответ, то его вычисление должно быть продолжено. Таким образом, мы ищем определенный динамический критерий, который бы задал нужное время приостановки вычисления конъюнкта, принимая во внимание внутренние свойства исполняемой программы.

Интересно, что точно такая же проблема возникает и в области *метавычислений*. В суперкомпиляции [93] символическое выполнение программы может привести к потенциально бесконечной развертке. Чтобы справиться с этой трудностью, существует техника *обобщения* [94], которая используется для гарантии сходимости процесса. Однако нежелательны как преждевременные, так и отсроченные обобщения, ведь при таких обобщениях метод суперкомпиляции приводят к построению неэффективных программ. На практике подтвердила свою применимость суперкомпиляция с обобщением, основанным на понятии *вполне квазиупорядочения* [95; 96].

Определение 12. Вполне квазиупорядочение на множестве Σ — это такой предпорядок “$\trianglelefteq$”, что в произвольной бесконечной последовательности $x_1, x_2, \ldots$ элементов Σ найдутся такие элементы x_i и x_j, что $i < j$ и $x_i \trianglelefteq x_j$.

С практической точки зрения вполне квазиупорядочение может помочь обнаружить расходимость определенной последовательности вычислительных шагов. В нашем подходе мы пользуемся этой идеей для получения справедливой семантики. Во-первых, мы определяем очень общую детерминистическую семантику, снабженную определенным предикатом, который определяет порядок переключения между конъюнктами. Затем мы доказываем, что при определенных требованиях к предикату семантика становится справедливой. Наконец, мы представляем конкретный предикат, основанный на конкретном вполне квазиупорядочении и показываем, что он удовлетворяет этим требованиям.

Введем множество *дополненных* состояний $\mathfrak{R}$:

$$
\begin{array}{rcll}
\mathcal{H} & = & \langle \theta, r\rangle^* & \text{история развёрток} \\
\mathfrak{R} & = & \times & \text{конечное состояние} \\
 & | & \mathfrak{R}^o & \text{не конечное состояние} \\
\mathfrak{R}^o & = & \langle \theta, i,\ \langle r, \mathcal{H}\rangle^*\rangle & \text{листовое состояние с историями развёрток} \\
 & | & \mathfrak{R}^o \oplus \mathfrak{R}^o & \text{дизъюнкция}
\end{array}
$$

Данное определение дополнено понятием *истории развёрток.* История развёрток — это список пар подстановок и применений отношений (возможно, пустой). В листовом состоянии теперь каждое применение отношения оснащено историей развёрток, которые привели к этому применению.

Функции **union** и **push**, введённые выше, сохраняют своё определение для расширенных состояний, поскольку наличие истории не меняет их поведения. Кроме того, нам нужна новая функция **set**, которая принимает обычное состояние и историю, конструируя расширенное состояние путем присоединения истории к каждому приложению отношений в этом состоянии:

$$
\begin{array}{rcl}
\mathbf{set}\,(\times, _) & = & \times, \\
\mathbf{set}\,(S_1 \oplus S_2, h) & = & \mathbf{set}\,(S_1, h) \oplus \mathbf{set}\,(S_2, h), \\
\mathbf{set}\,(\langle \theta, i, r_1 \dots r_n\rangle, h) & = & \langle \theta, i, (r_1, h) \dots (r_n, h)\rangle\,.
\end{array}
$$

Наша семантика (рис. 4.3, отношение перехода обозначается “$\rightarrow_p$”) параметризуется предикатом $\mathcal{P}\,(\theta, r, h)$, где θ — подстановка, r — применение отношения и h — история развёрток. Неформально говоря, $\mathcal{P}\,(\theta, r, h)$ определяет, желательно ли разворачивать применение r. Сама семантика полностью повторяет одношаговое отношение развёртки “$\Rightarrow$” и правила [Answer], [Disj] и [DisjStep] из определения ангелической семантики. Одношаговая развёртка возвращает обычное, недополненное состояние, в то время как правила обрабатывают обогащенные состояния (но синтаксически эти правила полностью сохраняют свою форму). Но правило для обработки конъюнкции [ConjUnfold] заменяется двумя другими правилами: [ConjClear] и [ConjUnfold*]. Если $\mathcal{P}$ истинно хотя бы для одного применения отношения, мы применяем правило [ConjUnfold*] и разворачиваем самое левое такое применение, задавая историю полученных приложений с помощью функции **set**. Если предикат ложен для всех применений отношений, и есть хотя бы одна непустая история развёрток, то применяем правило [ConjClear], которое удаляет историю для каждого приложения, заменяя на пустую историю ε.

$$\frac{\bigvee_{j=1}^{n} h_j \neq \varepsilon \qquad \bigwedge_{j=1}^{n} \neg \mathcal{P}\left(\theta, r_j, h_j\right)}{\langle \theta, i, (r_1, h_1) \ldots (r_n, h_n) \rangle \xrightarrow{\circ}_p \langle \theta, i, (r_1, \varepsilon) \ldots (r_n, \varepsilon) \rangle} \quad [\textsc{ConjClear}]$$

$$\frac{\bigwedge_{j=1}^{k-1} \neg \mathcal{P}\left(\theta, r_j, h_j\right) \qquad \mathcal{P}\left(\theta, r_k, h_k\right) \qquad \langle \theta, i \rangle \vdash r_k \Rightarrow R}{\begin{array}{l}\langle \theta, i, (r_1, h_1) \ldots (r_{k-1}, h_{k-1})(r_k, h_k)\pi \rangle \xrightarrow{\circ}_p \\ \qquad \mathbf{push}\left((r_1, h_1) \ldots (r_{k-1}, h_{k-1})\square\pi, \mathbf{set}\left(R,\ (\theta, r_k) : h_k\right)\right)\end{array}} \quad [\textsc{ConjUnfold}^*]$$

Рис. 4.3 — Обобщённая семантика языка miniKanren, параметризованная предикатом выбора $\mathcal{P}$

Параметризация предикатом выбора даёт нам целое семейство семантик, и не все элементы этого семейства хороши. Например, если предикат всегда остается ложным, то из состояния $\langle \Lambda, i,\ (r, \varepsilon) \rangle$ нельзя выполнить никаких шагов, что ставит под угрозу полноту. Следующая лемма предоставляет необходимое условие полноты.

Лемма 9. Если верно, что $\forall\ \theta, r\ :\ \mathcal{P}\left(\theta, r, \varepsilon\right)$, тогда для любого промежуточного состояния R существует состояние R' такое, что выполняется $R \xrightarrow{\alpha}_p R'$.

Доказательство. Докажем это утверждение методом структурной индукции по R.

Случай $R = R_1 \oplus R_2$ покрывается правилами [Disj] и [DisjStep].

Случай $R = \langle \theta, i, \varepsilon \rangle$ покрывается правилом [Answer].

Случай $R = \langle \theta, i, (r_1, h_1) \ldots (r_n, h_n) \rangle$ при условии $\bigvee_{j=1}^{n} \mathcal{P}\left(\theta, r_j, h_j\right)$ покрывается правилом [ConjUnfold*].

Случай $R = \langle \theta, i, (r_1, h_1) \ldots (r_n, h_n) \rangle$ при условиях $\bigvee_{j=1}^{n} h_j \neq \varepsilon$ и $\bigwedge_{j=1}^{n} \neg \mathcal{P}\left(\theta, r_j, h_j\right)$ покрывается правилом [ConjClear].

Единственный оставшийся случай — это $R = \langle \theta, i, (r_1, h_1) \ldots (r_n, h_n) \rangle$ при условии $\bigwedge_{j=1}^{n} h_j = \varepsilon$ и $\bigwedge_{j=1}^{n} \neg \mathcal{P}\left(\theta, r_j, h_j\right)$. Отсюда, в частности, следует, что $\neg \mathcal{P}\left(\theta, r_1, \varepsilon\right)$, но по условию леммы $\forall\ \theta, r\ :\ \mathcal{P}\left(\theta, r, \varepsilon\right)$, что невозможно. □

Все предикаты, которые мы будем рассматривать далее, будут тривиально удовлетворять условиям леммы 9.

Выбирая предикат, мы можем получить различные существующие семантики. Например, выбрав в качестве предиката тождественную истину, мы

получим семантику со смещенной влево конъюнкцией. Если предикат ограничивает размер истории, т.е.

$$\mathcal{P}_N(\theta, r, h) = length(h) \leqslant N,$$

то мы получаем семантику, которая последовательно разворачивает применения отношений слева направо до некоторой глубины $N > 0$. Эта техника напоминает *избегающие опустошения потоки* [40] — реализация потенциально бесконечных списков, основанная на использовании определенной структуры данных, называемой "фернами" ("ferns" [97]), которая разработана для того, чтобы откладывать расходящиеся вычисления.

В контексте данной работы важным классом предикатов являются *вполне квазиупорядочивающие предикаты*, которые мы рассмотрим в следующем разделе.

4.4 Семантика языка miniKanren, оснащенная вполне квазиупорядочивающим предикатом выбора

Пусть "$\trianglelefteq$" — вполне квазиупорядочение на множестве пар подстановок и применений отношений. Тогда определяем вполне квазиупорядочивающий предикат следующим образом:

$$\mathcal{P}_{\trianglelefteq}(\theta, r, h) = \nexists \langle \theta_0, r_0 \rangle \in h \ : \ \langle \theta_0, r_0 \rangle \trianglelefteq \langle \theta, r \rangle .$$

Данный класс предикатов гарантирует, что каждое применение отношения, содержащееся в состоянии, будет в конечном итоге развернуто после конечного числа шагов. Данное утверждение доказано в главе 5.

Как обсуждалось ранее, создание справедливой семантики не представляет сложности, если не принимать во внимание вопросы производительности. С этой точки зрения вся конструкция справедливой семантики с помощью вполне квазиупорядочения ничего не дает, если не представлено конкретный предикат вполне квазиупорядочения, который позволит добиться эффективной реализации. Это вполне квазиупорядочение, во-первых, не должно требовать больших вычислительных затрат и, во-вторых, оно должно обеспечивать хорошее предсказание расхождения конъюнктов. Например, *гомеоморфное вложение* [98],

часто используемое в качестве критерия остановки развёртки программы в методе суперкомпиляции, в нашем случае и требует значительных вычислительных затрат, и неэффективно предсказывает расхождение конъюнктов.

Теперь мы представляем предикат вполне квазиупорядочения, который предоставит нам практически важную версию справедливой семантики.

Во-первых, мы определяем следующее отношение "$\leqslant_h$" на кортежах семантических термов.

Определение 13. Пусть $\langle t_1^1, \ldots, t_n^1 \rangle$ и $\langle t_1^2, \ldots, t_n^2 \rangle$ — это кортежи семантических термов. Тогда следующие утверждения являются равносильными:

$$\langle t_1^1, \ldots, t_n^1 \rangle \leqslant_h \langle t_1^2, \ldots t_n^2 \rangle \quad \Leftrightarrow \quad \forall i \; : \; height\,(t_i^1) \leqslant height\,(t_i^2)$$

Отношение "$\leqslant_h$" сравнивает термы по их высоте. Оно требует, чтобы хотя бы один терм левого кортежа был строго меньше, чем соответствующий терм из правого кортежа. Остальные левые термы должны быть не больше соответствующих термов в правом кортеже.

Лемма 10. Отношение "$\leqslant_h$" является вполне квазиупорядочением.

Доказывается индукцией по сумме высот термов.

Теперь мы можем определить отношение "$\leqslant_{sr}$" на множестве пар подстановок и применений отношений. Предварительно для каждого отношения выявляем множество его структурно-рекурсивных аргументов. Для некоторого отношения R аргумент является структурно-рекурсивным, если при каждом его рекурсивном вызове значение аргумента структурно уменьшается. Это простое синтаксическое свойство отношения, которое можно проверить статически для каждого отношения реляционной программы перед вычислением.

Определение 14. Пусть θ_1, θ_2 являются подстановками, $r_1 = R\,(t_1^1, \ldots, t_n^1)$ и $r_2 = R\,(t_1^2, \ldots, t_n^2)$ — произвольные применения отношения R, $j_1, \ldots, j_k$ — номера структурно-рекурсивных аргументов в R. Тогда определим отношение "$\leqslant_{sr}$" для применений отношения R в контексте соответствующих им подстановок. Применения r_1 и r_2 удовлетворяют отношению $(\theta_1, r_1) \leqslant_{sr} (\theta_2, r_2)$, если их структурно-рекурсивные аргументы удовлетворяют условию $(t_{j_1}^1 \theta_1, \ldots, t_{j_k}^1 \theta_1) \leqslant_h (t_{j_1}^2 \theta_2, \ldots, t_{j_k}^2 \theta_2)$

Лемма 11. Отношение "$\leqslant_{sr}$" является вполне квазиупорядочением.

Доказательство непосредственно следует из леммы 10.

Легко видеть, что в этом доказательстве фактически не использовалась структурная рекурсия; на самом деле отношение "$\leqslant_{sr}$" остается вполне квазиупорядоченным, даже если мы выбираем все аргументы (или произвольные). Выбор именно такого определения заключается в том, что структурная рекурсия продемонстрировала наилучшие результаты на практике; другими словами, это хорошая эвристика для выбора вполне квазиупорядочения.

Глава 5. Справедливость конъюнкции в семантике языка miniKanren, оснащенной вполне квазиупорядочивающим предикатом выбора

В данной главе представлено формальное доказательство справедливости конъюнкции в семантике языка miniKanren с процедурой динамического управления порядком конъюнктов в случае использования вполне квазиупорядочивающего предиката выбора.

Прежде всего сделаем ряд упрощающих дальнейшие рассуждения замечаний о зависимости сходимости состояния и его листьев и о пополнении подстановки при развертке применения отношения.

5.1 Сходимость листьев состояния ангелической семантики и семантики, оснащенной квазиупорядочивающим предикатом выбора

Нам нужно связать две семантики — ангелическую, представленную в разделе 4.2, и справедливую семантику, оснащенную вполне квазиупорядочивающим предикатом выбора, представленную в разделе 4.3. Мы также намереваемся доказать, что в обоих семантиках произвольная цель одновременно сходится или расходится.

Обе семантики представляются как системы переходов с одним и тем же алфавитом меток и древовидными состояниями, в которых внутренние узлы соответствуют дизъюнкциям, а листья содержат упорядоченные конъюнкции применений отношений, подстановку и номер первой свежей семантической переменной. Более того, обе семантики используют идентичный набор правил, соответствующих случаю исполнения дизъюнкции. Также обе семантики корректны и полны относительно денотационной семантики языка miniKanren [90]. Полнота означает, что в состоянии ни один лист не может быть исключен из рассмотрения — каждый лист состояния будет рассмотрен за конечное число шагов и, следовательно, некоторое применение отношения будет развернуто в каждом листе. Это позволяет нам абстрагироваться от конкретных форм состо-

яний и вместо этого рассматривать их как наборы листьев. Следующая лемма оправдывает этот переход.

Лемма 12 (о сходимости листьев)**.** Пусть s является состоянием с набором листьев $\{\omega_i\}$. Тогда s сходится (в ангелической и справедливой семантике) тогда и только тогда, когда каждый лист ω_i сходится.

Доказательство. Лемма вытекает из следующего более общего факта.

Пусть $s \to s_1 \to s_2 \to \dots$ является (конечной или бесконечной) последовательностью вывода в ангелической или справедливой семантике (можно рассмотреть только ангелическую семантику, поскольку семантика, оснащенная вполне квазиупорядочивающим предикатом, является частным случаем ангелической). Тогда верно следующее:

- каждое состояние s_i может быть представлено в виде $s_i\,[w_1, \dots, w_k]$, где $\{w_j\}$ является дизъюнктным набором поддеревьев состояния s_i и содержит все его листья в порядке, соответствующем левостороннему обходу состояния s_i;
- каждый шаг ангелической семантики может быть представлен в виде $s_i\,[\dots w_{j-1}, w_j, w_{j+1}, \dots] \;\to\; s_{i+1}\,[\dots w'_{j-1}, w_j^1, \dots, w_j^m, w'_{j+1} \dots]$, где набор листьев (возможно пустой) $\{w_j^1, \dots, w_j^m\}$ получен из развёртки некоторого применения отношения, содержащегося в листе w_j, и остальных применений отношений из листа w_j посредством функции **push**, а набор листьев $\{w'_1, \dots, w'_{j-1}, w'_{j+1}, \dots, w'_k\}$ является перестановкой набора $\{w_1, \dots, w_{j-1}, w_{j+1}, \dots, w_k\}$;
- для каждого листа w_j можно выделить последовательность его вывода $\omega_k \to \dots \to w_j$, началом которого будет некоторый лист ω_k состояния s.

Докажем этот факт индукцией по длине пути между состояниями s и s_i. В случае базы индукции все листья состояния s в точности совпадают с набором $\{\omega_i\}$, элементы которого удовлетворяют всем требованиям.

Пусть по индукционному предположению есть промежуточное состояние $s_i\,[w_1, \dots, w_k]$, все листья которого удовлетворяют требованиям. Применение правил дизъюнкции пройдёт по состоянию s_i от корня к некоторому листу (если состояние не содержит дизъюнкций, то весь случай вырождается в $s_i = w_1$ и утверждение становится тривиальным) и, следовательно, придёт к определенному листу w_j. Одно из применений в листе w_j будет развернуто по правилу

для конъюнкции. Тогда следующее состояние можно представить как

$$s_{i+1}\,[\ldots w'_{j-1}, w^1_j, \ldots, w^m_j, w'_{j+1}\ldots],$$

где на месте листа w_j будут помещены листья результата развёртки w_j, дополненные остальными применениями отношений из листа w_j, которые не были развёрнуты.

Другими словами, любое вычисление состояния может быть разложено на независимые вычисления его листьев и наоборот. Поэтому сходимость состояния гарантирует сходимость каждого листа, как и сходимость каждого листа гарантирует сходимость всего состояния. □

5.2 Сохранение сходимости ангелической семантики

Следующее наблюдение касается "коммутативности" последовательных унификаций. Пусть $p_{1,2}, q_{1,2}$ — некоторые термы. Тогда справедливо следующее утверждение:

$$mgu\,(p_1, p_2)\cdot mgu\,(q_1, q_2) = mgu\,(q_1, q_2)\cdot mgu\,(p_1, p_2).$$

Действительно, последовательности унификаций $mgu\,(p_1, p_2)\cdot mgu\,(q_1, q_2)$ и $mgu\,(q_1, q_2)\cdot mgu\,(p_1, p_2)$ являются наибольшим общим унификатором для пар термов (p_1, p_2) и (q_1, q_2). Если они различны, то термы $C\,(p_1, q_1)$ и $C\,(p_2, q_2)$, где C является некоторым бинарным конструктором, будут иметь *различные* наибольшие общие унификаторы, что невозможно[1].

С учетом этого приходим к следующему свойству. Пусть $\theta \xrightarrow{u_1;\,u_2;\ldots u_k} \theta'$ является последовательностью унификаций $u_1, u_2, \ldots, u_k$, преобразующих подстановку θ в подстановку θ'. Тогда для любой перестановки π верно, что

$$\theta \xrightarrow{u_{\pi(1)};\,u_{\pi(2)};\ldots u_{\pi(k)}} \theta'.$$

Еще одна тонкость касается порядка введения свежих семантических переменных. Пусть у нас есть листовое состояние $\langle\theta, i, ab\rangle$, где θ — подстановка,

[1]Строго говоря, обоснованность этих утверждений зависит от представления подстановки. Например, в конкретном представлении подстановки $[x \mapsto y]$ и $[y \mapsto x]$ могут быть не равными, но эквивалентными.

a и b — применения отношений, а i — следующая свободная семантическая переменная. В ангелической семантике мы можем выполнять развертку a и b в произвольном порядке. Поскольку тела отношений, используемых в применениях a и b могут содержать конструкции `fresh`, то эти конструкции будут вычисляться в разном порядке, предоставляя разные конкретные семантические переменные и разные подстановки. Однако эти подстановки будут α-эквивалентны. Данное наблюдение отражает следующий интуитивно тривиальный факт: порядок введения семантических переменных не важен, пока ни одна переменная не выделяется более одного раза при вычислении в одной и той же ветви. Таким образом, мы можем не учитывать конструкцию `fresh` и рассматривать все подстановки с точностью до α-эквивалентности. В рамках дальнейших рассуждений это позволит нам исключить из состояний все операторы введения свежей переменной.

Лемма 13 (о пополнении подстановки при развертке)**.** Пусть r — применение отношения и θ — подстановка. Развёртка преобразует состояние $\langle\theta, r\rangle$ в набор состояний $\{\langle\theta_i, \rho_i\rangle\}$, где ρ_i конъюнкция применений, θ_i —некоторая подстановка. Тогда для каждого i существует последовательность унификаций $u_1^i \dots u_{k_i}^i$ такая, что выполнено следующее:

$$\theta \xrightarrow{u_1^i \dots u_{k_i}^i} \theta_i.$$

Эту лемму легко доказать индукцией по высоте вывода в семантике развертки.

Таким образом, мы можем ввести обозначение $[r \to \rho_i]$ для выполнения последовательности унификаций во время развёртки применения r, которое приводит к листовому состоянию ρ_i.

Лемма 14 (о сохранении сходимости ангелической семантики при уточнении)**.** Пусть θ — подстановка, ω — набор применений отношений, и пусть $\langle\theta, \omega\rangle$ сходится к $\{\}$ (пустому состоянию) в ангелической семантике. Тогда для произвольной подстановки σ и произвольного набора применений отношений ψ верно, что $\langle\theta \cdot \sigma, \omega\psi\rangle$ также сходится к $\{\}$.

Данную лемму легко доказать индукцией по длине вывода $\langle\theta, \omega\rangle \to^* \{\}$.

Эта лемма доказывает тот факт, что наложение дополнительных ограничений (более конкретная подстановка и добавление дополнительных конъюнктов) не может сделать сходящуюся программу расходящейся. Этот факт окажется важен при доказательстве следующей леммы.

Лемма 15 (о сохранении сходимости ангелической семантики)**.** Пусть $\varphi a \varphi'$ — набор применений отношений с выделенным элементом "a" и θ — подстановка. Пусть $\{\langle \theta \cdot [a \to \alpha_i], \alpha_i \rangle\}$ является результатом развёртки состояния $\langle \theta, a \rangle$. Тогда $\langle \theta, \varphi a \varphi' \rangle$ сходится тогда и только тогда, когда каждое состояние $\{\langle \theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi' \rangle\}$ сходится.

Доказательство.

$\Leftarrow$ Пусть все состояния из набора $\{\langle \theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi' \rangle\}$ сходятся. Тогда развёртка применения a в состоянии $\langle \theta, \varphi a \varphi' \rangle$ приводит нас к набору сходящихся по условию состояний $\{\langle \theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi' \rangle\}$.

$\Rightarrow$ Пусть $\langle \theta, \varphi a \varphi' \rangle$ сходится. Тогда возможны два случая.

Случай 1. При сходящемся выводе применение a не было развёрнуто. Тогда $\langle \theta, \varphi a \varphi' \rangle$ сходится к $\{\}$ (так как каждое промежуточное состояние содержит применение a). Более того, состояние $\langle \theta, \varphi \varphi' \rangle$ также сходится к $\{\}$ (мы можем повторить сходящийся вывод $\langle \theta, \varphi a \varphi' \rangle$ для состояния $\langle \theta, \varphi \varphi' \rangle$). По лемме 14 из сходимости вывода для состояния $\langle \theta, \varphi \varphi' \rangle$ следует, что все состояния $\langle \theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi' \rangle$ сходятся к $\{\}$.

Случай 2. При сходящемся выводе применение a было развёрнуто на некотором шаге. Тогда для состояния $\langle \theta, \varphi a \varphi' \rangle$ существует вывод

$$\langle \theta, \varphi a \varphi' \rangle \to^* \langle \theta \cdot \sigma, \pi a \rho \rangle \,, \qquad (*)$$

где σ является некоторым набором унификаций и следующим шагом этой последовательности будет развёртка применения a. Тогда результатом развёртки применения a будет набор состояний $\{\langle \theta \cdot \sigma \cdot [a \to \alpha_i], \pi \alpha_i \rho \rangle\}$. Повторим для состояний $\langle \theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi' \rangle$ развёртки из вывода $(*)$ получим состояние $\langle \theta \cdot [a \to \alpha_i] \cdot \sigma, \pi \alpha_i \rho \rangle$. Это состояние сходится, так как $\theta \cdot [a \to \alpha_i] \cdot \sigma = \theta \cdot \sigma \cdot [a \to \alpha_i]$.

□

Следствие 1. Пусть $\langle \theta, \omega \rangle$ — некоторое состояние, а $\{\langle \theta', \omega' \rangle\}$ — множество состояний, достижимых из состояния $\langle \theta, \omega \rangle$ за конечное число развёрток. Тогда

состояние $\langle \theta, \omega \rangle$ сходится тогда и только тогда, когда все состояния из множества $\{\langle \theta', \omega' \rangle\}$ сходятся.

5.3 Справедливость конъюнкции в семантике, оснащенной вполне квазиупорядочивающим предикатом выбора

Теперь у нас имеется всё необходимое для доказательства справедливости обобщенной семантики, оснащенной вполне квазиупорядочивающим предикатом выбора.

Теорема 3. Пусть $\mathcal{P}_{\trianglelefteq}$ — вполне квазиупорядочивающий предикат. Тогда "$\rightarrow_{\mathcal{P}_{\trianglelefteq}}$" является справедливой семантикой.

Доказательство. Пусть состояние g сходится в ангелической семантике. Рассмотрим применение отношения r, которое будет развёрнуто первым в сходящемся выводе обобщенной семантики, оснащенной вполне квазиупорядочивающим предикатом выбора. Возможны два случая.

Случай 1. Семантика "$\rightarrow_{\mathcal{P}_{\trianglelefteq}}$" выполнит в точности эту же развертку применения отношения r. В этом случае обе семантики выполнили идентичный шаг.

Случай 2. Семантика "$\rightarrow_{\mathcal{P}_{\trianglelefteq}}$" выбрала некоторое другое применение отношения. По свойству вполне квазипорядка будет выполнено конечное число разверток, прежде чем применение r будет развернуто. Повторим это конечное число разверток в справедливой семантики. По следствию 1 результат развёртки будет принадлежать множеству сходящихся состояний.

Таким образом семантика "$\rightarrow_{\mathcal{P}_{\trianglelefteq}}$" выполнит все шаги сходящегося вывода ангелической семантики и, следовательно, вывод семантики "$\rightarrow_{\mathcal{P}_{\trianglelefteq}}$" также сойдется.

$\square$

Данная теорема подтверждает тот факт, что при использовании вполне квазиупорядочевающего предиката в качестве параметра обобщенной семантики языка miniKanren мы получаем справедливую семантику.

Следствие 2. Семантика "$\rightarrow_{\mathcal{P}_{\leqslant_{sr}}}$" является справедливой.

Доказательство. Предикат $\leqslant_{sr}$ является вполне квазиупорядочевающим предикатом по лемме 11. Поэтому условие теоремы 3 выполнено для семантики "$\to_{\mathcal{P}_{\leqslant_{sr}}}$". □

Подведём итоги. Семантика, оснащенная вполне квазиупорядочивающим предикатом высоты структурно-рекурсивных аргументов, является справедливой и, как мы увидим в следующей главе, оказывается эффективной на практике.

Глава 6. Реализация и эксперименты

Данная глава посвящена программной реализации предложенного реляционного преобразования и справедливой семантики языка miniKanren, а также результатам экспериментов.

6.1 Реализация

Реализация реляционного преобразования выполнена в виде транслятора функциональных программ в реляционные. Реализация операционной семантики языка miniKanren, включая процедуру динамического управления порядком конъюнктов, выполнена в виде реляционного расширения языка OCaml. В качестве языка реализации был выбран OCaml версии 4.13 [99].

6.1.1 Транслятор функциональных программ в реляционные

В качестве исходного языка для транслируемых программ было выбрано подмножество языка OCaml, которое содержит все компоненты функционального языка, описанного в главе 2: λ-исчисление, рекурсивные и нерекурсивные `let`-привязки, конструкторы выражений алгебраических типов данных, оператор сопоставления с образцом, предопределенные логические константы `true` и `false` и полиморфное сравнение выражений первого порядка. В качестве целевого языка выбран OCanren [83], включающий все компоненты реляционного программирования, необходимые для трансляции.

Выбор языка OCaml в качестве языка реализации и исходного языка транслятора и OCanren (реляционного расширения языка OCaml) в качестве целевого языка обусловлен возможностью использования компонент инфраструктуры стандартного компилятора языка OCaml [68; 100] при разработке транслятора. Прежде всего использованы промежуточные представления программ для компилятора языка OCaml — *абстрактное синтаксическое дерево*

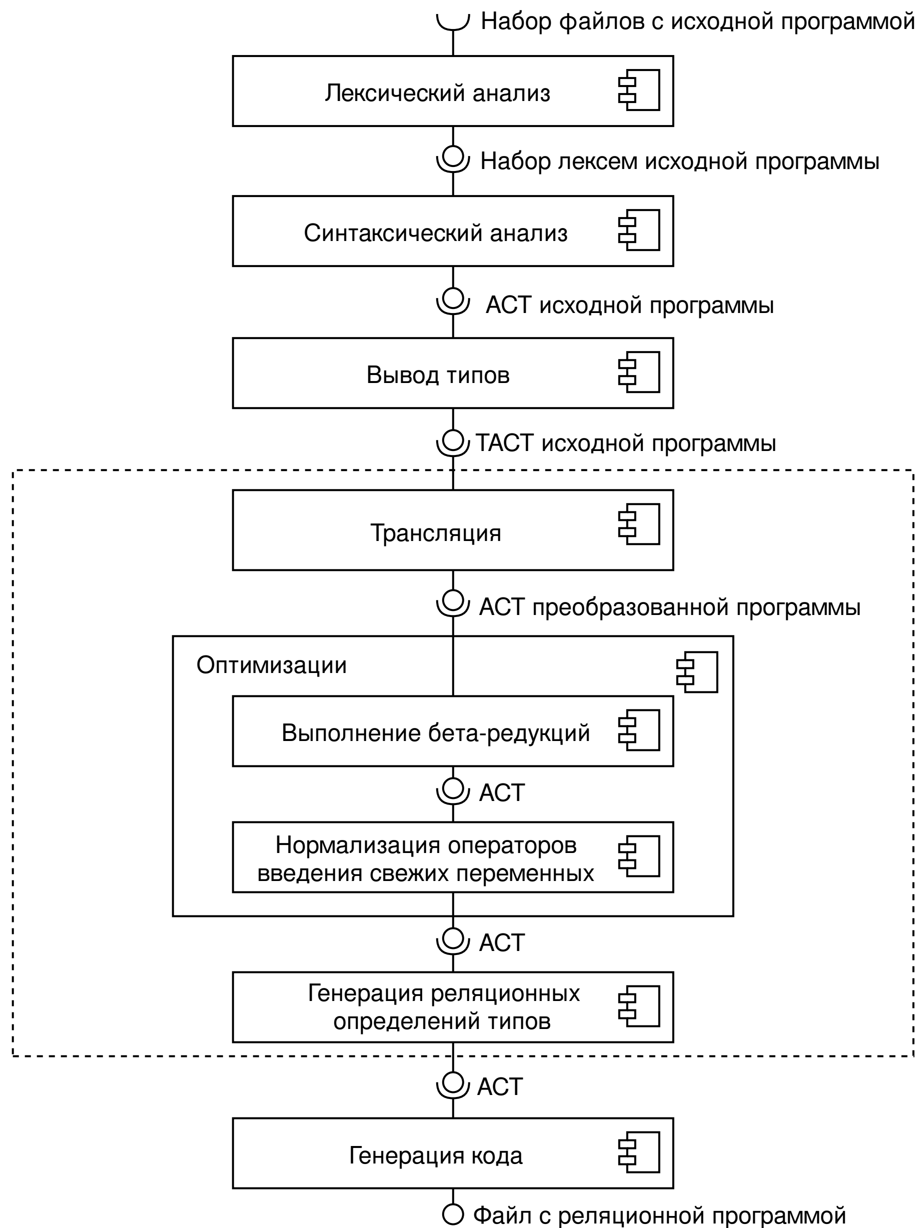


Рис. 6.1 — UML-диаграмма компонент транслятора функциональных программ в реляционные

программы (АСТ) и *типизированное абстрактное синтаксическое дерево* программы (ТАСТ). Оба представления описаны в виде набора алгебраических типов данных языка OCaml и являются классическим способом описания промежуточного представления программы. Также при разработке транслятора было использовано несколько готовых компонент компилятора OCaml: лексический анализ, синтаксический анализ, вывод типов и генерация кода.

Архитектура транслятора функциональных программ в реляционные представлена на рис. 6.1 в виде UML-диаграммы компонент, компоненты, разработанные автором данной работы выделены пунктирной линией. Отметим, что

архитектура транслятора имеет классический вид оптимизирующего транслятора [101] и содержит традиционные компоненты лексического и синтаксического анализа, вывода типов исходной программы, оптимизации и генерации кода.

Компонента *Лексический анализ* отвечает за выделение лексем языка в тексте исходной программы.

Компонента *Синтаксический анализ* формирует по набору лексем АСТ исходной программы.

Компонента *Вывод типов* вычисляет с помощью алгоритма Реми [102; 103] типы для всех всех выражений программы; эта компонента основана на алгоритме Хиндли-Милнера [81]. Результатом её работы является ТАСТ — представление программы, соответствующее АСТ, полученному на предыдущем этапе, в котором каждому выражению сопоставлен выведенный тип.

Компонента *Трансляция* является основной и выполняет построение реляционной программы по функциональной. Эта компонента реализует реляционное преобразование, представленное в главе 2. Благодаря тому, что целевой язык является расширением исходного, нет необходимости вводить ещё один вид промежуточного представления программы, и поэтому результат трансляции можно представить в виде АСТ для языка OCaml.

Компонента *Оптимизация* выполняет несколько эквивалентных преобразований АСТ для оптимизации итоговой программы с точки зрения её размера и эффективности. Компонента *Выполнения* β*-редукций*, которая входит в состав *Оптимизация*, отвечает за упрощение построенной программы посредством применения всех λ-абстракций к аргументам. Это необходимо вследствие возникновения во время трансляции множества λ-абстракций, которые могут быть применены статически. Данная компонента значительно улучшает качество результирующей программ как с точки зрения краткости и удобочитаемости, так и производительности. Компонента *Нормализацию операторов введения свежих переменных*, также являющаяся частью *Оптимизации*, нацелена на кластеризацию вводимых логических переменных с целью сокращения размера итоговой программы и накладных расходов, связанных с введением свежих переменных. Отметим, что транслятор содержит дополнительную компоненту оптимизации, ранжирующую конъюнкты в зависимости от оценки сложности их выполнения и степени недетерминизма, возникающего после их исполнения. К примеру операторы унификации и ограничения неравенством разумно переместить в начало набора конъюнктов, так как они детерминированы

и более просты в исполнении в сравнении с применениями отношений. Однако данная оптимизация необходима только в случае исполнения реляционной программы с использованием классической левоориентированной конъюнкции. В случае же исполнения с использованием справедливой конъюнкции, в данной компоненте оптимизации нет необходимости.

Компонента *Генерации реляционных определений типов* обобщает алгебраические типы данных исходной программы для корректного вывода типов итоговой реляционной программы.

Последняя компонента *Генерации кода* преобразовывает полученное в результате работы предыдущих компонент АСТ в реляционную программу, которую можно исполнить с использованием как классической лево-ориентированной конъюнкции, так и справедливой конъюнкции, а реализации которой пойдет речь в следующем разделе.

Общий объём реализации данного транслятора составил 1800 строк кода на языке OCaml, реализация доступна здесь [104].

6.1.2 Реляционное расширение языка OCaml, оснащенное вполне квазиупорядочевающим предикатом выбора

Для реализации реляционного расширения, оснащенного предикатом выбора, был выбран язык OCaml по следующим причинам. Во-первых, этот язык позволяет использовать представление реляционных термов, операторы унификации и ограничения неравенством, реализованные в существующем реляционном расширении с лево-ориентированной конъюнкцией OCanren. Во-вторых, для интеграции реляционного расширения и транслятора, описанного в предыдущем разделе, необходимо соответствие целевого языка транслятора и реализуемого расширения. По этой же причине синтаксис реляционного расширения, оснащенного предикатом выбора, совпадает с синтаксисом расширения OCanren.

Встраивание реляционного расширения в язык OCaml было выполнено следующим образом. Структура состояния реляционной программы была описана в виде алгебраического типа данных. Далее, все операторы расширения были выражены средствами исходного языка OCaml в виде функций высшего

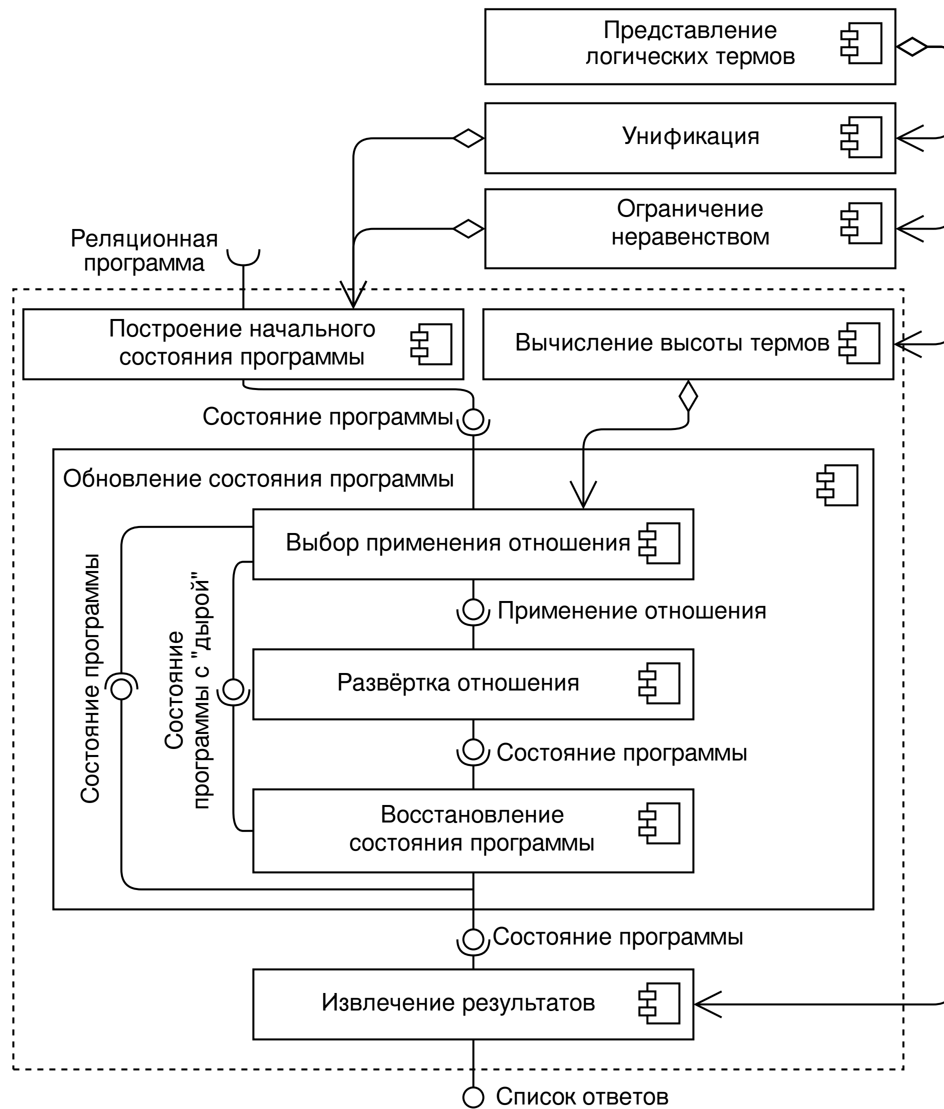


Рис. 6.2 — UML-диаграмма компонент реляционного расширения, оснащенного вполне квазиупорядочевающим предикатом выбора

порядка. Таким образом любая реляционная программа является комбинацией реляционных операторов и, как следствие, функцией языка OCaml. Так как результатом реляционной программы является потенциально бесконечный набор ответов, то для извлечения части ответов была разработана функция `run`, позволяющая извлечь конечную часть ответов в виде списка.

Архитектура реляционного расширения представлена на рис. 6.2 в виде UML-диаграммы компонент. Компоненты, разработанные автором данной работы выделены пунктирной линией.

Прежде всего, рассмотрим вспомогательные компоненты, являющиеся частью реляционного расширения OCanren. Компонента *Представление логических термов* отвечает за построение и корректную типизацию логических

термов, содержащих как конструкторы, так и логические переменные. Подробности данного представления описаны в работе [67]. Данное представление используется во всех компонентах, обрабатывающих термы, а именно в компоненте унификации, компоненте ограничения неравенством, компоненте вычисления высоты термов и компоненте извлечения результатов.

Компоненты *Унификация* и *Ограничение неравенством* отвечают за выполнение реляционных операторов унификации и ограничения неравенством соответственно. Принцип их работы подробно описан в разделе 2.2.

Компонента *Построение начального состояния программы* содержит реализации всех операторов реляционного расширения. Благодаря данной компоненте этап интерпретации реляционных программ исключен из реализации, так как все реляционные операторы являются функциями языка OCaml. В качестве начального состояния выступает реляционный запрос, содержащий переменные запроса — переменные, итоговые значения которых будут результатом выполнения программы. Также данная компонента содержит алгоритм автоматического обнаружения структурно-рекурсивных аргументов отношений. В случае отсутствия таких аргументов, производится обнаружение аргумента,структурно убывающего в максимальном количестве дизъюнктов отношения. Наконец, в данной компоненте разработчиком определяется максимальное количество результатов, которое необходимо вычислить. Если число не задано, исполнение будет продолжаться до полного исчерпания состояния программы.

Основной компонентой преобразования является *Обновление состояния программы*, которая отвечает за пошаговое перестроение состояния посредством вычисления отдельных частей состояния. Она включает в себя компоненту *Выбор применения отношения*, которая определяет лист состояния программы, необходимый для обновления, выбирает с помощью предиката применение отношения и отделяет его от состояния. Результатом работы данного компонента является отделенное применение и состояние программы с “дырой” на месте этого применения. Предикат выбора основан на сравнении высот аргументов, за который отвечает компонента *Вычисления высот термов*, опирающаяся в свою очередь на представление логических термов. Компонента *Развёртка отношения* выполняет замену применения отношения на тело отношение и подставляет аргументы в тело отношения. Результатом развёртки является состояние программы, которое необходимо подставить на место раз-

вёрнутого применения отношения. За это отвечает компонента *Восстановления состояния программы*. Далее возможны два случая. Если состояние программы содержит необходимое число листьев с результирующими подстановками, то процесс обновления завершается. В противном случае необходимо выполнить обновление нового состояния программы ещё раз.

Последняя компонента *Извлечение результатов* выбирает из результирующих подстановок термы, соответствующие переменным исходного реляционного запроса. Для извлечения результатов используются функция уточнения термов `walk`, содержащаяся в компоненте представления логических термов.

Общий объём реализации реляционного расширения, оснащенного предикатом выбора, составил 1200 строк кода, реализация доступна здесь [105].

6.2 Эксперименты

Корректность предложенного реляционного преобразования и семантики реляционного языка miniKanren формально доказана в главах 3 и 5 соответственно. Однако их производительность требует экспериментальной оценки. В случае реляционного преобразования необходимо оценить эффективность преобразованных программ. Для этого мы оптимизируем преобразованную программу вручную и сравним производительность оптимизированной и неоптимизированной версии. В случае семантики, реализованной в виде расширения языка OCaml, необходимо сравнить эффективность справедливой конъюнкции в сравнении с классической конъюнкцией в существующей реализацией OCanren.

Таким образом, *целью* данного экспериментального исследования является сравнение эффективности справедливой и классической конъюнкции при исполнении оптимизированных вручную и неоптимизированных преобразованных программ.

Для достижения данной цели необходимо ответить на следующие *вопросы*.

[Q1] Каковы накладные расходы справедливой конъюнкции?

[Q2] Какова производительность справедливой конъюнкции по сравнению с классической?

[Q3] Какова стабильность справедливой конъюнкции по сравнению с классической?

[Q4] Какова производительность неоптимизированных программ по сравнению с оптимизированными при использовании справедливой конъюнкции?

Для получения ответов на поставленные вопросы необходимо вычислить две следующие *метрики*:

[M1] время исполнения набора оптимизированных программ с использованием классической и справедливой конъюнкции;

[M2] время исполнения набора неоптимизированных программ с использованием классической и справедливой конъюнкции.

Для экспериментов были выбраны две простые программы, которые исполнялись как в прямом так и в обратном направлении: отношение разворота списка `revers`^{o} и отношение сортировки списка `sort`^{o}. Также были выбраны три более сложные программы: первая `hanoi`^{o} решает головоломку "Ханойские башни"[1], вторая `bridge`^{o} решает головоломку "о мосте и факеле"[2], и третья программа `water`^{o} решает головоломку "с переливанием воды"[3]. Все эти программы были реализованы на подмножестве функционального языка OCaml для их последующего реляционного преобразования. После чего были получены реляционные образы данных программ и затем оптимизированы вручную посредством переупорядочивания конъюнктов.

И оптимизированные, и неоптимизированные реляционные программы были скомпилированы как с использованием расширения OCanren, так и с использованием разработанного реляционного расширения. Для уменьшения погрешности при измерении времени работы, каждая программа запускалась 20 раз с целью вычисления среднего времени её работы.

В таблице 1 в соответствии с метрикой **[M1]** показаны результаты выполнения оптимизированных вручную программ с использованием классической и справедливой конъюнкции. Как можно видеть, время исполнения программ с использованием классической конъюнкции незначительно меньше по сравнению с временем исполнения программ с использованием справедливой конъюнкции. Так как данные программы оптимизированы для конкретного

[1] https://en.wikipedia.org/wiki/Tower_of_Hanoi

[2] https://en.wikipedia.org/wiki/Bridge_and_torch_problem

[3] https://en.wikipedia.org/wiki/Water_pouring_puzzle

Программа	Размер входа	Классическая конъюнкция	Справедливая конъюнкция
revers^o в прямом направлении	30	0.002	0.002
	60	0.012	0.012
	90	0.044	0.045
revers^o в обратном направлении	30	0.002	0.002
	60	0.014	0.014
	90	0.054	0.055
sort^o в прямом направлении	30	0.031	0.032
	60	0.357	0.362
	30	1.324	1.388
sort^o в обратном направлении	3	0.001	0.001
	4	0.001	0.001
	5	0.012	0.012
	6	0.150	0.158
hanoi^o	-	0.597	0.620
bridge^o	-	0.074	0.075
water^o	-	1.247	1.317

Таблица 1 — Время исполнения (в секундах) набора оптимизированных вручную программ с использованием классической и справедливой конъюнкции

запроса с помощью подбора оптимального порядка конъюнктов, справедливая конъюнкция не приводит к приросту производительности, однако её использование добавляет незначительные накладные расходы, которые составляют менее 6% от общего времени исполнения. Таким образом, ответ на вопрос **[Q1]** выглядит так: накладные расходы незначительны и не превосходят 6% от общего времени исполнения.

В таблице 2 в соответствии с метрикой **[M2]** показаны результаты выполнения неоптимизированных программ с использованием классической и справедливой конъюнкции. В случае использования классической конъюнкции при неоптимальном порядке конъюнктов исполняемая программа имеет экспоненциальную асимптотическую сложность по времени. В этом случае исполнение программы было ограничено 300 секундами, поэтому точное время их исполнения неизвестно. В таблице этому случаю соответствует обозначение “>300”. В данном случае исполняемые программы получены автоматически и

Программа	Размер входа	Классическая конъюнкция	Справедливая конъюнкция
$revers^o$ в прямом направлении	30	0.002	0.002
	60	0.012	0.012
	90	0.044	0.046
$revers^o$ в обратном направлении	30	0.016	0.003
	60	0.176	0.015
	90	0.860	0.060
$sort^o$ в прямом направлении	30	0.034	0.035
	60	0.363	0.377
	90	1.350	1.417
$sort^o$ в обратном направлении	3	0.001	0.001
	4	6.785	0.001
	5	>300	0.013
	6	>300	0.161
$hanoi^o$	-	>300	0.644
$bridge^o$	-	34.165	0.075
$water^o$	-	>300	1.326

Таблица 2 — Время исполнения (в секундах) набора неоптимизированных программ с использованием классической и справедливой конъюнкции

не оптимизированы для конкретного запроса, поэтому время исполнения программ с использованием классической конъюнкции значительно выше, чем время исполнения программ с использованием справедливой конъюнкции. В лучшем случае для неоптимизированных программ производительность справедливой конъюнкции превосходит классическую более чем в 450 раз. Таким образом, ответ на вопрос **[Q2]** выглядит так: в случае неоптимизированных реляционных программ производительность справедливой конъюнкции выше в сравнении с классической, и в пиковом случае превосходит классическую конъюнкцию более чем в 450 раз.

Для получения ответа на вопросы **[Q3]** и **[Q4]** сравним показатели времени справедливой конъюнкции для неоптимизированных и оптимизированных программ. Как видно из таблиц 1 и 2, время исполнения программ с использованием справедливой конъюнкции не имеет значительных различий и позволяет ответить на вопрос **[Q3]** следующим образом: справедливая конъюнкция де-

монстрирует высокую стабильность и не зависит от порядка конъюнктов в программе. Одновременно с этим мы можем ответить и на вопрос **[Q4]** так: при использовании справедливой конъюнкции производительность неоптимизированных программ соответствует производительности оптимизированных программ.

Подводя итог, сделаем следующий вывод: справедливая конъюнкция, основанная на структурной рекурсии, сопоставима по эффективности с классической направленной конъюнкцией в случае оптимизированных программ, а также демонстрирует более высокую производительность в случае неоптимизированных программ. В то же время при использовании справедливой конъюнкции, производительность неоптимизированных программ сопоставима с производительностью оптимизированных вручную программ.

Заключение

В качестве итогов сформулируем результаты, достигнутые в ходе выполнения данного исследования.

1. Предложен новый метод реляционного преобразования функциональных программ общего вида, доказана его статическая и динамическая корректность.
2. Введена формальная ангелическая семантика реляционного языка miniKanren, доказана эквивалентность этой семантики декларативной семантикой языка miniKanren. Введено понятие справедливости конъюнкции как свойства ангелической семантики.
3. Определена формальная семантика реляционного языка miniKanren с процедурой динамического управления порядком вычисления конъюнктов, доказана справедливость конъюнкции в данной семантике.
4. Выполнена реализация на языке OCaml реляционного преобразования для подмножества функционального языка OCaml, а также проведено экспериментальное исследование, показавшее высокую эффективность автоматически полученных реляционных программ при использовании справедливой конъюнкции.
5. Выполнено встраивание языка miniKanren с процедурой динамического управления порядком конъюнктов в функциональный язык OCaml и проведено экспериментальное исследование, демонстрирующее высокую эффективность процедуры в сравнении с классической реализацией miniKanren при неоптимальном порядке конъюнктов и незначительность накладных при оптимальном порядке конъюнктов.

В качестве **рекомендаций для использования полученных результатов** в промышленности и научных исследованиях укажем следующее. На текущий момент разработка реляционных программ является сложной задачей, требующей от разработчиков глубоких познаний в данной области. В результате реляционное программирование используется преимущественно в научных исследованиях отдельными экспертами для решения узкого класса задач поиска решений, валидации и синтеза программ [38; 46; 60]. Однако, благодаря предложенным в данной работе методам реляционное программирование становится доступнее и, как следствие, привлекательнее для более широкого круга поль-

зователей. Кроме того автоматизированное построение реляционных программ по функциональным позволит разрабатывать комплексные целевые решения, совмещающие функциональное и реляционное фрагменты в рамках инфраструктуры одного языка.

В качестве **перспектив дальнейшей разработки тематики** может быть отмечено следующее:

- дальнейшее исследование особенностей справедливого исполнения реляционной конъюнкции, включая поиск эффективных предикатов выбора для реляционных программ специального вида, изучение предикатов выбора помимо вполне квазиупорядочивающих предикатов, рассмотренных в данном диссертационном исследовании;
- обобщение свойства справедливости для реляционной дизъюнкции;
- расширение исходного языка для реляционного преобразования, в частности, добавление оператора сопоставления с образцом общего вида.

Список литературы

1. *Prawitz, D.* An Improved Proof Procedure [Текст] / D. Prawitz // Theoria. — 1960. — Vol. 26, no. 2. — P. 102—139.
2. *Gilmore, P. C.* A Proof Method for Quantification Theory: Its Justification and Realization [Текст] / P. C. Gilmore // IBM Journal of Research and Development. — 1960. — Vol. 4, no. 1. — P. 28—35.
3. *Davis, M.* A Computing Procedure for Quantification Theory [Текст] / M. Davis, H. Putnam // J. ACM. — New York, NY, USA, 1960. — Vol. 7, no. 3. — P. 201—215.
4. *Hewitt, C.* PLANNER: A Language for Proving Theorems in Robots [Текст] / C. Hewitt // Proceedings of the 1st International Joint Conference on Artificial Intelligence. — Washington, DC : Morgan Kaufmann Publishers Inc., 1969. — P. 295—301.
5. *Kowalski, R.* Predicate Logic as a Programming Language [Текст] / R. Kowalski // Information Processing. — Stockholm, North Holland, 1974. — P. 569—574.
6. Un système de communication homme-machine en Français [Текст] / A. Colmerauer [et al.] // Groupe de recherche en Intelligence Artificielle. — Marseille, France, 1973.
7. Information Technology — Programming Languages — Prolog — Part 1: General Core [Текст] : Standard / International Organization for Standardization. — 1995.
8. Information Technology — Programming Languages — Prolog — Part 2: Modules [Текст] : Standard / International Organization for Standardization. — Geneva, Switzerland, 2000.
9. *Colmerauer, A.* The Birth of Prolog [Текст] / A. Colmerauer, P. Roussel. — 1996.
10. *Kowalski, R.* Algorithm = Logic + Control [Текст] / R. Kowalski // Commun. ACM. — New York, NY, USA, 1979. — Vol. 22, no. 7. — P. 424—436.

11. *Merritt, D.* Building Expert Systems in Prolog [Текст] / D. Merritt. — Berlin, Heidelberg : Springer-Verlag, 1989.

12. *Beckert, B.* leanTAP: Lean Tableau-based Deduction [Текст] / B. Beckert, J. Posegga // Journal of Automated Reasoning. — 1995. — Vol. 15. — P. 339—358.

13. *Clocksin, W. F.* Clause and Effect: Prolog Programming for the Working Programmer [Текст] / W. F. Clocksin. — Secaucus, New Jersey, USA : Springer, 1997.

14. *Weijland, W.* Semantics for Logic Programs without Occur Check [Текст] / W. Weijland // Theoretical Computer Science. — 1990. — Vol. 71, no. 1. — P. 155—174.

15. *Knuth, D. E.* The Art of Computer Programming: Combinatorial Algorithms, Part 1 [Текст] / D. E. Knuth. — 3rd. — Addison-Wesley Professional, 1997.

16. *Arbab, B.* Operational and Denotational Semantics of Prolog [Текст] / B. Arbab, D. M. Berry // The Journal of Logic Programming. — 1987. — Vol. 4, no. 4. — P. 309—329.

17. *Stroetmann, K.* A Declarative Semantics for the Prolog Cut Operator [Текст] / K. Stroetmann, T. Glaß // Proceedings of the 5th International Workshop on Extensions of Logic Programming. — Berlin, Heidelberg : Springer-Verlag, 1996. — P. 255—271.

18. *Ceri, S.* What You Always Wanted to Know About Datalog (And Never Dared to Ask) [Текст] / S. Ceri, G. Gottlob, L. Tanca // IEEE Transactions on Knowledge and Data Engineering. — 1989. — T. 1, № 1. — C. 146—166.

19. *Cheney, J.* Nominal Logic Programming [Текст] : дис. ... канд. / Cheney J. — Valencia, Spain : Cornell University, 08.2004.

20. *Cheney, J.* αProlog: A Logic Programming Language with Names, Binding, and α-Equivalence [Текст] / J. Cheney, C. Urban // Proceedings of the 20th Intl. Conf. on Logic Programming. — Saint-Malo, France, 2004. — Март.

21. *Nadathur, G.* An Overview of λprolog [Текст] / G. Nadathur, D. Miller // Logic Programming, Proceedings of the Fifth International Conference and Symposium / под ред. R. Kowalski, K. Bowen. — Seattle, Washington, USA : MIT Press, 1988. — C. 810—827.

22. *Nadathur, G.* The Metalanguage λprolog and Its Implementation [Текст] / G. Nadathur // Proceedings of the 5th International Symposium on Functional and Logic Programming. — Berlin, Heidelberg : Springer-Verlag, 2001. — С. 1—20.

23. *Hanus, M.* Curry: A Truly Functional Logic Language [Текст] / M. Hanus, H. Kuchen, J. J. Moreno-Navarro //. — 1995.

24. *Hanus, M.* Search Strategies for Functional Logic Programming [Текст] / M. Hanus, B. Peemöller, F. Reck // Software Engineering 2012. Workshopband / ed. by S. Jähnichen, B. Rumpe, H. Schlingloff. — Bonn : Gesellschaft für Informatik e.V., 2012. — P. 61—74.

25. *Hanus, M.* A Generic Analysis Environment for Declarative Programs [Текст] / M. Hanus // Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming. — Tallinn, Estonia : Association for Computing Machinery, 2005. — P. 43—48.

26. *Hanus, M.* A Typeful Integration of SQL into Curry [Текст] / M. Hanus, J. Krone // Electronic Proceedings in Theoretical Computer Science. — 2016. — Dec. — Vol. 234. — P. 104—119.

27. *Hanus, M.* Declarative Programming of User Interfaces [Текст] / M. Hanus, C. Kluß //. — 01/2009. — P. 16—30.

28. *Somogyi, Z.* The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language [Текст] / Z. Somogyi, F. Henderson, T. Conway // The Journal of Logic Programming. — 1996. — Vol. 29, no. 1. — P. 17—64. — High-Performance Implementations of Logic Programming Systems.

29. *Overton, D.* Precise and Expressive Mode Systems for Typed Logic Programming Languages [Текст] : PhD thesis / Overton David. — Department of Computer Science, Software Engineering, The University of Melbourne, 2004.

30. *Hanus, M.* Ontology Driven Software Engineering for Real Life Applications [Текст] / M. Hanus, J. Krone //. — Innsbruck, Austria, 2007.

31. *Hill, P.* The Gödel Programming Language [Текст] / P. Hill, J. W. Lloyd. — The MIT Press, 1994.

32. *Pfenning, F.* System Description: Twelf — A Meta-Logical Framework for Deductive Systems [Текст] / F. Pfenning, C. Schürmann // Proceedings of 16th International Conference on Automated Deduction. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1999. — P. 202—206.

33. *Crary, K.* Higher-Order Representation of Substructural Logics [Текст] / K. Crary // Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. — Baltimore, Maryland, USA : Association for Computing Machinery, 2010. — P. 131—142.

34. *Lee, D. K.* Towards a Mechanized Metatheory of Standard ML [Текст] / D. K. Lee, K. Crary, R. Harper // Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — Nice, France : Association for Computing Machinery, 2007. — P. 173—184.

35. The Reasoned Schemer [Текст] / D. P. Friedman [et al.]. — 2nd. — The MIT Press, 2005.

36. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl) [Текст] / O. Kiselyov [et al.] // SIGPLAN Not. — New York, NY, USA, 2005. — Vol. 40, no. 9. — P. 192—203.

37. *Rozplokhas, D.* Scheduling Complexity of Interleaving Search [Текст] / D. Rozplokhas, D. Boulytchev // Proceedings of the 16th International Symposium on Functional and Logic Programming. — 2022.

38. *Byrd, W. E.* MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl) [Текст] / W. E. Byrd, E. Holk, D. P. Friedman // Proceedings of the Annual Workshop on Scheme and Functional Programming. — Copenhagen, Denmark : Association for Computing Machinery, 2012. — P. 8—29.

39. *Near, J. P.* αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic [Текст] / J. P. Near, W. E. Byrd, D. P. Friedman // Logic Programming / ed. by M. Garcia de la Banda, E. Pontelli. — Berlin, Heidelberg, 2008. — P. 238—252.

40. *Byrd, W. E.* Relational Programming in miniKanren: Techniques, Applications, and Implementations [Текст] : PhD thesis / Byrd William E. — Indiana University, 09/2009.

41. *Hemann, J.* μKanren: A Minimal Functional Core for Relational Programming [Текст] / J. Hemann, D. P. Friedman // Proceedings of the 2013 Workshop on Scheme and Functional Programming. — Alexandria, VA, 2013.

42. *Swords, C.* rKanren: Guided Search in miniKanren [Текст] / C. Swords, D. P. Friedman // In Proceedings of the 2013 Workshop on Scheme and Functional Programming. — Alexandria, VA, USA, 2013.

43. *Hemann, J.* A Framework for Extending microKanren with Constraints [Текст] / J. Hemann, D. P. Friedman // In Proceedings of the 2015 Workshop on Scheme and Functional Programming. — 2015.

44. *Byrd, W. E.* αKanren A Fresh Name in Nominal Logic Programming [Текст] / W. E. Byrd, D. P. Friedman // In Proceedings of the 2007 Workshop on Scheme and Functional Programming. — 2007. — P. 79—90.

45. *Moiseenko, E.* Constructive Negation for miniKanren [Текст] / E. Moiseenko // Proceedings of the 2019 miniKanren and Relational Programming Workshop. — 2019. — P. 58—78.

46. A Unified Approach to Solving Seven Programming Problems (Functional Pearl) [Текст] / W. E. Byrd [et al.] // Proceedings of ACM Program. Lang. — New York, NY, USA, 2017. — 8:1—8:26.

47. Guarded Fresh Goals: Dependency-Directed Introduction of Fresh Logic Variables [Текст] / C. E. Alvis [et al.] // Proceedings of the 2021 miniKanren and Relational Programming Workshop. — 2021.

48. *Lu, K.-C.* Towards a miniKanren with Fair Search Strategies [Текст] / K.-C. Lu, W. Ma, D. P. Friedman // Proceedings of the 2019 miniKanren and Relational Programming Workshop. — 2019. — P. 1—15.

49. *Rozplokhas, D.* Improving Refutational Completeness of Relational Search via Divergence Test [Текст] / D. Rozplokhas, D. Boulytchev // Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming. — New York, NY, USA : Association for Computing Machinery, 2018. — 18:1—18:13.

50. *Schrijvers, T.* Tor: Extensible Search with Hookable Disjunction [Текст] / T. Schrijvers, M. Triska, B. Demoen // Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. — Leuven, Belgium : Association for Computing Machinery, 2012. — P. 103—114.

51. *S.M.A., P.* Tabling in contextual abduction with answer subsumption [Текст] / P. S.M.A., S. A., P. L.M. // 2017 International Conference on Advanced Computer Science and Information Systems, ICACSIS 2017. — 2018. — P. 459—464.

52. *J., A.* Evaluation of the Implementation of an Abstract Interpretation Algorithm using Tabled CLP [Текст] / A. J., C. M. // Theory and Practice of Logic Programming. — 2019. — Vol. 19. — P. 1107—1123.

53. *Tamaki, H.* OLD Resolution with Tabulation [Текст] / H. Tamaki, T. Sato // Proceedings of the Third International Conference on Logic Programming. — 07/1986. — P. 84—98.

54. *Naish, L.* Automating Control for Logic Programs [Текст] / L. Naish // J. Log. Program. — 1985. — Vol. 2. — P. 167—183.

55. *Lloyd, J. W.* Foundations of Logic Programming [Текст] / J. W. Lloyd. — Berlin, Heidelberg : Springer-Verlag, 1984.

56. Parallel Execution of Prolog Programs: A Survey [Текст] / G. Gupta [et al.] // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 2001. — Vol. 23, no. 4. — P. 472—602.

57. *Lozov, P.* Typed Relational Conversion [Текст] / P. Lozov, A. Vyatkin, D. Boulytchev // Trends in Functional Programming. — Springer International Publishing, 2018. — P. 39—58.

58. *Лозов, П.* Преобразование типизированных функций в реляционную форму [Текст] / П. Лозов, Д. Булычев // Труды Института системного программирования РАН. — 2018. — Т. 30, № 2. — С. 45—64.

59. *Minsky, Y.* Real World OCaml: Functional Programming for the Masses [Текст] / Y. Minsky, A. Madhavapeddy, J. Hickey. — O'Reilly Media, 2013.

60. *Lozov, P.* Relational Interpreters for Search Problems [Текст] / P. Lozov, E. Verbitskaia, D. Boulytchev // Proceedings of the 2019 miniKanren and Relational Programming Workshop. — 2019. — P. 43—57.

61. *Lozov, P.* On Fair Relational Conjunction [Текст] / P. Lozov, D. Boulytchev // Proceedings of the 2020 miniKanren and Relational Programming Workshop. — 2020. — P. 1—12.

62. *Lozov, P.* Efficient Fair Conjunction for Structurally-Recursive Relations [Текст] / P. Lozov, D. Boulytchev // Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. — Association for Computing Machinery, 2021. — P. 58—73.

63. *Kosarev, D.* Relational Synthesis for Pattern Matching [Текст] / D. Kosarev, P. Lozov, D. Boulytchev // Programming Languages and Systems / ed. by B. C. d. S. Oliveira. — Springer International Publishing, 2020. — P. 293—310.

64. *Wright, A.* A Syntactic Approach to Type Soundness [Текст] / A. Wright, M. Felleisen // Inf. Comput. — Duluth, MN, USA, 1994. — Vol. 115, no. 1. — P. 38—94.

65. *Barendregt, H. P.* Handbook of Logic in Computer Science (Vol. 2) [Текст] / H. P. Barendregt ; ed. by S. Abramsky, D. M. Gabbay, S. E. Maibaum. — Oxford University Press, Inc., 1992. — Chap. Lambda Calculi with Types. P. 117—309.

66. *Urzyczyn, P.* Inhabitation in Typed Lambda-Calculi (A Syntactic Approach) [Текст] / P. Urzyczyn // Proceedings of the Third International Conference on Typed Lambda Calculi and Applications. — Berlin, Heidelberg : Springer-Verlag, 1997. — P. 373—389.

67. *Kosarev, D.* Typed Embedding of a Relational Language in OCaml [Текст] / D. Kosarev, D. Boulytchev // Proceedings Electronic Proceedings in Theoretical Computer Science. Vol. 285 / ed. by K. Asai, M. R. Shinwell. — Nara, Japan, 2016. — P. 1—22.

68. *Rémy, D.* Objective ML: A Simple Object-Oriented Extension of ML. [Текст] / D. Rémy, J. Vouillon // Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1997. — P. 40—53.

69. A Small Embedding of Logic Programming with a Simple Complete Search [Текст] / J. Hemann [et al.] // SIGPLAN Not. — New York, NY, USA, 2016. — Vol. 52, no. 2. — P. 96—107.

70. cKanren: miniKanren with Constraints [Текст] / C. E. Alvis [et al.] // Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming. — 2011.

71. *Baader, F.* Handbook of Automated Reasoning [Текст] / F. Baader, W. Snyder ; ed. by A. Robinson, A. Voronkov. — Amsterdam, The Netherlands, The Netherlands : Elsevier Science Publishers B. V., 2001. — Chap. Unification Theory.

72. *Cardelli, L.* On Understanding Types, Data Abstraction, and Polymorphism [Текст] / L. Cardelli, P. Wegner // ACM Comput. Surv. — New York, NY, USA, 1985. — Vol. 17, no. 4. — P. 471—523.

73. *Schmidt, D. A.* Denotational Semantics: A Methodology for Language Development [Текст] / D. A. Schmidt. — USA : William C. Brown Publishers, 1986.

74. *Winskel, G.* The Formal Semantics of Programming Languages: An Introduction [Текст] / G. Winskel. — MIT Press, 1993.

75. *Fernández, M.* Programming Languages and Operational Semantics: An Introduction [Текст] / M. Fernández. — King's College Publications, 2004.

76. A Simple Applicative Language: Mini-ML [Текст] / D. Clément [et al.] // Proceedings of the 1986 ACM Conference on LISP and Functional Programming. — New York, NY, USA : Association for Computing Machinery, 1986. — P. 13—27.

77. *Plotkin, G.* Call-by-name, call-by-value and the λ-calculus [Текст] / G. Plotkin // Theoretical Computer Science. — 1975. — Vol. 1, no. 2. — P. 125—159.

78. *Plotkin, G.* A Structural Approach to Operational Semantics [Текст] : tech. rep. / G. Plotkin. — University of Aarhus, 1981. — DAIMI FN—19.

79. *Felleisen, M.* The Calculi of lambda-nu-cs Conversion: a Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages [Текст] : PhD thesis / Felleisen Matthias. — Indiana University, 1987.

80. Oficial miniKanren web-page [Текст]. — URL: http://minikanren.org (visited on 06/01/2022).

81. *Hindley, R.* The Principal Type-Scheme of an Object in Combinatory Logic [Текст] / R. Hindley // Transactions of the American Mathematical Society. — 1969. — Vol. 146. — P. 29—60.

82. *Milner*, *R.* A Theory of Type Polymorphism in Programming [Текст] / R. Milner // Journal of Computer and System Sciences. — 1978. — Vol. 17, no. 3. — P. 348—375.

83. OCanren web-page [Текст]. — URL: https://github.com/JetBrains-Research/OCanren (visited on 06/01/2022).

84. *Lassez*, *J.-L.* Foundations of Deductive Databases and Logic Programming [Текст] / J.-L. Lassez, M. J. Maher, K. Marriott ; ed. by J. Minker. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1988. — Chap. Unification Revisited. P. 587—625.

85. *Pierce*, *B.* Types and Programming Languages [Текст] / B. Pierce. — MIT Press, 2002. — Chap. 8.3 Safety = Progress + Preservation.

86. *Hodges*, *W.* Formal Features of Compositionality [Текст] / W. Hodges // Journal of Logic, Language, and Information. — 2001. — Vol. 10, no. 1. — P. 7—28.

87. *Church*, *A.* Some Properties of Conversion [Текст] / A. Church, J. B. Rosser // Transactions of the American Mathematical Society. — 1936. — Vol. 39, no. 3. — P. 472—482.

88. *N. A. Lynch*, *F. W. V.* Forward and Backward Simulations: I. Untimed Systems [Текст] / F. W. V. N. A. Lynch // Information and Computation. — 1995. — Vol. 121. — P. 214—233.

89. *N. A. Lynch*, *F. W. V.* Forward and Backward Simulations, II: Timing-Based Systems [Текст] / F. W. V. N. A. Lynch // Information and Computation. — 1996. — Vol. 128. — P. 1—25.

90. *Rozplokhas*, *D.* Certified Semantics for Relational Programming [Текст] / D. Rozplokhas, A. Vyatkin, D. Boulytchev // Programming Languages and Systems / ed. by B. C. d. S. Oliveira. — Springer International Publishing, 2020. — P. 167—185.

91. *Keller*, *R. M.* Formal Verification of Parallel Programs [Текст] / R. M. Keller // Commun. ACM. — 1976. — Vol. 19, no. 7. — P. 371—384.

92. *Bertot*, *Y.* Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions [Текст] / Y. Bertot, P. Castéran. — Springer, 2004. — (Texts in Theoretical Computer Science. An EATCS Series).

93. *Turchin, V. F.* The Concept of a Supercompiler [Текст] / V. F. Turchin // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 1986. — Vol. 8, no. 3. — P. 292—325.

94. *Sørensen, M. H.* An Algorithm of Generalization in Positive Supercompilation [Текст] / M. H. Sørensen, R. Glück // Proceedings of ILPS'95, the International Logic Programming Symposium. — MIT Press, 1995. — P. 465—479.

95. *Kruskal, J. B.* Well-Quasi-Ordering, The Tree Theorem, and Vazsonyi's Conjecture [Текст] / J. B. Kruskal // Transactions of the American Mathematical Society. — 1960. — Vol. 95, no. 2. — P. 210—225.

96. *Higman, G.* Ordering by Divisibility in Abstract Algebras [Текст] / G. Higman // Proceedings of the London Mathematical Society. — 1952. — No. 1. — P. 326—336.

97. *Friedman, D. P.* Fancy Ferns Require Little Care [Текст] / D. P. Friedman, D. S. Wise // Symposium on Functional Languages and Computer Architecture. — 1981. — P. 124—156.

98. *Leuschel, M.* On the Power of Homeomorphic Embedding for Online Termination [Текст] / M. Leuschel // Static Analysis / ed. by G. Levi. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. — P. 230—245.

99. OCaml 4.13.0 Release Notes [Текст]. — URL: https://ocaml.org/releases/4.13.0 (visited on 06/01/2022).

100. The core OCaml system web-page [Текст]. — URL: https://github.com/ocaml/ocaml (visited on 06/01/2022).

101. *Aho, A. V.* The Theory of Parsing, Translation, and Compiling [Текст] / A. V. Aho, J. D. Ullman. — USA : Prentice-Hall, Inc., 1972.

102. *Wright, A. K.* A Syntactic Approach to Type Soundness [Текст] / A. K. Wright, M. Felleisen // Inf. Comput. — 1994. — Vol. 115. — P. 38—94.

103. *Rémy, D.* Using, Understanding, and Unraveling the OCaml Language From Practice to Theory and Vice Versa [Текст] / D. Rémy // Applied Semantics / ed. by G. Barthe [et al.]. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. — P. 413—536.

104. noCanren web-page [Текст]. — URL: https://github.com/Lozov-Petr/noCanren (visited on 06/01/2022).

105. OCanren with fair conjunction web-page [Текст]. — URL: https://github.com/JetBrains-Research/OCanren/tree/fair (visited on 06/01/2022).

Список рисунков

1.1 Пример выполнения ручного преобразования 19
1.2 Некорректный случай для ручного преобразования 20
1.3 Семантика большого шага для языка арифметических выражений . 25
1.4 Семантика малого шага для языка арифметических выражений . . . 26
1.5 Семантика в нотации Феллейсена для языка арифметических выражений 28

2.1 Синтаксис входного языка 32
2.2 Правила типизации для исходного языка 33
2.3 Семантика исходного языка 34
2.4 Синтаксис реляционного расширения 35
2.5 Правила типизации для реляционного расширения 36
2.6 Семантика реляционного расширения 37

4.1 Семантика малого шага локальных вычислений 67
4.2 Ангелическая семантика языка miniKanren 68
4.3 Обобщённая семантика языка miniKanren, параметризованная предикатом выбора $\mathcal{P}$ 73

6.1 UML-диаграмма компонент транслятора функциональных программ в реляционные 85
6.2 UML-диаграмма компонент реляционного расширения, оснащенного вполне квазиупорядочевающим предикатом выбора . . 88

Список таблиц

1 Время исполнения (в секундах) набора оптимизированных вручную программ с использованием классической и справедливой конъюнкции 92
2 Время исполнения (в секундах) набора неоптимизированных программ с использованием классической и справедливой конъюнкции 93

SAINT-PETERSBURG STATE UNIVERSITY

On the rights of the manuscript

Peter Lozov

Automated Synthesis and Efficient Execution of Relational Programs

Scientific specialty
2.3.5. Mathematical and software support
for computers, complexes and computer networks

DISSERTATION
for the degree of Candidate of Physico-Mathematical Sciences

Translation from Russian

Scientific supervisor:
Doctor of Science, Assistant Professor
Dmitry Koznov

Saint Petersburg
2022

Contents

Page

Introduction 4

Chapter 1. Background 12
1.1 Relational Programming and miniKanren 12
1.2 Methods of Relational Execution of Functional Programs 16
1.3 Operational Semantics 20

Chapter 2. Relational Conversion of Typed Functional Programs . 27
2.1 Source Functional Language 28
2.2 Relational Extension 32
2.3 Relational Conversion 37

Chapter 3. Static and Partial Dynamic Correctness of Relational Conversion 42
3.1 Static Correctness of Relational Conversion 42
3.2 Partial Dynamic Correctness of Relational Conversion 45

Chapter 4. The miniKanren Semantics with Dynamic Control of Conjunct Order 53
4.1 Classic Left Biased Conjunction in miniKanren 53
4.2 Angelic Semantics and Fairness 58
4.3 General miniKanren Semantics with Selection Predicate 63
4.4 Fair miniKanren Semantics by Well Quasi-Ordering 67

Chapter 5. Conjunction Fairness in miniKanren Semantics Equipped with a Well Quasi-Ordering Choice Predicate 69
5.1 Convergence of State Leaves of Angelic Semantics and Semantics with a Quasi-Ordering Choice Predicate 69
5.2 Preserving Convergence of Angelic Semantics 71
5.3 Conjunction Fairness in Semantics with a Well Quasi-Ordering Choice Predicate 73

Chapter 6. Implementation and Experiments 75
6.1 Implementation 75

Page

6.2 Experiments 80

Conclusion 85

References 87

List of Figures 98

List of Tables 99

Introduction

Actuality. Logic programming first appeared in the late 60's, resulting from the research in the fields of artificial intelligence and automated theorem proving (ATP) [1—4]. Namely, the approaches of automated reasoning, originally a component of artificial intelligence and ATP systems, laid the foundation of logic programming thanks to the work of R. Kowalski [5] and A. Colmerauer [6]. Modern logic programming languages, among which Prolog [7—9] is the most well-known, follow the idea of R. Kowalski [10] that consists in dividing any kind of algorithm into the data necessary to solve the problem and the solution-searching strategy. This allows the developer to focus on the description of the problem, leaving the search for its solution to the computer. This provides logic programming languages with a high degree of abstraction both at the level of conceptual problem solution and in their code.

The main application of Prolog is the development of symbolic rule systems, in which declarative knowledge is encoded in terms of first-order logic [11—13]. The language is optimized for expressiveness and efficiency of such tasks, sometimes via deviating from basic ideas of mathematical logic. In particular, classic Prolog contains a high-performance, but mathematically incorrect unification without "Occurs Check" [14]. For more efficient and flexible search, Prolog uses depth-first search [15; 16] and the cut operator [17], as a result of which solution search in classic Prolog is not complete.

Many other logical programming languages strive for higher "logical purity". We can mention the domain-specific language Datalog [18], designed to create queries for deductive databases. Datalog, unlike Prolog, does not include mathematically incorrect components. However, like many other domain-specific languages, it is non-Turing-complete. We should also note αProlog [19; 20], which is intended for compiler, interpreter and theorem proving tool development with the use of a nominal unification system and λProlog [21; 22], which is designed for effective program analysis, and partially supports higher-order functions and higher-order unification. Additionally, note the functional logic programming language Curry [23] based on Haskell, which allows its user to choose a search strategy [24]. Curry is used for program analysis and synthesis, as well as development of declarative user interfaces and database management systems [25—27]. Next, we have to mention

Mercury [28], a typed functional logic language equipped with a mode system [29]: this allows the developer to explicitly describe input and output parameters, as well as declare the determinism categories of the developed program. This information is necessary for the specification of the logic program for the purpose of optimizing it. Mercury is a general-purpose language used for application development [30]. Further on, we note Gödel [31], a general-purpose logic programming language, which is intended for creating metaprograms for analyzing, transforming, compiling, validating and debugging programs in other languages. This language is strictly typed, and has a declarative cut operator and unification. Finally, we would like to mention Twelf [32] with dependent typing, used to formalize mathematical theories [33] and metatheories of programming languages [34].

However, it should be noted that one of the main components of "logical purity" — the symmetry of disjunction and conjunction (which are basic operators in logic programming), is not implemented in these languages. It should be noted that in mathematical logic, these operations are independent of the order of calculation of arguments, and in existing logic programming languages, the order of calculation of conjunctions and disjunctions affects the convergence and efficiency of the search procedure. Operators that exhibit such asymmetric behavior will be called *unfair*.

A separate research area has been established in order to eliminate the unfairness of disjunction in logic programming — namely, relational programming [35]. Currently, this area is researched by the group led by William E. Byrd (USA), as well as the Department of System Programming of St. Petersburg State University under the guidance of Dmitri Boulytchev. ICFP, a major programming language conference, hosts the annual international miniKanren and the Relational Programming Workshop.

The main distinguishing feature of relational programming is fair disjunction, which is based on interleaving search [36; 37]: the solution search for each disjunct of the program is split into steps that are guaranteed to be finite. This makes possible conducting a complete search for an arbitrary number of disjuncts. The fairness of relational disjunction allows us to describe the problem at hand as a program that consists of a collection of declarative, non-directed relations [38; 39]. The main language that is used in relational programming is miniKanren [35; 40]. Initially, it was a minimalistic extension of the Scheme and Racket [41] languages and contained only five operators. Its simplest implementation consisted of less than a hundred lines

of code. Today, miniKanren is actually a family of languages which contain various extensions designed to increase its expressiveness and declarativeness [42—45].

The development of relational programs is a difficult task, since the developer is required to possess special knowledge and skills, as well as a deep understanding of the semantics of each relational operator. It was noticed that in practice, the development of a relational program consists of two stages: first, a program is written in a functional language and then it is "manually converted" into a relational one. The second stage turns out to be mechanistic and repetitive, thus, it can be automated. Thus, the task arises of creating an automated method for converting functional programs into relational ones.

It should be noted that in the general case, the original functional program does not contain sufficient information necessary to unambiguously map it to its relational image. In particular, when constructing such an image, the optimal order of conjuncts cannot be determined, which is a key factor in the efficiency and convergence of the relational program execution process. The importance of the order of conjuncts is due to the unfairness of the conjunction inherited from logic programming. Therefore, an automatically constructed relational program requires "manual" editing for each particular relational query. However, the optimal order of conjuncts can be set during execution based on the current state of the relational program. Such dynamic control of conjunct order increases execution efficiency of automatically constructed relational programs, as well as eliminates the influence of conjunct order on the convergence of the execution process. This gives rise to the task of creating a fair method of managing conjunct order in the execution of relational programs.

Background. W. E. Byrd proposed *unnesting*: a method for converting functional programs into relational ones [40]. It consists of converting each function into a relation by adding an additional argument, eliminating nested calls and replacing the pattern matching with a disjunction of unifications with each pattern. This method does in theory construct a relational program from a functional program, but it supports only first-order functions, has no formal description and has not been implemented. W. Bird, M. Ballantyne, G. Rosenblatt and M. Might have considered a similar task: executing of a functional program by means of a relational interpreter. This approach enables relational execution of functional programs by introducing free logical variables into the source program. The proposed approach also enables synthesis of functional programs using a set of test data [46]. The

disadvantage of this approach is its low effectiveness due to the additional level of interpretation.

The problem of effective management of conjunct order in the field of relational programming has been considered in many various ways [40; 46; 47]. One aspect of the unfair conjunction behavior is prioritizing the computation of independent disjuncts generated by the conjunction. In particular, in the classic implementation of miniKanren, the highest priority is given to earlier disjuncts generated by the conjunction. There is, however, an approach proposed by K.-C. Lu, W. Ma and D. P. Friedman, which balances computation time of various disjuncts generated by the conjunction [48]. This makes the conjunction fairer, but the order of the conjuncts still affects both the performance and the convergence of the computation of the relational program. In addition, the behavior of the conjunction can be made more fair by detecting divergent conjuncts and postponing their calculation. D. Rozplokhas and D. Boulytchev discover the divergence of conjuncts during the execution of a relational program and propose to dynamically reorder them [49]. In this case, the data obtained during the execution of the divergent conjunct is erased. This approach proved to be effective in practice, however, the conservative permutation of conjuncts does not use the information obtained during the execution of the conjunct before the rearrangement. There are also examples where, within this approach, the order of conjuncts still affects convergence. Similar tasks were also considered in a broader context. It is worth noting that many of the results dedicated to the problems of controlling the order of computation in Prolog are focused on overcoming the inherent incompleteness of depth-first search. For example, to better control the structure of the search tree, T. Schrijvers, M. Triska and B. Demoen proposed a fairer disjunction [50]. To ensure the completeness of the search in Prolog, tabling is used in some cases [51—53], which, however, is not a universal solution due to its large overhead costs. Note that many problems of logic programming for managing the evaluation order are not relevant in the case of relational programming, since for the latter there is a complete search procedure [54; 55]. There is a certain similarity between the problems we tackle and the problems that arise in multithreaded Prolog implementations. G. Gupta, E. Pontelli and their colleagues proposed “Dependent And-parallelism”, which enables parallel conjunct execution and deals with the effects caused by the variability of the execution of conjuncts depending on their order [56]. However, unlike our case, their main efforts

are aimed at improving performance while preserving the semantics of left-to-right depth-first search.

Thus, the considered problem of effective conversion and execution of functional programs is specific for relational programming, and its solution will significantly reduce the complexity of relational program development.

Aim of this work is to create an approach to construction and execution of relational programs via relational conversion of functional programs and dynamic control of execution order of relational conjuncts.

We have identified the following **tasks** that are necessary to achieve this goal.

1. Development of a method for relational conversion of typed general-form functional programs.
2. Proof of static and dynamic correctness of relational conversion.
3. Development of miniKanren semantics with a procedure for dynamic control of conjunct calculation order.
4. Proof of conjunction fairness for the proposed semantics.

The main results submitted for defense.

1. We propose a new approach to relational conversion of general-form functional programs and prove its static and dynamic correctness.
2. We introduce formal angelic semantics of the miniKanren and prove their equivalence to declarative miniKanren semantics. We propose the notion of conjunction fairness as property of angelic semantics.
3. We define formal semantics of relational miniKanren with dynamic control of the conjunct calculation order and prove conjunction fairness.
4. We implement on OCaml relational conversion for a subset of OCaml, and conduct an experimental study that shows high effectiveness of automatically obtained relational programs.
5. Finally, we implement on OCaml an embedding of miniKanren with dynamic control of conjunct calculation order in functional language OCaml, for which our experimental study indicates high efficiency in comparison to the classic implementation of miniKanren with a non-optimal order of conjuncts and the insignificance of overhead with an optimal order of conjuncts.

Scientific novelty of the results obtained in the study is as follows.

1. We are the first to present a method of relational conversion of general-form functional programs with proven static and dynamic correctness.

2. The angelic semantics of miniKanren are formally described for the first time, which lets us be the first to introduce the concept of conjunction fairness in relational programming.
3. The formal semantics of miniKanren with dynamic control of conjunct order are described for the first time.

Practical influence of the work consists in creating and implementing a method of relational conversion of general-form functional programs. This method can be used to simplify the creation of relational programs up to the complete exclusion of "manual" development of relational code. Furthermore, the proposed method of execution of relational programs with dynamic control of conjunct order is also of practical significance: it enables effective execution of relational programs and eliminates the need to edit programs for various relational queries.

Methodology and research methods. Our research methodology is based on a formal approach to the description of programming language semantics. The paper uses classic methods for describing the semantics of large and small steps in the form of a set of inference rules. We utilize the basic concepts of logical programming, such as the method of unifying logical expressions, automatic solution search, and non-deterministic execution. The presented work also uses the relational programming apparatus, referring to unification operators and disequality constraints as the main tool for constructing solutions, complete interleaving search. The software implementation of the theoretical results is conducted in OCaml, Haskell and Scheme, all of which are functional languages.

Approbation. The main results of our research were reported at the following scientific events: the PLC 2017 conference (April 3-5, 2017, Rostov-on-Don, Russia), the TFP 2017 Symposium (June 19-21, 2017, Canterbury, UK), the ML 2017 seminar, combined with ICFP 2017 (September 3-9, 2017, Oxford, UK), the miniKanren 2019 seminar combined with ICFP 2019 (September 18-23, 2019, Berlin, Germany), the miniKanren 2020 seminar combined with ICFP 2020 (August 20-28, 2020, New Jersey, USA), at the APLAS 2020 Symposium (November 30 - December 2, 2020, Fukuoka, Japan), the PEPM 2021 workshop, combined with the POPL 2021 Symposium (January 17-22, 2021, Copenhagen, Denmark).

The author's **personal contribution** in publications made with co-authors is distributed as follows. In the works [57; 58], we have developed a method of relational conversion of functional programs, proven its static and dynamic correctness, and also created an implementation in OCaml [59] and conducted experiments. Its co-

authors participated in the discussion of ideas, formalization of the method, and improved the text of the article. In [60], we have participated in the creation of the basic approach presented in the article, and developed a functional interpreter for a subset of the OCaml language, the relational image of which solved the search problem. Its co-authors proposed the idea of using relational interpreters to solve search problems and adapted the method of conjunctive partial deduction for relational programs. In [61], we proposed and formalized the miniKanren semantics with directed conjunction based on unnesting, miniKanren semantics with dynamic control of the order of conjuncts based on free variables in call arguments, as well as implemented the proposed ideas in Haskell and conducted experiments. The article's co-author participated in the discussion of the main ideas as well as the formalization of semantics, and, additionally, improved the text of the article. In [62], our contribution consists of formalization of miniKanren's angelic semantics and semantics with directed conjunction, a proof of conjunction fairness in the proposed semantics, and an experimentally tested implementation in Haskell. The co-author suggested using angelic semantics to determine conjunction fairness, participated in its proof, and improved the text of the article. In [63], we have proposed using relational conversion in order to create a relational interpreter for pattern matching, as well as participated in the discussion of the other ideas proposed in the article. Our co-authors developed the method and conducted the experiments.

Publications. The main results on the topic of the dissertation are presented in 6 scientific papers, 1 of the publications in a journal recommended by the Higher Attestation Commission, 3 — in periodic scientific journals indexed by Web of Science and Scopus.

Scope and structure of work. The dissertation consists of an introduction, 6 chapters and conclusion.

The full scope of the dissertation is 99 pages including 16 figures and 2 tables. The list of references contains 105 titles.

Acknowledgments. First of all, I would like to thank Dmitry Bulychev for the opportunity to realize the beauty of relational programming, for his guidance in the early stages of this research, for his invaluable contribution to my work, for his willingness to support and share his experience. I thank William Byrd for the very creation of relational programming, the many discussions about my dissertation, and research assistance.

With great warmth I would like to express my gratitude to my supervisor, Dmitry Koznov, for his inexhaustible energy, insight and wisdom, which he shared throughout our work.

I express my gratitude to Andrey Terekhov and the Department of System Programming of St. Petersburg State University, as well as to JetBrains and Huawei for the unique opportunity to engage in science as the main activity. I am grateful to Dmitry Kosarev for his invaluable support and assistance in the practical component of this work. I would like to express special gratitude to my fiancee, Anastasia Sadykova, because she gives me inspiration and energy for everything that I do. Finally, I am grateful to my parents who have been my support throughout my life and have shown an inexhaustible interest in my scientific work.

Chapter 1. Background

This chapter provides an overview of the miniKanren relational language, provides examples of its use, and discusses existing approaches to relational execution of functional programs. In addition, the basic concepts used in this thesis are introduced: big-step and small-step operational semantics, various types of operational semantics in the notation of Matthias Felleisen [64].

1.1 Relational Programming and miniKanren

The main distinguishing feature of the miniKanren [35; 40] language is non-directional evaluation of relational programs: the parameters and the result of relational relations do not differ at the syntax level, which makes it possible to describe non-directional programs and execute them in different "directions". This approach turns out to be useful in practice because it is much easier to express tasks as queries to non-directional relational programs. This observation is confirmed by a number of examples. As illustrations, consider the problem of type inference for Simply Typed Lambda Calculus [65] and the problem of Type Inhabitation [66]. These tasks can be expressed as queries to an easier-to-implement relational program that checks the correctness of lambda expression typing.

Another illustration is the task of constructing "quines" [38] — programs whose execution returns an exact copy of their source text. This task is represented as a query to the relational interpreter of the language in which the quine is to be built. Moreover, the development of a relational interpreter is simpler compared to quine construction. Finally, the solution to the problem of constructing all permutations of a given list can be expressed as a query to relational list sorting, which is easier to implement.

In the context of this paper, we will consider a specific implementation of the miniKanren language, which is called OCanren [67] and is an extension of the OCaml [59] functional language. OCanren conforms to the classical implementation of miniKanren [41; 68] with a disequality constraint [69]. This particular version

differs from it in strict typing, which makes it possible to prevent some of errors that could be made during the development of relational programs during compilation.

Next, we will examine the OCanren language from the user's point of view, considering only the intuitive definition of its constructions; a formal description of OCanren will be presented in Chapter 2. In this and subsequent chapters, we will use a simplified syntax that differs slightly from the actual syntax of the OCanren implementation in order to make the text more readable. We will not consider relational query execution operators `run N` and `run`*, the two-parameter type of representation of logical expressions, fully polymorphic data types necessary for simultaneous use of expressions in functional and logical domains, etc. The syntax of OCanren is presented in more detail in [67].

The basic concept of the miniKanren language is a *goal*. In the OCanren language, a goal can be an arbitrary expression of the reserved goal type $\mathfrak{G}$. The result of evaluating the goal is a data stream (possibly infinite) containing answers that meet the conditions of the goal. There are only five syntactic forms of goals, denoted below as g, g_1, g_2, etc.

- Unification is the first basic operator for creating goals, which is defined as $t_1 \equiv t_2$, where t_1 and t_2 are some terms consisting of constructors and variables [70]. If the terms t_1 and t_2 can be unified, then the goal is considered successful, and in this case the result of unification is a singleton stream containing the most general unifier of the terms t_1 and t_2. Otherwise, the goal is considered unsuccessful, and its result is an empty stream.
- The disequality constraint operator is necessary for constructing goals. Its behaviour can be characterized as inverse to unification. It is defined as $t_1 \not\equiv t_2$, where t_1 and t_2 are some terms.
- Disjunction is an operator of the form $g_1 \vee g_2$, where g_1 and g_2 are goals, and both goals are evaluated independently (the so-called fair disjunction). The result of executing disjunction is a union of the answers of the goals g_1 and g_2.
- Conjunction is an operator of the form $g_1 \wedge g_2$, where g_1 and g_2 are goals. During evaluation, g_1 is evaluated first, then g_2 is evaluated in the context of each of the answers of g_1 (thus, the conjunction is not fair). As a result, a stream of answers that satisfy both g_1 and g_2 is produced.
- The operator for introducing a fresh variable is defined as $\underline{\texttt{fresh}}(x)\ \ g$, where x is a variable and g is a goal. This operator is necessary to introduce

variable x missing in the current context, which is used in the goal g (further on in this paper we will call such variables *fresh*).

The terms used in unification operators and disequality constraints are arbitrary expressions of the polymorphic logical type α^o. The postfix $\square^o$ is the traditional way to denote relational entities, and we will use it for both relation names and types[1].

The simplest logical type expression is a variable introduced by the fresh operator. Another example is a primitive expression introduced into the logical domain using the built-in primitive "$\uparrow$", such as $\uparrow 3$ (its type is int^o) or $\uparrow$true (its type is bool^o). Other types (tuples, lists, user algebraic data types, etc.) can also be used in relational programs if they are introduced into the logical domain via the same primitive. For example, the expression $\uparrow$(1, "abc") has the type (int * string)^o, and the expression $\uparrow$[1; 2; 3] has the type (int list)^o. However, since unification and disequality constraint are recursive and work only with logical type expressions, logical type "^{o}" must be applied to all elements of the type. Indeed, a boolean variable can only be located at the position where a boolean type is expected. Thus, when unifying, you can use a value of the type (int * int)^o as *integer* value, but for relational management of *contents* of the pair, the type (int^o, int^o)^o is required. This makes it impossible to use some built-in and standard types in relational code — for example, the predefined list type is not flexible enough, since in the standard definition the tail of the list is not supplemented by the logical domain type. Instead, you need to introduce a logical list type:

```
type α llist = [] | (::) of α° * (α llist)°
```

With this definition, we can implement various relations over lists, for example:

```
val append : (α llist)° → (α llist)° → (α llist)° → 𝔊
let rec append° = λ x y xy .
  (x ≡ ↑[] ∧ xy ≡ y) ∨
  fresh (h t ty)
     x  ≡ ↑(h :: t) ∧
     xy ≡ ↑(h :: ty) ∧
     append° t y ty
```

[1]In the current implementation of OCanren, terms have a more complex two-parameter type that encodes the tagging necessary to perform unifications and convert the results of a relational program into a functional form for further use in a functional context; these details, however, are irrelevant to the goals of this work, and we will adhere to the simplified version.

In this example, we defined the ternary concatenation relation of relational lists `append`^{o}, a canonical example in relational programming. This relation is constructed using case analysis and recursion.

1. If the first list is empty, then the second and third lists should be equal.
2. Otherwise, the first list can be split into head and tail denoted by two fresh variables h and t. We also need a new variable ty to denote a list that will be equal to the concatenation of y and t. To ensure this, we use a recursive call of `append`^{o}. We get the final result by combining h and ty.

The definition's of `append`^{o} parameters are three logical lists `x`, `y` and `xy`. It describes a goal that can be executed or combined with other goals. The result of the evaluation is a stream of answers, and each element of the stream contains a description of the constraints for logical variables that must be satisfied in order for the parameters to belong to the relation.

We will denote the primitive of relational query execution with the "$\rightsquigarrow$" symbol. Then the query

$$\underline{\texttt{fresh}}\ (q)\ \texttt{append}^o\ \uparrow(\uparrow\texttt{1}\ ::\ \uparrow\texttt{[]})\ q\ \uparrow\texttt{[]}\ \rightsquigarrow\ \texttt{[]}$$

will return an empty stream, since there is no list q, the concatenation of which with `(1 :: [])` will result in an empty list. At the same time, evaluation of query

$$\underline{\texttt{fresh}}\ (q)\ \texttt{append}^o\ q\ \uparrow\texttt{[]}\ \uparrow(\uparrow\texttt{1}\ ::\ \uparrow\texttt{[]}])\ \rightsquigarrow\ \texttt{[}q \mapsto\ \texttt{1 :: []]}$$

will return the expected constraint for q.

As can be seen from the type of relation, relational concatenation is polymorphic, as is its functional counterpart. However, query

$$\underline{\texttt{fresh}}\ (q)\ \texttt{append}^o\ \uparrow(\uparrow\lambda x.x\ ::\ \uparrow\texttt{[]}])\ q\ \uparrow(\uparrow\lambda y.y\ ::\ \uparrow\texttt{[]}])$$

ends with a runtime error due to the inability to unify higher-order expressions. This is a fundamental limitation that exists in the original miniKanren, which uses first-order syntactic unification [70]. This example demonstrates that, unlike in pure OCaml, typing in OCanren is weaker. To restore strict typing, some type variables must be limited to first-order values only. Lack of direct support for bounded polymorphism [71] in OCaml makes checking this constraint problematic. However, our experience shows that in practice this disadvantage, although rarely, leads to errors in the development of relational programs. In the following, we assume that in polymorphic types, some type variables can be implicitly restricted to a set of first-order types, and these restrictions are adhered to in all instances of these type variables.

1.2 Methods of Relational Execution of Functional Programs

As mentioned in the previous section, relational programming facilitates effectively solving many problems via non-directional relational evaluation. However, writing such programs is a non-trivial task, which includes optimizing relations for all possible execution options, building efficient non-deterministic evaluation, etc.

In this regard, the possibility of relational execution of programs written in other languages is relevant. Such an opportunity will makes it possible to use familiar programming languages for program development (primarily of functional programs due to their similarity to relational ones), which will then be executed relationally. This approach provides the ability, for example, to simulate evaluation of inverse functions without direct use of relational programming. This makes development easier in the case when the solution of the inverse problem is much simpler in comparison with the solution of the original problem.

Currently, there are two approaches to relational execution of functional programs: relational conversion [40; 57], which creates a relational program equivalent to the original functional program, as well as relational execution of functional programs using relational interpreters [46].

1.2.1 Syntactic relational conversion of first-order functional programs

The "Unnesting" method [35; 40] was proposed in order to build a relational program based on a functional one. This method has not been implemented, so in this paper we will consider it as a manual conversion.

This conversion is based on introducing new variables for each nested function application. After each subexpression is associated with a variable, all pattern matchings must be converted to a disjunction. Further, all previously introduced variables, as well as variables used in pattern matching, should be declared using the `fresh` operator. Finally, a corresponding variable must be added as an additional parameter to each function application, and each converted function must be supplemented with an argument that must be unified with the result of the evaluation. In the case of the OCanren language, each constructor in the source

```
let rec append = λ x y.
  match x with
  | [] → y
  | h :: t →
    h :: append t y
```

a)

```
let rec append = λ x y.
  match x with
  | [] → y
  | h :: t →
    let ty = append t y in
    h :: ty
```

b)

```
let rec appendᵒ = λ x y xy.
  (x ≡ ↑[]  ∧  xy ≡ y)  ∨
  (fresh (h t ty)
    x  ≡ ↑(h :: t)  ∧
    xy ≡ ↑(h :: ty)  ∧
    appendᵒ t y ty)
```

c)

Figure 1.1 — Example of Unnesting conversion

program must be supplemented with an auxiliary constructor $\uparrow$ to transfer all constructors to the logical domain.

Consider again concatenation of two lists (Fig. 1.1, a). This example shows a classic implementation that moves elements from the first list to the second one step by step and eventually returns a list containing all the elements of the source lists. After adding names for all nested function applications — in this example, this is the variable ty for the only application of `append` t y, we get a function equivalent to the original one (Fig. 1.1, b). Next is the main conversion step: replace the pattern matching with a disjunction; add an additional argument xy and add unification of this variable with the result in each disjunction; declare all the pattern matching variables and the variable ty introduced for nested application using the `fresh` operator; apply the nested call to the corresponding one variable ty and we get the final relational program (Fig. 1.1, c).

However, due to the syntactic nature of the conversion, not every definition can be converted into a relational form with this method. This conversion works correctly only with first-order functions. However, if higher-order functions are used, this method can construct an incorrect relational program. Consider, for example, the

```
let bar = λ x.
  let f = λ x. x in
  let g = λ a. f in
  g A y
```

a)

```
let barᵒ = λ y r.
  let f = λ x r. x ≡ r in
  let g = λ a r = f ≡ r in
  g ↑A y r
```

b)

Figure 1.2 — An invalid case of Unnesting conversion

definition in Fig. 1.2, a (this program is provided only to illustrate the functioning of the conversion and is artificial). The conversion in question will produce the program shown in Fig. 1.2, b. Obviously, this result is incorrect, since the resulting relation contains unification of function f and logical variable r. In order for this method to build a correct relational program in this case, it is necessary as a preliminary step to use an η-extension to the definition of g, explicitly revealing its functional nature syntactically.

1.2.2 Relational Execution of Functional Programs Using Relational Interpreters

Relational interpreters are a powerful and flexible tool for partial or complete program synthesis due to the possibility of introducing logical variables into the interpreted program. This makes it possible to synthesize both parts of programs and whole small programs.

Let $\texttt{eval}^o$ be some relational interpreter whose parameters are the program to be interpreted, input data, and the expected result. Then we can directly execute some program $PROG$ with input data IN using the following query:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o\ PROG\ IN\ q.$$

This query will associate the q variable with the interpretation result. However, in addition to interpreting programs, which can be done by any other interpreter, we can use a relational interpreter for different tasks of program synthesis and validation. For example, we can synthesize a program from a set of test data $(IN_1,\ OUT_1), \ldots (IN_n,\ OUT_n)$ via the following query:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o\ q\ IN_1\ OUT_1\ \wedge\ \ldots\ \wedge\ \texttt{eval}^o\ q\ IN_n\ OUT_n.$$

The result of this query will be all programs that return OUT_i answers with IN_i inputs. If we substitute a partial program in place of q, we will synthesize the subexpressions of the program. In the case of a full program, we will get a check of whether test data holds. In addition, using a relational interpreter, we can solve the non-trivial task of quine construction. To do this, it is sufficient to describe the following program which returns itself as an answer:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o\ q\ \texttt{()}\ q.$$

This request will return a possibly infinite collection of all quines for the interpreted language.

In the context of this work, the most important application of relational interpreters is relational execution of functional programs. Indeed, depending on the location of logical variables in the parameters of the input data and the result of interpretation, we can execute a functional program in different directions. For example, the previously considered concatenation function of two lists `append` can be executed in the forward direction (for better readablity, `append` is presented in OCaml syntax; in practice, it is necessary to represent this function as a syntax tree):

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o \left(\begin{array}{l} \underline{\texttt{let}}\ \underline{\texttt{rec}}\ \texttt{append = }\lambda\ x\ y\texttt{.} \\ \quad \underline{\texttt{match}}\ x\ \underline{\texttt{with}} \\ \quad \texttt{[]}\ \rightarrow\ y \\ \quad h\ \texttt{::}\ t\ \rightarrow \\ \quad\quad h\ \texttt{::}\ \texttt{append}\ t\ y \end{array} \right) \texttt{([1], [2])}\ q$$

The response to this query will be `[1; 2]` — the result of concatenation of `[1]` and `[2]`. We can also execute the `append` function in the opposite direction:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o \left(\begin{array}{l} \underline{\texttt{let}}\ \underline{\texttt{rec}}\ \texttt{append = }\lambda\ x\ y\texttt{.} \\ \quad \underline{\texttt{match}}\ x\ \underline{\texttt{with}} \\ \quad \texttt{[]}\ \rightarrow\ y \\ \quad h\ \texttt{::}\ t\ \rightarrow \\ \quad\quad h\ \texttt{::}\ \texttt{append}\ t\ y \end{array} \right) \texttt{([1], }q\texttt{)}\ \texttt{[1; 2]}$$

As a result, we will get the list `[2]`, which must be added to the list `[1]` to get `[1; 2]`.

In the previous examples, we described deterministic queries. However, nondeterministic execution of interpreted programs is also possible. Consider the following query:

$$\underline{\texttt{fresh}}\ (q)\ \texttt{eval}^o \begin{pmatrix} \underline{\texttt{let}}\ \underline{\texttt{rec}}\ \texttt{append = }\lambda\ x\ y. \\ \quad \underline{\texttt{match}}\ x\ \underline{\texttt{with}} \\ \quad \texttt{[]}\ \rightarrow\ y \\ \quad h\ ::\ t \rightarrow \\ \qquad h\ ::\ \texttt{append}\ t\ y \end{pmatrix} (p,\ q)\ \texttt{[1; 2]}$$

This query describes all pairs of lists whose concatenation is equal to `[1; 2]`. The result of its execution will be a collection of all such pairs `[([], [1; 2]); ([1], [2]); ([1; 2], [])]`.

As it can be seen, this approach has many applications and, among other things, provides relational execution of a functional program. However, due to the additional level of interpretation, such execution has low performance compared to the execution of relational programs obtained via relational conversion of functional programs.

1.3 Operational Semantics

In the theory of formal languages, formal semantics are the classical method of describing the "meaning" of programs. There are various ways to represent formal semantics. *Denotational semantics* map mathematical objects to the syntactic representation of a program, while abstracting from the process of program execution [72]. *Axiomatic semantics* consist of a set of axioms by means of which the inference of the program execution results is carried out [73]. To represent the program execution process as a set of transition rules, *operational semantics* are used [74]. In the context of this work, operational semantics are used to describe the semantics of functional and relational languages, since this representation describes the process of program execution in the most detail. This makes it possible to maintain the congruence between the theoretical description of the language represented as operational semantics and the practical implementation represented as an interpreter.

There are two classes of operational semantics: big-step semantics (also called natural semantics) and small-step semantics (also called structural operational semantics). In addition, an important subclass of small-step semantics are semantics in Felleisen notation [64], another name of which is reductive semantics.

In this thesis, we will use operational semantics of all three classes. Therefore, we will look at each class in more detail. We will use the language of binary arithmetic expressions as an example to describe various types of operational semantics for. It is quite concise, but it will allow us to demonstrate the main features of various classes of operational semantics. To simplify the examples, we will use only one binary operation — addition, since the semantics of the other operators are defined in a similar way. The syntax of the arithmetic expression language includes natural numbers, variables, and the binary addition operator $L = \mathbb{N} \mid \mathbb{X} \mid L + L$.

To evaluate the values of variables in each of the semantics, it is necessary to define as an environment a mapping from variable names to natural numbers $\sigma : \mathbb{X} \Rightarrow \mathbb{N}$.

Finally, addition is denoted by "+". The semantic addition function is denoted by "$\oplus$". It takes two natural numbers and returns a natural number that is the result of addition.

Now we have everything we need to define operational semantics. In the following sections we will introduce big-step semantics, small-step semantics, and semantics in Felleisen notation and consider their distinctive features.

1.3.1 Big-step semantics

The operational big-step semantics was introduced by Gilles Kahn to represent the Mini-ML language, which is a dialect of the ML functional language [75]. The main distinguishing feature of the big-step semantics is the direct mapping of the program into the result. Therefore, such semantics are defined by a relation on a set of programs and in a certain *semantic domain*, the elements of which determine all possible results of program execution. In the case of the example under consideration, the semantic domain is the set of natural numbers.

The big-step semantics relation is usually represented as a recursive set of inference rules. Moreover, each rule compares the result of the evaluation of

$$\frac{}{(\sigma, n) \Rightarrow n} \quad [\textsc{Num}_B]$$

$$\frac{}{(\sigma, x) \Rightarrow \sigma(x)} \quad [\textsc{Var}_B]$$

$$\frac{(\sigma, e_1) \Rightarrow n_1 \qquad (\sigma, e_2) \Rightarrow n_2}{(\sigma, e_1 + e_2) \Rightarrow n_1 \oplus n_2} \quad [\textsc{Add}_B]$$

Figure 1.3 — Big-step semantics for the language of binary arithmetic expressions

this program to the environment and the program corresponding to the specified template. Some of the rules contain a number of conditions that the subexpressions of the program must satisfy. Rules without conditions are called axioms.

The big-step semantics (denoted "$\Rightarrow$") for the arithmetic expression language is shown in Fig. 1.3. This semantics consists of three inference rules. The axiom for constants $\textsc{Num}_B$ maps a constant to its own value. The second axiom for variables $\textsc{Var}_B$ with the help of the environment σ replaces the variable with its value. The last rule for addition $\textsc{Add}_B$ requires evaluating the values of each term and determines the result as the sum of these values using the "$\oplus$" function.

As it can be seen, simplicity of representation is an advantage of big-step semantics: its description requires fewer inference rules in comparison with other operational semantics. Because of this, proofs of various properties of semantics, for example, correctness and completeness, also become simpler.

However, in big-step semantics, there are no inference trees for divergent evaluations, which makes it impossible to define and prove their properties. Also, big-step semantics does not provide full control over evaluation order. For example, in the inference rule AB the evaluation order of the terms is not specified. At the same time, controlling the evaluation order is necessary in defining the semantics of the language: for example, in the case of λ-calculus, various strategies for evaluating function arguments, namely call by value and call by name, are expressed precisely through ordering the evaluation of parameters and function bodies [76]. For a more detailed definition of semantics of a language small-step operational semantics are used.

$$\frac{}{(\sigma, x) \to (\sigma, \sigma(x))} \quad [\textsc{Var}_S]$$

$$\frac{}{(\sigma, n_1 + n_2) \to (\sigma, n_1 \oplus n_2)} \quad [\textsc{AddV}_S]$$

$$\frac{(\sigma, e_1) \to (\sigma, \widetilde{e_1})}{(\sigma, e_1 + e_2) \to (\sigma, \widetilde{e_1} + e_2)} \quad [\textsc{AddL}_S]$$

$$\frac{(\sigma, e_2) \to (\sigma, \widetilde{e_2})}{(\sigma, n_1 + e_2) \to (\sigma, n_1 + \widetilde{e_2})} \quad [\textsc{AddR}_S]$$

Figure 1.4 — Small-step semantics for the language of binary arithmetic expressions

1.3.2 Small-step semantics

Small-step semantics was introduced by Gordon Plotkin [77]. In this notation, instead of mapping the result of its evalution to the program, we iteratively simplify the program by evaluating only some subexpression of it.

The small-step semantics is also represented as a set of inference rules, in each of which a simplified version is mapped to the source program. To fully evaluate the program, it is necessary to iteratively apply the rules of semantics, as long as at least one of them is applicable. This process is defined by the reflexive-transitive closure of the relation defined by the semantics inference rules.

The small-step semantics (denoted “$\to$”) for the language of arithmetic expressions is shown in Fig. 1.4 and consists of four inference rules. The first rule for variables $\textsc{Var}_S$ is an axiom and, as in the case of big-step semantics, maps a variable to its value. Note that for iterative application of semantics, the σ environment must also be transferred to the inference rule execution result. The second rule used for addition of the evaluated terms $\textsc{AddV}_S$ is also an axiom and performs addition. The rule for simplifying the left term $\textsc{AddL}_S$ requires performing a semantics step for the left term if it is not a constant. After that, the simplified version of the expression replaces the original left term. The last rule of simplification of the right term $\textsc{AddR}_S$ requires that the left term is a constant, and the right term is not. In this case, the right term will be simplified. Note that in this semantics there is no

rule corresponding to a natural number — since a number is a program that does not require any actions to be evaluated, there is no need for a separate rule.

Let us express the final semantics using a reflexive-transitive closure. Let p be some arithmetic expression, σ be some environment, and n be some natural number. Then the expression p in σ is computable in the semantics of "$\rightarrow$" and the result of the evaluation is n, if it is true that $(\sigma, p) \rightarrow^* (\sigma, n)$, where "$\rightarrow^*$" is the reflexive-transitive closure of the relation "$\rightarrow$".

First of all, we note that there are more rules in this semantics in comparison with big-step semantics for the same language. As a consequence, the proof of most properties of small-step semantics will be larger and more complex in comparison with a similar proof for big-step semantics. On the other hand, this semantics, due to three rules for addition, strictly determines the subexpression evaluation order: first the left term, then the right. In the case of λ-calculus, it is this feature that allows us to describe various evaluation strategies. Note that divergent computations in this semantics have an infinite, but structured chain of inference, which makes it possible to describe and prove the properties of divergent programs.

However, in this semantics, a single step is not atomic. In general, a single step also includes steps for subexpressions. Informally speaking, one step of the small-step semantics includes both the search for an expression that needs to be simplified, and the simplification itself. This complicates the analysis and proof of the properties of the language for which the semantics are developed. Semantics in Felleisen notation have been developed in order to provide a description in which each step is atomic.

1.3.3 Semantics in Felleisen Notation

Semantics in Felleisen notation are an alternative way of describing small-step semantics. The main ideas were proposed by Gordon Plotkin [76] and generalized by Matthias Felleisen in his PhD thesis [78]. Semantics in Felleisen notation is represented as a set of reduction rules, each of which defines one potential reduction step. The main distinguishing feature of this semantics is the atomicity of each step. This is achieved by dividing the search for the subexpression to be simplified into a set of atomic steps.

$$(C : S, \sigma, n) \rightsquigarrow (S, \sigma, C[n]) \qquad [\textsc{Num}_F]$$

$$(S, \sigma, x) \rightsquigarrow (S, \sigma, \sigma(x)) \qquad [\textsc{Var}_F]$$

$$(S, \sigma, n_1 + n_2) \rightsquigarrow (S, \sigma, n_1 \oplus n_2) \qquad [\textsc{AddV}_F]$$

$$(S, \sigma, e_1 + e_2) \rightsquigarrow (\square + e_2 : S, \sigma, e_1) \qquad [\textsc{AddL}_F]$$

$$(S, \sigma, n_1 + e_2) \rightsquigarrow (n_1 + \square : S, \sigma, e_2) \qquad [\textsc{AddR}_F]$$

Figure 1.5 — Semantics in Felleisen notation for the language of binary arithmetic expressions

To describe semantics in Felleisen notation, it is necessary to introduce the concept of *context* — an expression containing a “hole”:

$$K = \square + L \mid L + \square.$$

The semantics environment will still contain a function for replacing variables to numbers, but it will be supplemented with a *stack of contexts* containing deferred expressions of the form $S = \varepsilon \mid K : S$.

A context arises when it is necessary to evaluate a subexpression of the executed program. In this case, in the executed program, the subexpression is replaced by a “hole and the subexpression itself takes the place of the executed expression.

Semantics in Felleisen notation (denoted “$\rightsquigarrow$”) for the language of arithmetic expressions is shown in Fig. 1.5. It contains five reduction rules. If the evaluated expression is a natural number (rule $\textsc{Num}_F$), and the stack of contexts is not empty, we extract the head element from the context and substitute this number in place of the “hole”, since the evaluated expression cannot be simplified. When evaluating a variable (the $\textsc{Var}_F$ rule), as in the aforementioned semantics, we will replace the variable to the number using the σ function. The context in this case remains unchanged. If it is necessary to execute addition of two natural numbers (the $\textsc{AddV}_F$ rule), then we will conduct it using the “$\oplus$” function , also leaving the context unchanged. The remaining two rules $\textsc{AddL}_F$ and $\textsc{AddR}_F$ determine

the order of term evaluation: if the left term is not a number, evaluate it first, and otherwise evaluate the second term. In both rules, the stack receives the context corresponding to the addition operator in question, in which one of the terms was replaced by a “hole”. Note that in this semantics, all rules are axioms, since they do not contain conditions. Also, not a single rule corresponds to the case when the executed expression is a natural number and the stack of contexts is empty. It is this case that signals the completion of the execution.

The final semantics is determined using a reflexive-transitive closure, as well as small-step semantics. Let p be some arithmetic expression, σ be an environment, and n be some natural number. Then the expression p in the environment of the function σ and an empty stack of contexts ε is computable and takes the value n if it is true that $(\varepsilon, \sigma, p) \rightsquigarrow^* (\varepsilon, \sigma, n)$, where “$\rightsquigarrow^*$” is the reflexive-transitive closure of the relation “$\rightsquigarrow$”.

This semantics makes it possible to describe in detail the process of evaluation: i.e. to set the order of evaluation of subexpressions and to save the path to the executed subexpression using of a stack of contexts. The stack of contexts also makes it possible to express such language features as exception handling and continuations. Finally, such a representation of semantics simplifies the analysis of the properties of the evaluated program due to the atomicity of each step of execution.

Chapter 2. Relational Conversion of Typed Functional Programs

Our approach is based on the conversion of functional programs into relational programs, and it is assumed that the source functional language has a *relational extension*. This extension is used to write down converted programs. The relational extension approach appears to be natural, since the primary relational language, miniKanren, is an embedded domain-oriented language: there are miniKanren implementations embedded into Scheme [40], OCaml [67], Haskell[1] and other languages. Thus, in a relational language, it is possible to reuse many of the source language constructs instead of redefining them, which would have to be done when creating a separate relational language. For example, function definition, abstraction, constructors, etc. In addition, the relational extension approach is justified from a practical point of view, since it assumes that the programmer works in a single language infrastructure — first, they write a program in a functional language, and then they automatically obtain a relational program for the relational extension of the same language.

This chapter provides a formal description of relational conversion of typed functional programs. The conversion consists of the following components:

- syntax, type system, and semantics of the source functional language,
- syntax, type system, and semantics of relational extension of the source language,
- a system of rules for converting types and programs into a relational form.

As our source functional language, we have utilized the simplest subset that includes λ-calculus, let-bindings and pattern matching. All these components are present in most real functional languages, such as ML, Haskell, Racket, F#, etc. Due to this, the proposed approach for converting functional programs into relational programs is not constrained to a specific language. Instead, it easily can be implemented for the required functional language. Moreover, despite its minimalism, this language contains everything necessary for embedding a relational extension. As a result, we can reuse the syntax, the type system, and the semantics of the source functional language when describing the semantics of a relational extension.

[1] The official website of miniKanren [79] contains the full list of its embeddings.

2.1 Source Functional Language

The syntax of the source functional language is shown in Fig. 2.1. First of all, the language includes λ-calculus consisting of functional variables, application, and abstraction. Lambda-calculus is necessary to describe anonymous functions — a fundamental tool for describing non-recursive computations of any functional programming language. In addition to anonymous functions, the language is provided with the ability to describe named functions using let bindings, which make it possible to refer to functions by their names and are a classic way of describing recursive computations.

Furthermore, this language contains constructors with a fixed number of parameters C^n, which are necessary for constructing language values. Note that we define two nullary constructors true and false among the constructors, and the equality operator "=" among the functional variables. Together, they supply the language with polymorphic comparison, the behavior of which is similar to relational unification. Therefore, polymorphic comparison has an effective relational image.

$$
\begin{array}{rcl}
\mathcal{E} & = & x \\
& | & \lambda x.e \\
& | & e_1\ e_2 \\
& | & C^n(e_1, \ldots, e_n) \\
& | & \underline{\texttt{true}} \\
& | & \underline{\texttt{false}} \\
& | & \underline{\texttt{let}}\ x\ \texttt{=}\ e_1\ \underline{\texttt{in}}\ e_2 \\
& | & \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ \texttt{=}\ \lambda x.e_1\ \underline{\texttt{in}}\ e_2 \\
& | & e_1 = e_2 \\
& | & \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \rightarrow e_i\} \\
\\
\mathcal{P} & = & C^n(x_1, \ldots, x_n)
\end{array}
$$

Figure 2.1 — Source language syntax

Finally, the source language includes pattern matching — a fundamental functional programming operator used to branch the program execution flow and deconstruct values. Note that in the context of this work, pattern matching only allows primitive patterns consisting of a constructor and a set of variables $C^n(x_1 \ldots x_n)$. This constraint is necessary to simplify the semantics of pattern matching. However, it is not significant, because generalized patterns can be expressed with primitive patterns. Furthermore, the "wildcard" (_) pattern is absent, since it requires a fundamentally different approach to pattern matching.

Types:

$$
\begin{array}{lcll}
\mathcal{X} & = & \alpha, \beta, \ldots & \text{(type variables)} \\
\mathcal{D} & = & \texttt{bool}, T^n, \ldots & \text{(constructor types)} \\
\mathcal{T} & = & \alpha \mid T^k(t_1, \ldots, t_k) \mid t_1 \to t_2 & \text{(types)} \\
\mathcal{S} & = & \forall \bar{\alpha}.t & \text{(closed types)}
\end{array}
$$

Typing rules:

$$\Gamma \vdash \underline{\texttt{true}}, \underline{\texttt{false}} : \texttt{bool} \quad [\textsc{Bool}_T]$$

$$\frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \texttt{bool}} \quad [\textsc{Eq}_T]$$

$$\frac{\Gamma \vdash e_i : t_i^C}{\Gamma \vdash C^n(e_1, \ldots, e_n) : t^C} \quad [\textsc{Constr}_T]$$

$$\Gamma, x : \forall \bar{\alpha}.t \vdash x : t[\bar{\alpha} \leftarrow \bar{t}'] \quad [\textsc{Var}_T]$$

$$\frac{\Gamma \vdash f : t_1 \to t_2 \quad \Gamma \vdash e : t_1}{\Gamma \vdash f\ e : t_2} \quad [\textsc{App}_T]$$

$$\frac{\Gamma, x : t_1 \vdash f : t_2}{\Gamma \vdash \lambda x.f : t_1 \to t_2} \quad [\textsc{Abs}_T]$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma, x : \forall \bar{\alpha}.t_1 \vdash e_2 : t}{\Gamma \vdash \underline{\texttt{let}}\ x\ = e_1\ \underline{\texttt{in}}\ e_2 : t}, \ \bar{\alpha} = FV(t_1) \setminus FV(\Gamma) \quad [\textsc{Let}_T]$$

$$\frac{\Gamma, f : t_1 \vdash \lambda x.e_1 : t_1 \quad \Gamma, f : \forall \bar{\alpha}.t_1 \vdash e_2 : t}{\Gamma \vdash \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ = \lambda x.e_1\ \underline{\texttt{in}}\ e_2 : t}, \ \bar{\alpha} = FV(t_1) \setminus FV(\Gamma) \quad [\textsc{LetRec}_T]$$

$$\frac{\Gamma \vdash e : t^C \quad \Gamma, x_1^i : t_1^{C_i}, \ldots, x_{k_i}^i : t_{k_i}^{C_i} \vdash e_i : t}{\Gamma \vdash \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{C_i^{k_i}(x_1^i \ldots x_{k_i}^i)\ \to e_i\} : t} \quad [\textsc{Match}_T]$$

Figure 2.2 — Typing rules of the source language

This language uses the Hindley-Milner type system [80; 81], which supports polymorphism and is the de facto basis of type systems for real functional languages. The resulting type system is presented as a set of rules in Fig. 2.2. In addition to type variables and functional types, this system contains a set of implicitly defined algebraic types T^k, and each constructor belongs to only one type. The $\textsc{Constr}_T$ rule assumes that type t^C has the form $T^k(t_1, \ldots, t_k)$, where each of the types t_i can be recovered from t_i^C as the type of the corresponding argument of the constructor of type T^k. In addition, in the $\textsc{Match}_T$ rule, the types of all patterns $C_i^{k_i}(x_1^i, \ldots, x_{k_i}^i)$

must be equal to t^C, and $t_j^{C_i}$ is the type of j-th argument of the C_i constructor used in the pattern. The $\textsc{Eq}_T$ rule specifies that both arguments of the equality operator must have the same, albeit arbitrary, type. Thus, this operator is a "polymorphic equality" operator.

The semantics of the source language (fig. 2.3) is a transition system over a set of *states*, each of which consists of execution of expression e with a stack of contexts $\mathcal{S}$. The transition relation is defined by a single step:

$$\langle \mathcal{S}, e \rangle \to \langle \mathcal{S}', e' \rangle.$$

The result of this step is a new stack of contexts $\mathcal{S}'$ and a new expression e'. A context is an expression containing a unique symbol called a "hole" ($\square$); informally, a stack of contexts can be defined as the path of evaluating an expression from the outer level to the position where the evaluation is currently taking place. Using context C and expression e, the full expression $C[e]$ can be constructed by substituting e in place of a unique "hole" in C. For any state $\langle C_1 : C_2 : \ldots : C_k, e \rangle$, it is possible to construct expression $C_k[\ldots[C_2[C_1[e]]]\ldots]$, which is an intermediate result of the evaluation in accordance with small-step semantics. This type of semantics description is called the Matthias Felleisen notation [64] for small-step semantics. Our choice of it was motivated by the fact that it can be extended to the case of the relational extension.

This semantics describes left-to-right evaluation of function parameters using a call by value evaluation strategy. The rules Beta, Mu, LetVal, LetRec and MatchVal are responsible for substitution according to variable names in abstractions and let bindings. The rule MatchVal assumes that exactly one pattern corresponds to the matched expression — this is an important difference from the classic semantics of pattern matching, in which patterns are checked sequentially from top to bottom until the first successful match. The rules EqTrue and EqFalse assume that values v, v_1, v_2 do not have the form $\lambda x \ldots$ or $\mu f \ldots$.

Finally, for a closed expression e and value v, $e \rightsquigarrow^f v$ is defined if and only if the following holds:

$$\langle \varepsilon, e \rangle \to^* \langle \varepsilon, v \rangle \,,$$

where ε is an empty stack of contexts, and $\to^*$ is a reflexive-transitive closure for $\to$.

Value:

$$\mathcal{V} = C^n(v_1, \ldots, v_n) \mid \lambda x.e \mid \mu f \lambda x.e \mid \underline{\texttt{true}} \mid \underline{\texttt{false}}$$

Context:

$$\mathcal{C} \;=\; \square\, e \mid v\, \square \mid C^n(\bar{v},\square,\bar{e}) \mid \square\texttt{=e} \mid \texttt{v=}\square$$
$$\mid\; \underline{\texttt{let}}\ x\ \texttt{=}\ \square\ \underline{\texttt{in}}\ e \mid \underline{\texttt{match}}\ \square\ \underline{\texttt{with}}\ \{p_i \to e_i\}$$

Stack of contexts:

$$\mathcal{S} = \varepsilon \mid \mathcal{C} : \mathcal{S}$$

State:

$\langle \mathcal{S}, e\rangle$ (stack of contexts, expression); $\langle \varepsilon, e\rangle$ (initial state); $\langle \varepsilon, v\rangle$ (final state)

Transition rules:

$$\langle C : \mathcal{S}, v\rangle \to \langle \mathcal{S}, C[v]\rangle \qquad [\textsc{Value}]$$

$$\langle \mathcal{S}, f\ e\rangle \to \langle \square\, e : \mathcal{S}, f\rangle \qquad [\textsc{AppL}] \qquad\qquad \langle \mathcal{S}, v\ e_2\rangle \to \langle v\, \square : \mathcal{S}, e_2\rangle \qquad [\textsc{AppR}]$$

$$\langle \mathcal{S}, e_1 = e_2\rangle \to \langle \square = e_2 : \mathcal{S}, e_1\rangle \qquad [\textsc{EqL}] \qquad\qquad \langle \mathcal{S}, v = e\rangle \to \langle v = \square : \mathcal{S}, e\rangle \qquad [\textsc{EqR}]$$

$$\langle \mathcal{S}, v = v\rangle \to \langle \mathcal{S}, \underline{\texttt{true}}\rangle \qquad [\textsc{EqTrue}]$$

$$\langle \mathcal{S}, v_1 = v_2\rangle \to \langle \mathcal{S}, \underline{\texttt{false}}\rangle,\ v_1 \neq v_2 \qquad [\textsc{EqFalse}]$$

$$\langle \mathcal{S}, (\lambda x.e)\ v\rangle \to \langle \mathcal{S}, e[x \leftarrow v]\rangle \qquad [\textsc{Beta}]$$

$$\langle \mathcal{S}, (\mu f \lambda x.e)\ v\rangle \to \langle \mathcal{S}, e[f \leftarrow \mu f \lambda x.e, x \leftarrow v]\rangle \qquad [\textsc{Mu}]$$

$$\langle \mathcal{S}, C^n(v_1, \ldots, v_{k-1}, e_k, \ldots, e_n)\rangle \to \langle C^n(v_1, \ldots, v_{k-1}, \square, \ldots, e_n) : \mathcal{S}, e_k\rangle \qquad [\textsc{Constr}]$$

$$\langle \mathcal{S}, \underline{\texttt{let}}\ x\ \texttt{=}\ e_1\ \underline{\texttt{in}}\ e_2\rangle \to \langle \underline{\texttt{let}}\ x\ \texttt{=}\ \square\ \underline{\texttt{in}}\ e_2 : \mathcal{S}, e_1\rangle \qquad [\textsc{Let}]$$

$$\langle \mathcal{S}, \underline{\texttt{let}}\ x\ \texttt{=}\ v\ \underline{\texttt{in}}\ e\rangle \to \langle \mathcal{S}, e[x \leftarrow v]\rangle \qquad [\textsc{LetVal}]$$

$$\langle \mathcal{S}, \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ \texttt{=}\ \lambda x.e_1\ \underline{\texttt{in}}\ e_2\rangle \to \langle \mathcal{S}, e_2[f \leftarrow \mu f \lambda x.e_1]\rangle \qquad [\textsc{LetRec}]$$

$$\langle \mathcal{S}, \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \to e_i\}\rangle \to \langle \underline{\texttt{match}}\ \square\ \underline{\texttt{with}}\ \{p_i \to e_i\} : \mathcal{S}, e\rangle \qquad [\textsc{Match}]$$

$$\left\langle \mathcal{S}, \underline{\texttt{match}}\ C_k^{n_k}(v_1 \ldots v_{n_k})\ \underline{\texttt{with}}\ \{C_i^{n_i}(x_1^i \ldots x_{n_i}^i) \to e_i\}\right\rangle \to \left\langle \mathcal{S}, e_k[x_j^k \leftarrow v_j]\right\rangle \qquad [\textsc{MatchVal}]$$

Figure 2.3 — Semantics of the source language

2.2 Relational Extension

Our relational extension adds five classical miniKanren operators to the source functional language in order to describe relational goals: unification, disequality constraint, disjunction, conjunction, and the operator for introducing a fresh variable. The extended syntax of the relational extension is shown in Fig. 2.4.

Note that adding relational operators makes it possible to construct incorrect expressions, for example, `λ x.(x ∧λ y.y)`. To eliminate such cases, a special extension of the source language typing system was developed. This approach corresponds to the current implementation of OCanren — the relational extension of OCaml [82]. A careful choice of types to represent expressions and goals in the OCaml implementation makes it possible to prune most incorrect programs during compilation.

$$
\begin{array}{lll}
\mathcal{E} & += & \underline{\texttt{fresh}}\ (x)\ \ e \\
 & | & e_1 \equiv e_2 \\
 & | & e_1 \not\equiv e_2 \\
 & | & e_1 \vee e_2 \\
 & | & e_1 \wedge e_2
\end{array}
$$

Figure 2.4 — Relational extension syntax

To extend the type system, we introduce a polymorphic constructor of type $\square^o$ with a corresponding logical term constructor $\uparrow$. Additionally, we introduce a special unique type $\mathfrak{G}$ intended to express relational goals. The typing rules for the relational extension are shown in Fig. 2.5.

These rules impose the following constraints: unification and disequality constraint allow only terms of the same logical type; and conjunction and disjunction are applicable only to relational goals. Note that within the framework of the proposed extension, the term can be computed as the result of executing an arbitrary expression in the original functional language (provided that this expression has the expected boolean type). However, such terms of a “higher order” cannot be obtained as a result of relational conversion. Therefore, this relational extension defines a richer language than is required for relational conversion.

The semantics of the extended language is shown in Fig. 2.6. First of all, the state of the original semantics has been expanded. In addition to the stack of contexts and the current expression, it contains a set of *semantic variables* Σ and *logical state* σ. Semantic variables are allocated and substituted instead of logical variables when the expression <u>`fresh`</u> is executed in the FRESH transition rule. The logical state is

Types:

$$\begin{array}{lcl} \mathcal{L} & = & \alpha^o \mid (T^n(l_1, \ldots, l_n))^o \quad \text{(logical types)} \\ \mathcal{T} & += & \mathfrak{G} \end{array}$$

Typing rules:

$$\frac{\Gamma, x : l \vdash e : \mathfrak{G}}{\Gamma \vdash \underline{\texttt{fresh}}\ (x)\ e : \mathfrak{G}} \quad [\textsc{Fresh}_T]$$

$$\frac{\Gamma \vdash e_1 : l \quad \Gamma \vdash e_2 : l}{\Gamma \vdash e_1 \equiv e_2 : \mathfrak{G}} \quad [\textsc{Unify}_T] \qquad \frac{\Gamma \vdash e_1 : l \quad \Gamma \vdash e_2 : l}{\Gamma \vdash e_1 \not\equiv e_2 : \mathfrak{G}} \quad [\textsc{Disequality}_T]$$

$$\frac{\Gamma \vdash e_1 : \mathfrak{G} \quad \Gamma \vdash e_2 : \mathfrak{G}}{\Gamma \vdash e_1 \wedge e_2 : \mathfrak{G}} \quad [\textsc{Conjunction}_T] \qquad \frac{\Gamma \vdash e_1 : \mathfrak{G} \quad \Gamma \vdash e_2 : \mathfrak{G}}{\Gamma \vdash e_1 \vee e_2 : \mathfrak{G}} \quad [\textsc{Disjunction}_T]$$

Figure 2.5 — Typing rules of the relational extension

updated when unification or disequality constraint is executed. All the transition rules of the source language are preserved, but they require supplementing the state with a set of semantic variables and a logical state. These added state components remain unchanged when executing the rules for the source language.

The set of values is extended with two new types of values: semantic variables and a special value success. Semantic variables are the result of executing free logical variables, the value success is the result of executing a successful goal.

The definition of context has also been extended with four new types of expressions with a “hole”. New contexts define deterministic left-to-right execution of unification and disequality constraints. The execution of disjunction and conjunction is non-deterministic. During the execution of a disjunction, only one sub-goal is selected (rules DisjL and DisjR). For a conjunction, it is determined nondeterministically which of the sub-goals will be evaluated first (rules ConjStartL and ConjStartR). After the evaluation of the selected sub-goal to the success value, the evaluation of the second sub-goal will begin (rules ConjL and ConjR).

Existing miniKanren implementations use various search methods. In our work, we have chosen a non-deterministic version of semantics in order to eliminate dependence on the details of a specific implementation. The downside of this solution is that for a specific program and a specific implementation of miniKanren, the

Semantic variables:

$$\mathfrak{S} = \mathfrak{s}_1, \mathfrak{s}_2, \dots$$

$$\Sigma, \Sigma' \dots \subset 2^{\mathcal{S}} \text{ (sets of allocated semantics variables)}$$

$$\langle \Sigma', \mathfrak{s} \rangle \leftarrow \underline{\texttt{new}}\ \Sigma,\ \Sigma' = \Sigma \cup \{\mathfrak{s}\},\ \mathfrak{s} \notin \Sigma \text{ (allocation of a new semantic variable)}$$

Values:

$$\mathcal{V} \mathrel{+}= \underline{\texttt{success}} \mid \mathfrak{s}$$

Context:

$$\mathcal{C} \mathrel{+}= \square \equiv e \mid v \equiv \square \mid \square \not\equiv e \mid v \not\equiv \square \mid \square \wedge e \mid e \wedge \square$$

State:

$$\langle \Sigma, \mathcal{S}, e, \sigma \rangle \text{ (set of allocated semantic variables, stack of contexts, expression, logical state)}$$

$$\langle \varnothing, \varepsilon, e, \iota \rangle \text{ (nitial state)}$$

Transition rules:

$$\langle \Sigma, \mathcal{S}, \underline{\texttt{fresh}}(x)\ e, \sigma \rangle \rightsquigarrow \langle \Sigma', \mathcal{S}, e[x \leftarrow \mathfrak{s}], \sigma \rangle,\ \langle \Sigma', \mathfrak{s} \rangle \leftarrow \underline{\texttt{new}}\ \Sigma \qquad [\textsc{Fresh}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \equiv e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \square \equiv e_2 : \mathcal{S}, e_1, \sigma \rangle \qquad [\textsc{UnifyL}]$$

$$\langle \Sigma, \mathcal{S}, v \equiv e, \sigma \rangle \rightsquigarrow \langle \Sigma, v \equiv \square : \mathcal{S}, e, \sigma \rangle \qquad [\textsc{UnifyR}]$$

$$\langle \Sigma, \mathcal{S}, v_1 \equiv v_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, \underline{\texttt{success}}, \sigma' \rangle,\ \textbf{unify}\,(\sigma, v_1, v_2) = \sigma' \qquad [\textsc{Unify}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \not\equiv e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \square \not\equiv e_2 : \mathcal{S}, e_1, \sigma \rangle \qquad [\textsc{DisEqL}]$$

$$\langle \Sigma, \mathcal{S}, v \not\equiv e, \sigma \rangle \rightsquigarrow \langle \Sigma, v \not\equiv \square : \mathcal{S}, e, \sigma \rangle \qquad [\textsc{DisEqR}]$$

$$\langle \Sigma, \mathcal{S}, v_1 \not\equiv v_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, \underline{\texttt{success}}, \sigma' \rangle,\ \textbf{diseq}\,(\sigma, v_1, v_2) = \sigma' \qquad [\textsc{DisEq}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \vee e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, e_1, \sigma \rangle \qquad [\textsc{DisjL}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \vee e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, e_2, \sigma \rangle \qquad [\textsc{DisjR}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \wedge e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, \square \wedge e_2 : \mathcal{S}, e_1, \sigma \rangle \qquad [\textsc{ConjStartL}]$$

$$\langle \Sigma, \mathcal{S}, e_1 \wedge e_2, \sigma \rangle \rightsquigarrow \langle \Sigma, e_1 \wedge \square : \mathcal{S}, e_2, \sigma \rangle \qquad [\textsc{ConjStartR}]$$

$$\langle \Sigma, \mathcal{S}, \underline{\texttt{success}} \wedge e, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, e, \sigma \rangle \qquad [\textsc{ConjL}]$$

$$\langle \Sigma, \mathcal{S}, e \wedge \underline{\texttt{success}}, \sigma \rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, e, \sigma \rangle \qquad [\textsc{ConjR}]$$

Figure 2.6 — Semantics of the relational extension

result of the computation may not coincide with that prescribed by the semantics. For example, in a particular implementation, a program may diverge, while non-deterministic semantics does terminate. However, in this case, the miniKanren

program or interpreter can be rewritten so that they converge according to this scenario.

Now let us consider the structure of the logical state and the semantics of unification and disequality constraint. This semantics in many aspects corresponds to the implementation of miniKanren, extended with the disequality constraint presented in [69], as well as standard approaches to the implementation of unification [70; 83]. Further on, we will use the following classic concepts:

- substitution (θ) that maps logical terms to semantic variables;
- application of the substitution θ to a term with $(t\,\theta)$ to replace semantic variables in term t in accordance with the substitution θ;
- composition of substitutions $(\theta\theta')$, the application of which to the term t corresponds to the sequential application of substitutions θ and θ';
- the most general substitution of two terms $(mgu\,(t_1,t_2))$, defining the most general substitution of θ, in which the terms $(t_1\,\theta)$ and $(t_2\,\theta)$ are syntactically equal.

The logical state of semantics consists of the following two components:

$$\sigma = (\theta,\Theta^-),$$

where θ is a substitution, and Θ^- is a set of negative substitutions describing disequality constraints that can potentially be violated. The initial state of semantics contains an indefinite substitution and an empty set:

$$\iota = (\bot,\varnothing).$$

The unification operation accepts two terms and a logical state as input parameters:

$$\mathbf{unify}\,(\sigma, t_1, t_2) = unify\,((\theta,\Theta^-), t_1, t_2).$$

Unification is conducted in several steps. First of all, it is necessary to evaluate the most general unifier for terms t_1 and t_2, taking into account the current substitution θ:

$$\rho = mgu\,(t_1\,\theta, t_2\,\theta).$$

If such a unifier ρ does not exist, unification ends in failure. Note that there are no rules in the semantics of the relational extension for this case, so the computation of a relational program also ends without an output. In case of successful completion

of unification, the substitution ρ must be checked for consistency with the disequality constraints represented by Θ^- (if the set Θ^- is empty, the check immediately succeeds).

Let's consider two terms

$$\begin{array}{rcl} t_l & = & (\mathfrak{s}_1,\ldots,\mathfrak{s}_k), \\ t_r & = & (\rho(\mathfrak{s}_1),\ldots,\rho(\mathfrak{s}_k)), \end{array}$$

where $\{\mathfrak{s}_i\} = dom\,(\rho)$. For each substitution $\theta^- \in \Theta^-$, compute $mgu\,(t_l\,\theta^-,t_r\,\theta^-)$. In this case, the following options are possible.

1. Unification ended in failure. It follows from this that the disequality constraint represented by θ^- can no longer be violated. In this case, we remove θ^- from Θ^- and continue the check with the next disequality constraint.
2. Unification was successful with an empty substitution as the result. This means that the disequality constraint represented by θ^- is violated. Consequently, the check is completed and the initial unification operation ends in failure.
3. Unification was successful with a non-empty substitution of θ'^-. In this case, in order not to violate the disequality constraint θ^-, it is necessary to check consistency with θ'^- in future unifications. Replace θ^- with θ'^- in Θ^- and continue the check with the next disequality constraint.

The result of a successful check of disequality constraints is a modified set of Θ'^-, which is the second and last component of the final logical state:

$$\mathbf{unify}\,((\theta,\Theta^-),t_1,t_2) = (\theta\rho,\Theta'^-).$$

The disequality constraint operation is executed in a similar way and also takes a pair of terms and a logical state as parameters:

$$\mathbf{diseq}\,(\sigma,t_1,t_2) = \mathbf{diseq}\,((\theta,\Theta^-),t_1,t_2).$$

1. Unification ended with a negative result, which means that the disequality constraint has already been met in the current substitution.
2. Unification was completed successfully with an empty substitution as the result. Therefore, the disequality constraint is violated.
3. Unification was successfully completed with a non-empty substitution θ'^-. This means that this substitution describes an disequality constraint that must hold in the future, so we add it to Θ^-.

The result of a successful evaluation of the disequality constraint is a (potentially) modified set of Θ'^{-} so it is necessary to update the logical state:

$$\textbf{diseq}\left((\theta,\Theta^{-}), t_1, t_2\right) = (\theta,\Theta'^{-}).$$

Finally, for a closed goal g and a logical state σ, we define $g \rightsquigarrow^{r} \sigma$ if $\langle\varnothing,\varepsilon,g,\iota\rangle \rightsquigarrow^{*} \langle\Sigma,\varepsilon,\underline{\texttt{success}},\sigma\rangle$ for some logic state Σ, where "$\rightsquigarrow^{*}$" is the reflexive-transitive closure of the relation "$\rightsquigarrow$".

It can be noticed that the relational extension typing rules add some interpreted types and symbols in relation to the type system of the source language. Thus, it can be expected that the relational extension inherits all its useful properties, such as **progress** and **type preservation** [84]. However, this is not true. Indeed, the only value for a goal is $\underline{\texttt{success}}$, but obviously not the goal succeeds (for example, $\texttt{A} \equiv \texttt{B}$ always fails). Thus, the relational extension lacks the progress property — there is a correctly typed non-value goal sometimes cannot make a step of semantics. However, this circumstance does not obstruct our work, since in any case the failure value for goals can be added to the language along with failure propagation rules.

2.3 Relational Conversion

In general, functional programs operate with high-order values, while miniKanren is limited to first-order unification. Therefore, not every functional program can be converted into a relational one with simple conversions. We will formulate a few constraints for source programs before introducing relational conversion.

Let us introduce the set of **ground types** $\mathcal{G}$ as follows:

$$\mathcal{G} = \alpha \mid T^{k}(g_1, \ldots, g_k).$$

Moreover, all expressions of ground type must be supplemented with the logical expression constructor $\uparrow$, which at the type level can be described using

the conversion $[\![\bullet]\!]^o$:

$$[\![\alpha]\!]^o = \alpha,$$
$$[\![T^k(g_1, \ldots, g_k)]\!]^o = (T^k([\![g_1]\!]^o, \ldots, [\![g_k]\!]^o)^o.$$

Informally speaking, a ground type value cannot contain higher-order subexpressions. Therefore, we will introduce the following three constraints for programs designed to be converted to a relational form.

1. Parameters of type constructors must be type variables.
2. Polymorphic equality and constructors can only be applied to ground type expressions.
3. Any pattern matching expression must be of ground type.

The first constraint requires that all algebraic types of the program be completely polymorphic. Because of this, the second constraint limits polymorphism for relational programs: all type variables contained in type constructors can only be replaced with ground types. Note that this condition is sufficient, but not necessary. The third restriction is introduced to simplify the representation of relational conversion. If a pattern matching has an non-ground type, it can still be converted to an equivalent expression containing only a pattern mapping of ground type using the η-extension:

$$\underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \ \rightarrow\ e_i\} \rightsquigarrow \lambda\ \bar{x}.\underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \{p_i \ \rightarrow\ e_i\,\bar{x}\},$$

where $\bar{x}$ is a vector of new variables not contained in expressions e, e_i, and p_i. Moreover, the implementation of the proposed relational conversion described in chapter 6 executes the η-extension for any pattern matching of an non-ground type. Note that this is the only case of using the source program type and executing the η extension during relational conversion.

The basic idea of relational conversion can be described at the type level: an expression of the source language with the type t is converted to an expression of a relational extension with the type $[\![t]\!]^t$, where conversion of the type $[\![\bullet]\!]^t$ is defined as follows:

$$[\![g]\!]^t = [\![g]\!]^o \rightarrow \mathfrak{G},$$
$$[\![t_1 \rightarrow t_2]\!]^t = [\![t_1]\!]^t \rightarrow [\![t_2]\!]^t.$$

In other words, an expression of ground type is converted to an unary function that accepts an expression of the corresponding logical type and returns a goal. This function matches the logical form of the original value with the passed parameter.

Since the argument can contain several occurrences of free variables, this function tries to match the corresponding subexpressions of the original expression to these variables. For example, the constant constructor `Nil` will be converted to the function $\lambda q.\ q \equiv \uparrow\texttt{Nil}$.

Now let us consider the relational conversion of an arbitrary expression t that satisfies the constraints described above. Let us denote this conversion as $[\![t]\!]^c$.

$$
\begin{array}{rcl}
[\![x]\!]^c & = & x \\
[\![\lambda\ x\ .\ e]\!]^c & = & \lambda\ x\ .\ [\![e]\!]^c \\
[\![f\ e]\!]^c & = & [\![f]\!]^c\ [\![e]\!]^c \\
[\![\underline{\texttt{let}}\ x\ = e_1\ \underline{\texttt{in}}\ e_2]\!]^c & = & \underline{\texttt{let}}\ x\ =\ [\![e_1]\!]^c\ \underline{\texttt{in}}\ [\![e_2]\!]^c \\
[\![\underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ = \lambda x.e_1\ \underline{\texttt{in}}\ e_2]\!]^c & = & \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ =\ [\![\lambda x.e_1]\!]^c\ \underline{\texttt{in}}\ [\![e_2]\!]^c
\end{array}
$$

The first five rules completely preserve the structure of the initial expression and apply conversions to all nested expressions. The essential part of the conversion consists in the conversion of various applications of constructors, pattern matching, and the polymorphic equality operator. Note that the fresh variable operator <u>`fresh`</u> in the following rules introduces a set of logical variables at once. This syntactic sugar is introduced to simplify the perception of rules and is similar to a set of operators <u>`fresh`</u>, introducing one variable at a time.

$$
[\![C^k(e_1, \ldots, e_k)]\!]^c \quad = \quad
\begin{array}{l}
\lambda\ q\ .\ \underline{\texttt{fresh}}\ (q_1 \ldots q_k) \\
\qquad ([\![e_1]\!]^c\ q_1)\ \ \wedge \\
\qquad \ldots \\
\qquad ([\![e_k]\!]^c\ q_k)\ \ \wedge \\
\qquad (q \equiv\ \uparrow C^n(q_1, \ldots, q_k))
\end{array}
$$

Converting constructor application, it is known that each expression e_i is of ground type. Therefore, the relational images corresponding to them are unary functions that return a goal as a result. For each expression e_i, create a fresh boolean variable using <u>`fresh`</u> and apply $[\![e_i]\!]$ to these variables to macth the evaluation results with the corresponding variables. The result of the conversion of constructor application is also an unary function that returns a goal. Therefore, we extend the expression with an abstraction for the variable q and unify this variable with the constructor applied to the corresponding logical variables. We also apply the logical constructor $\uparrow$ to bring it into compliance with the unification typing rule.

$$\left[\!\left[\begin{array}{l} \underline{\texttt{match}}\ e\ \underline{\texttt{with}} \\ \quad \{C_i^{n_i}(x_1^i, \ldots, x_{n_i}^i) \to e_i\} \end{array} \right]\!\right]^c = \begin{array}{l} \lambda\ q\ .\ \underline{\texttt{fresh}}\ (q_e) \\ \quad ([\![e]\!]^c\ q_e)\ \wedge \\ \quad \bigvee_i\ ((\underline{\texttt{fresh}}\ (q_1^i \ldots q_{n_i}^i) \\ \qquad\quad (q_e \equiv\ \uparrow C_i^{n_i}(q_1^i, \ldots, q_{n_i}^i))\ \wedge \\ \qquad\quad (\lambda\ x_1^i \ldots x_{n_i}^i\,.\,[\![e_i]\!]^c) \\ \qquad\qquad (\equiv q_1^i)\ \ldots\ (\equiv q_{n_i}^i)\ q \\ \quad\ )) \end{array}$$

The pattern matching conversion rule is organized in a similar way. The matched expression has a ground type, since it is mapped to constructors. Select a fresh variable q_e and associate it with the value of the matched expression, as in the case of constructor application. Then, for all branches, create a set of logical variables (in each branch for each pattern variable) and express pattern matching via unification using the appropriate constructor and the created logical variables. For the final step of the conversion, note that e_i is an expression with free variables corresponding to those contained in the original pattern. Therefore, we convert e_i, abstract the result with abstractions that close these variables and get a function that replaces these variables with the passed values. It remains to associate the variables q_j^i with the pattern variables in the converted expression $[\![e_i]\!]^c$. This is achieved by applying the resulting function to unary functions $(\equiv q_j^i)$. As a result, we get an unary function again, which we apply to the external resulting variable q.

$$[\![e_1 = e_2]\!]^c = \begin{array}{l} \lambda\ q\ .\ \underline{\texttt{fresh}}\ (q_1\, q_2) \\ \qquad [\![e_1]\!]^c\, q_1\ \wedge \\ \qquad [\![e_2]\!]^c\, q_2\ \wedge \\ \qquad ((q_1 \equiv\ q_2\ \wedge\ q \equiv\ \uparrow\underline{\texttt{true}})\ \vee \\ \qquad\ \ (q_1 \not\equiv\ q_2\ \wedge\ q \equiv\ \uparrow\underline{\texttt{false}}) \\ \qquad) \end{array}$$

The final rule, which is intended to convert the polymorphic equality operator, is based on the the same pattern: both arguments have a ground type, so we convert them into unary functions that return a goal as a result. Apply these functions to the fresh variables q_1 and q_2 and consider the two following cases. If these variables are equal, we associate constructor <u>`true`</u> with the resulting variable q. Otherwise, we will associate the constructor <u>`false`</u>. Note that this is the only case of using an disequality constraint in a relational conversion.

In conclusion, we will discuss a few notable properties of the relational conversion presented in this chapter. The first property is the complete preservation of expressions consisting only of variables, abstraction, application and `let` bindings. Therefore, many useful higher-order functions (for example, application, composition, fixed-point operator) are already relational and can be used in relational programs without modification.

The second property is compositionality [85]: the relational application image is an application of relational images. It is due to compositionality that the application operator remains unchanged during relational conversion. As a result, the latter supports separate application — a set of source programs can be divided into several parts and converted independently while maintaining the possibility of joint execution. In practice, this is conversion of a program divided into several files.

Finally, it is worth noting that the result of the relational conversion is executed deterministically in forward direction corresponding to the execution of the original program. Therefore, relational conversion leads to no more than a linear slowdown when executed in forward direction. In other words, a query that requires evaluating the last parameter of a relation when the remaining ground parameters exactly corresponds to the evaluation of the original function with the same parameters. A relational query with non-ground parameters will generally be evaluated nondeterministically, so its evaluation time increases exponentially with the size of the parameters. At the same time, it is not possible to compare performance with the corresponding source function, since a query with non-ground arguments cannot be compared with any call of the source function.

Chapter 3. Static and Partial Dynamic Correctness of Relational Conversion

Preserving program semantics, which is called conversion correctness, is an essential part of program conversion. This property guarantees that programs before and after conversion will be executed in the same way, i.e. they will produce the same output on the same input values.

In the previous chapter, we defined two semantics for both the source language and the relational extension: type inference semantics and execution semantics. Traditionally, type inference semantics is called static, since the result of its execution does not depend on specific parameters. In turn, execution semantics is called dynamic.

This chapter presents proofs of correctness of relational conversion for both static and dynamic semantics. We will prove static correctness of relational conversion (type preservation) and partial dynamic correctness (preservation of semantics of relational conversion). Note that dynamic correctness is partial, since there are queries for the relational image that cannot be reproduced in the source program.

3.1 Static Correctness of Relational Conversion

Static correctness is an important property both from a theoretical and practical point of view. From a theoretical point of view, it is a necessary requirement for confirming dynamic correctness, since it is not possible to preserve semantics in a typed language without a guarantee of typing correctness. From a practical point of view, static correctness is necessary to guarantee successful type checking when compiling a relational image. This section presents a proof of the theorem on the typing correctness of a converted program, provided that the typing of the original functional program is correct.

Theorem 1 (Static Correctness)**.** Let expression e be of type t in the source language. Then in the relational extension, after relational conversion, the image of this expression $[\![e]\!]^c$ will be of type $[\![t]\!]^t$.

Before proceeding to the proof of the theorem, it is necessary to consider how the types of variables of the contexts of the source language semantics and the relational extension semantics are related. To do this, we formulate the definition of *relational image of the context* of the source language semantics, which defines the conversion of types of the variables contained in the context of relational extension semantics. Furthermore, we formulate a lemma that conversion of variable types during relational conversion is in compliance with the application of the relational semantics context image. Therefore, we will confirm that all types are preserved correctly for all variables contained in the source program.

Definition 1. Let e be an expression of the source language, x be a node in the syntax tree of expression e, Γ be the context that occurs when the type of expression e is inferred in the node x. Then we will call $[\![\Gamma]\!] = \{(s : [\![t]\!]^t) \mid (s : t) \in \Gamma\}$ the *relational image* of the context Γ.

Lemma 1. Let e be an expression of the source language, x a node in the syntax tree of the expression e, $\Gamma \vdash e : t$ be the state during type inference of expression e in the node x, $\bar{\Gamma} \vdash \bar{e} : \bar{t}$ be the state during type inference of the converted expression $[\![e]\!]^c$ in the node $[\![x]\!]^c$, which is the image of the node x. Then $[\![\Gamma]\!] \subset \bar{\Gamma}$.

Proof. We will prove this lemma by induction in the length of the path from the root of the syntax tree of expression e to the node x.

In induction base case, the node x is the root of the syntax tree of the expression e. This node corresponds to the root of the syntax tree of the converted expression $[\![e]\!]^c$. At the first step of type inference, the context is empty. Hence $\Gamma = \varnothing$ and $\bar{\Gamma} = \varnothing$. Therefore, it is true that $[\![\Gamma]\!] = [\![\varnothing]\!] = \varnothing$. Since $\varnothing \subset \varnothing$, the result is $\varnothing = [\![\Gamma]\!] \subset \bar{\Gamma} = \varnothing$, which proves the base case of induction.

For the induction step, it is necessary to make sure that for each type inference rule of the source language, the statement $[\![\Gamma]\!] \subset \bar{\Gamma}$ is preserved for all antecedents, provided that the statement $[\![\Gamma]\!] \subset \bar{\Gamma}$ is true for their consequent.

Consider the case when one of the rules $\textsc{Bool}_T$, $\textsc{Eq}_T$, $\textsc{Constr}_T$ or $\textsc{App}_T$ is applied during type inference of the node x. Applying these rules, the context Γ will

not receive new variables. When type of node $[\![x]\!]^c$ is inferred, the context $\bar{\Gamma}$ can only be supplemented, so the relation $\Gamma \subset \bar{\Gamma}$ will be preserved.

If the rules VAR_T, ABS_T, LET_T and LETREC_T are applied, the nodes x and $[\![x]\!]^c$ have an identical operator as the root, because a relational conversion does not change an expression if it is a variable, an abstraction, or a let-binding. Therefore, the contexts Γ and $\bar{\Gamma}$ will be supplied with identical variables and the relation $\Gamma \subset \bar{\Gamma}$ will be preserved.

It remains to consider the application of the MATCH_T rule. In this case, the root of node x is a pattern matching and has the following structure: $\underline{\texttt{match}}\ e_0\ \underline{\texttt{with}}\ \{C_i^{k_i}(x_1^i, \ldots, x_{k_i}^i) \to e_i\}$. The corresponding node $[\![x]\!]^c$ will be relational, therefore, in accordance with the rule of relational conversion for pattern matching, we have the following:

$$
[\![x]\!]^c = \begin{array}{l}
\lambda\ q\ .\ \underline{\texttt{fresh}}\ (q_{e_0}) \\
\quad ([\![e_0]\!]^c\ q_{e_0})\ \wedge \\
\quad \bigvee_i\ ((\underline{\texttt{fresh}}\ (q_1^i \ldots q_{n_i}^i) \\
\qquad\quad (q_{e_0} \equiv\ \uparrow C_i^{n_i}(q_1^i, \ldots, q_{n_i}^i))\ \wedge \\
\qquad\quad (\lambda\ x_1^i \ldots x_{n_i}^i\, .\, [\![e_i]\!]^c) \\
\qquad\qquad (\equiv q_1^i)\ \ldots\ (\equiv q_{n_i}^i)\ q \\
\quad)).
\end{array}
$$

When inferring the node types x of the expression e for each subexpression e_i, the context Γ will be supplied with variables x_j^i, which will be of types $t_j^{C_i}$, which are the types of parameters of the constructor $C_i^{n_i}$. It is necessary to make sure that for the node $[\![x]\!]^c$ of the converted expression $[\![e]\!]^c$ for each subexpression $[\![e_i]\!]^c$, the context $\bar{\Gamma}$ will also be replenished with variables x_j^i with converted types $[\![t_j^{C_i}]\!]^t$. As we can see, each expression $[\![e_i]\!]^c$ is abstracted across all variables x_j^i, so these variables will replenish the context of $\bar{\Gamma}$ when inferring types. It remains to check that they will match the correct type $[\![t_j^{C_i}]\!]^t$.

First of all, we need to determine the types of all logical variables q_j^i. These types are defined by expressions $(q_{e_0} \equiv \uparrow C_i^{n_i}(q_1^i, \ldots, q_{n_i}^i))$, where these variables are parameters of constructors $C_i^{n_i}$. Therefore, the types of variables q_j^i are equal to $[\![t_j^{C_i}]\!]^o$. It follows directly from the pattern matching conversion rule that the abstraction $(\lambda x_1^i \ldots x_{n_i}^i\, .\, [\![e_i]\!]^c)$ is applied to the set of parameters $(\equiv q_1^i), \ldots, (\equiv q_{n_i}^i), q$. Thus, each variable x_j^i will have a corresponding type of expression $(\equiv q_j^i)$. Since type $[\![t_j^{C_i}]\!]^o$ corresponds to variable q_j^i, then for expression $(\equiv q_j^i)$ we obtain

type $[\![t_j^{C_i}]\!]^o \to \mathfrak{G}$. This exactly corresponds to the required type, because according to the rule of conversion of ground type we have that $[\![t_j^{C_i}]\!]^t = [\![t_j^{C_i}]\!]^o \to \mathfrak{G}$. Thus, the induction step is proven. □

This lemma confirms the fact that during type inference of the relational image, all variables of the source program will be assigned correct types.

The theorem itself is easily proven in accordance with the proof scheme of Lemma 1 by applying structural induction using Lemma 1 when proving the induction base case.

This theorem confirms that the introduced relational conversion preserves the correctness of typing, but it bypasses the issue of preserving the semantics of the converted program. The next section is devoted to solving this problem.

3.2 Partial Dynamic Correctness of Relational Conversion

The second theorem proven in this chapter confirms the partial dynamic correctness or, in other words, the preservation of semantics during the relational conversion of a correct program of the source language. It is important to note that we prove *partial* dynamic correctness, which guarantees the preservation of the semantics of the relational image evaluated in the *forward direction*. In other words, if the source program contains the application of some function f to the set of arguments $e_1, \ldots, e_n$, then the preservation of semantics is guaranteed for the application of $[\![f]\!]_c\ [\![e_1]\!]_c\ \ldots\ [\![e_n]\!]_c\ q$, where q is a fresh variable. However, in the case of a relational image, more complex applications are possible: any parameter can be replaced by a fresh variable either completely or partially by replacing the subexpression of the parameter with a fresh variable. It is impossible to match any original functional program to such a relational program as its inverse image due to the strict direction of functional program evaluation.

Theorem 2 (Partial Dynamic Correctness)**.** Let the first-order expression e be of type t, and let there exist a first-order value v such that $e \leadsto^f v$. Then $\underline{\texttt{fresh}}(x)\ ([\![e]\!]^c\ x) \leadsto^r (\theta,\varnothing)$, and $\theta(\mathfrak{s}) = v$, where $\mathfrak{s}$ is the semantic variable associated with x at the first step of evaluation.

First of all, note that the set of negative substitutions is empty. Addition to this set is possible only when a disequality constraint is executed. This construction can be executed in a converted program only if the source program contains a syntactic comparison applied to its parameters. During the execution of a converted relational program in the forward direction (the last parameter is a fresh variable, the remaining parameters are fully defined), if the disequality constraint is executed, both parameters are closed, which leads to the immediate resolution of the disequality constraint without addition to the initially empty set of negative substitutions.

Note that this theorem cannot be proved by induction in the length of derivation, since each *application* of the source program contains a function as a left subexpression. And this function is obviously not a first-order expression. This constraint could be removed if it were possible to prove a generalization of $p \rightsquigarrow^f f \Rightarrow [\![p]\!]^c \rightsquigarrow^r [\![f]\!]^c$ for an arbitrary p of any type. This statement, however, turned out to be false, since the expression `C ((λ x.x) A)` can be presented as a counterexample.

The reason for this is that during the conversion, constructors, pattern matching and syntactic comparisons are *functionalized*, and, consequently, the evaluation order of the relational image changes in comparison with the original functional program. Thus, it is necessary to solve this problem for the proof.

To eliminate the differences in the evaluation order of the source program and the relational image, we developed a modified semantics of the functional language, the evaluation order in which is close to the evaluation order of the relational image. This semantics was called *deferred*, since it postpones the evaluation of constructors, pattern matching, and syntactic comparison. This semantics can be derived from the original functional semantics in two steps. First, it is necessary to consider an truncated version of the original functional semantics, which treats constructors, pattern matching, and syntactic comparisons as computed values. Then define deferred semantics as iterative application of truncated semantics to the parameters of these new values (parameters of constructors or syntactic comparison, as well as the scrutinee of pattern matching).

Further, we note that if a first-order expression evaluates to some value in the original semantics, then it is also evaluated to the same value in deferred semantics. This property is based on the following observations:

– both semantics have standard properties of **progress** and **type preservation** [84];
– the Church-Rosser property [84; 86] for λ-calculus is true for both semantics;
– deferred semantics uses a subset of the rules of the original semantics.

Now we will proceed to the proof of the theorem. We will use the simulation method [87; 88] for the source program in deferred semantics by a relational image in relational semantics. But beforehand, we need to formulate several lemmas and definitions.

Definition 2. Define two sets of contexts of the source language: *set of functional contexts* (1) and *set of atomic contexts* (2):

$$C_f = \square\ e \mid v\ \square \mid \underline{\texttt{let}}\ x\ \texttt{=}\ \square\ \underline{\texttt{in}}\ e, \tag{1}$$

$$C_g = \underline{\texttt{match}}\ \ \square\ \ \underline{\texttt{with}}\ \{p_i{\rightarrow}e_i\} \mid C^n(\bar{v},\square,\bar{e}) \mid \square\texttt{=e} \mid \texttt{v=}\square. \tag{2}$$

Note that the sets of functional and atomic contexts are disjuncts, and their union is exactly equal to the set of all contexts of the source language.

Lemma 2. Let $\langle \mathcal{S}, e\rangle$ be an arbitrary state of a sequence of evaluations in deferred semantics. Then the following holds: $\mathcal{S} = C_f^* C_g^*$.

In other words, during evaluation in deferred semantics, the stack of contexts can be divided into two (possibly empty) segments: all atomic contexts are located below all functional ones.

This lemma is easily proven by induction in the length of derivation.

Definition 3. Define two sets of expressions of the source language: *set of functional expressions* (3) and *set of atomic expressions* (4):

$$E_f = e_1\, e_2 \mid \lambda x.e \mid \mu f.\lambda x.e \mid \underline{\texttt{let}}\ x\ \texttt{=}\ e_1\ \underline{\texttt{in}}\ e_2 \mid \underline{\texttt{let}}\ \underline{\texttt{rec}}\ f\ \texttt{=}\ \lambda x.e_1\ \underline{\texttt{in}}\ e_2, \tag{3}$$

$$E_g = (e_1 = e_2) \mid \underline{\texttt{match}}\ e\ \underline{\texttt{with}}\ \texttt{\{}p_i\ \rightarrow\ e_i\texttt{\}.} \mid \texttt{C}^k\ \ \texttt{(}e_1 \ldots e_k\texttt{)}. \tag{4}$$

Note that the sets of functional and atomic expressions are disjuncts, and their union is exactly equal to the set of all expressions of the source language.

Definition 4. For an arbitrary substitution θ, *extended relational conversion* of an expression of the source language $[\![\bullet]\!]_\theta$ is defined as follows:

$$\begin{aligned} [\![f]\!]_\theta &= [\![f]\!]^c, \\ [\![v]\!]_\theta &= (\lambda x.x \equiv \mathfrak{s}), \text{if } \theta(\mathfrak{s}) = v. \end{aligned}$$

Here θ is a substitution, f is an arbitrary functional expression, and v is an arbitrary first-order value in the source semantics (i.e., constructor composition).

Note that different cases in this definition are not disjunctive, and in the second case there may be more than one variable with the requested property, so the extended conversion defines a set of relational expressions.

Lemma 3. Let f and e be arbitrary expressions in the source language, and θ an arbitrary substitution. Then the following is true:

$$[\![f[x \leftarrow e]]\!]_\theta = [\![f]\!]_\theta[x \leftarrow [\![e]\!]_\theta].$$

In this case, equality must be interpreted as the equality of two sets.

This lemma is easily proven by structural induction.

Definition 5. For an arbitrary substitution of θ, we define *conversion of functional context* $[\![\bullet]\!]_\theta$ as follows:

$$\begin{array}{rcl} [\![\square\, e]\!]_\theta & = & \square\, [\![e]\!]_\theta, \\ [\![v\, \square]\!]_\theta & = & [\![v]\!]_\theta\, \square, \\ [\![\underline{\texttt{let}}\ x\ =\ \square\ \ \underline{\texttt{in}}\ e]\!]_\theta & = & \underline{\texttt{let}}\ \ x\ =\ \square\ \ \underline{\texttt{in}}\ \ [\![e]\!]_\theta. \end{array}$$

Here e is an arbitrary functional expression, v is a λ-abstraction.

Definition 6. For arbitrary semantic variables $\mathfrak{s}_1$, $\mathfrak{s}_2$ and an arbitrary substitution θ, we define *conversion of atomic contexts* $[\![\bullet]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2}$ as follows:

$$\begin{array}{l} [\![C^k(v_1, \ldots, v_{i-1}, \square, e_{i+1}, \ldots, e_k)]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} = \\ \qquad \square \wedge \\ \qquad ([\![e_{i+1}]\!]_\theta\ \mathfrak{s}'_{i+1}) \wedge \\ \qquad \ldots \\ \qquad ([\![e_k]\!]_\theta\ \mathfrak{s}'_k) \wedge \\ \qquad (\mathfrak{s}_2 \equiv \uparrow C^k(\mathfrak{s}'_1, \ldots, \mathfrak{s}'_{i-1}, \mathfrak{s}_1, \mathfrak{s}'_{i+1}, \ldots, \mathfrak{s}_k)), \text{if } \theta(\mathfrak{s}'_j) = v_j,\ j < i; \end{array}$$

$$\begin{array}{rcl} [\![\square = e]\!]_\theta^{\mathfrak{s}_1\mathfrak{s}_2} & = & \square \wedge \\ & & ([\![e]\!]_\theta\ \mathfrak{s}') \wedge \\ & & (((\mathfrak{s}_1 \equiv \mathfrak{s}') \wedge (\mathfrak{s}_2 \equiv \uparrow \underline{\texttt{true}})) \vee \\ & & ((\mathfrak{s}_1 \not\equiv \mathfrak{s}') \wedge (\mathfrak{s}_2 \equiv \uparrow \underline{\texttt{false}}))); \end{array}$$

$$\begin{aligned}\llbracket v = \Box \rrbracket_\theta^{\mathfrak{s}_1\mathfrak{s}_2} \quad &= \quad \Box \wedge \\ &\quad (((\mathfrak{s}' \equiv \mathfrak{s}_1) \wedge (\mathfrak{s}_2 \equiv \uparrow \underline{\texttt{true}})) \vee \\ &\quad ((\mathfrak{s}' \not\equiv \mathfrak{s}_1) \wedge (\mathfrak{s}_2 \equiv \uparrow \underline{\texttt{false}}))), \text{if } \theta(\mathfrak{s}) = v;\end{aligned}$$

$$\begin{aligned}&\llbracket \underline{\texttt{match}}\ \Box\ \ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i,\ \ldots, y_{n_i}^i)\ \rightarrow e_i\} \rrbracket_\theta^{\mathfrak{s}_1\mathfrak{s}_2} = \\ &\quad \Box \wedge \bigvee\nolimits_i \\ &\quad (\underline{\texttt{fresh}}\ (s_1^i \ldots s_{n_i}^i) \\ &\qquad (\mathfrak{s}_1 \equiv \uparrow C_i^{n_i}(s_1^i, \ldots, s_{n_i}^i)) \\ &\qquad (\lambda y_1^i. \ldots \lambda y_{n_i}^i. \llbracket e_i \rrbracket_\theta)\ (\equiv s_1^i) \ldots (\equiv s_{n_i}^i)\ \mathfrak{s}_2).\end{aligned}$$

Here $\mathfrak{s}'$ and $\mathfrak{s}_i'$ are arbitrary semantic variables, v_i are arbitrary values in the source language, and e_i are arbitrary expressions in the source language. We will also require that θ be undefined for all specified semantic variables, unless the opposite is explicitly stated.

Definition 7. For an arbitrary substitution θ, an arbitrary semantic variable $\mathfrak{s}_m$ and a functional expression e, we define *conversion of stack of contexts* $\llbracket \bullet \rrbracket_\theta^{e,\mathfrak{s}_m}$ as follows:

$$\llbracket f_n \ldots f_1 g_m \ldots g_1 \rrbracket_\theta^{e,\mathfrak{s}_m} = \begin{cases} \llbracket g_m \rrbracket_\theta^{\mathfrak{s}_m\mathfrak{s}_{m-1}} \ldots \llbracket g_1 \rrbracket_\theta^{\mathfrak{s}_1\mathfrak{s}_0}, & \text{if } n = 0 \text{ and } e \in E_g \\ \llbracket f_n \rrbracket_\theta \ldots \llbracket f_1 \rrbracket_\theta (\Box\, \mathfrak{s}_m) \llbracket g_m \rrbracket_\theta^{\mathfrak{s}_m\mathfrak{s}_{m-1}} \ldots \llbracket g_1 \rrbracket_\theta^{\mathfrak{s}_1\mathfrak{s}_0}, & \text{otherwise.} \end{cases}$$

Here $\mathfrak{s}_0 \ldots \mathfrak{s}_{m-1}$ are arbitrary unique semantic variables.

Definition 8. For an arbitrary substitution θ and an arbitrary semantic variable $\mathfrak{s}_m$, we define *simulation conversion* $\llbracket \bullet \rrbracket_\theta^{\mathfrak{s}_m}$ for a source language expression as follows:

$$\begin{aligned}\llbracket e_1 = e_2 \rrbracket_\theta^{\mathfrak{s}_m} \quad &= \quad (\llbracket e_1 \rrbracket_\theta\ \mathfrak{s}_1') \wedge \\ &\quad (\llbracket e_2 \rrbracket_\theta\ \mathfrak{s}_2') \wedge \\ &\quad (((\mathfrak{s}_1' \equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \uparrow \underline{\texttt{true}})) \vee \\ &\quad ((\mathfrak{s}_1' \not\equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \uparrow \underline{\texttt{false}})));\end{aligned}$$

$$\begin{aligned}\llbracket v = e \rrbracket_\theta^{\mathfrak{s}_m} \quad &= \quad (\llbracket e \rrbracket_\theta\ \mathfrak{s}_2') \wedge \\ &\quad (((\mathfrak{s}_1' \equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \uparrow \underline{\texttt{true}})) \vee \\ &\quad ((\mathfrak{s}_1' \not\equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \uparrow \underline{\texttt{false}}))), \text{if } \theta(\mathfrak{s}_1') = v;\end{aligned}$$

$$\begin{aligned}\llbracket v_1 = v_2 \rrbracket_\theta^{\mathfrak{s}_m} \quad &= \quad (((\mathfrak{s}_1' \equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \uparrow \underline{\texttt{true}})) \vee \\ &\quad ((\mathfrak{s}_1' \not\equiv \mathfrak{s}_2') \wedge (\mathfrak{s}_m \equiv \uparrow \underline{\texttt{false}}))), \text{if } \theta(\mathfrak{s}_j') = v_j;\end{aligned}$$

$$\llbracket C^k(v_1, \ldots, v_{i-1}, e_i, \ldots, e_k)\rrbracket_\theta^{\mathfrak{s}_m} =$$
$$(\llbracket e_i \rrbracket_\theta\ \mathfrak{s}'_i) \wedge$$
$$\ldots$$
$$(\llbracket e_k \rrbracket_\theta\ \mathfrak{s}'_k) \wedge$$
$$(\mathfrak{s}_m \equiv \uparrow C^k(\mathfrak{s}'_1, \ldots, \mathfrak{s}'_k)), \text{if } \theta(\mathfrak{s}'_j) = v_j, j < i;$$

$$\llbracket C^k(v_1, \ldots, v_k)\rrbracket_\theta^{\mathfrak{s}_m} = (\mathfrak{s}_m \equiv\ C^k(\mathfrak{s}'_1, \ldots, \mathfrak{s}'_k)), \text{if } \theta(\mathfrak{s}'_j) = v_j;$$

$$\llbracket C^k(v_1, \ldots, v_k)\rrbracket_\theta^{\mathfrak{s}_m} = (\mathfrak{s}_m \equiv\ \mathfrak{s}'), \text{if } \theta(\mathfrak{s}') = C^k(v_1, \ldots, v_k);$$

$$\llbracket \underline{\texttt{match}}\, e\ \ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i,\ \ \ldots, y_{n_i}^i)\ \to e_i\}\rrbracket_\theta^{\mathfrak{s}_m} =$$
$$\llbracket e \rrbracket_\theta\ \mathfrak{s}'\ \wedge\ \bigvee_i$$
$$(\underline{\texttt{fresh}}\ (s_1^i \ldots s_{n_i}^i)$$
$$(\mathfrak{s}' \equiv \uparrow C_i^{n_i}(s_1^i, \ldots, s_{n_i}^i))$$
$$(\lambda y_1^i. \ldots \lambda y_{n_i}^i. \llbracket e_i \rrbracket_\theta)\ (\equiv s_1^i) \ldots (\equiv s_{n_i}^i)\ \mathfrak{s}_m);$$

$$\llbracket \underline{\texttt{match}}\, v\ \ \underline{\texttt{with}}\ \{C_i^{n_i}(y_1^i,\ \ \ldots, y_{n_i}^i)\ \to e_i\}\rrbracket_\theta^{\mathfrak{s}_m} =$$
$$\bigvee_i$$
$$(\underline{\texttt{fresh}}\ (s_1^i \ldots s_{n_i}^i)$$
$$(\mathfrak{s}' \equiv \uparrow C_i^{n_i}(s_1^i, \ldots, s_{n_i}^i))$$
$$(\lambda y_1^i. \ldots \lambda y_{n_i}^i. \llbracket e_i \rrbracket_\theta)\ (\equiv s_1^i) \ldots (\equiv s_{n_i}^i)\ \mathfrak{s}_m), \text{if } \theta(\mathfrak{s}') = v.$$

Here, all $\mathfrak{s}'$ and $\mathfrak{s}'_i$ are arbitrary semantic variables, e is arbitrary expression of the source language, and v is an arbitrary value for the semantics of the source language. We will also require that θ be undefined for all specified semantic variables, unless the opposite is explicitly stated.

Definition 9. $\langle \mathcal{S}, e \rangle$ be a state in deferred semantics, and $\left\langle \Sigma, \hat{\mathcal{S}}, \hat{e}, (\theta, \varnothing) \right\rangle$ be a state in relational semantics. We will call these states *connected* if there exists a semantic variable q_m such that the following holds:

- $\hat{\mathcal{S}} \in \llbracket \mathcal{S} \rrbracket_\theta^{e, \mathfrak{s}_m}$;
- $\hat{e} \in \begin{cases} \llbracket e \rrbracket_\theta^{\mathfrak{s}_m} & , \text{ if } e \in E_g \text{ and } \mathcal{S} \cap C_f = \varnothing \\ \llbracket e \rrbracket_\theta & , \text{ otherwise;} \end{cases}$
- Σ contains all semantic variables from $\hat{e}$, $\hat{\mathcal{S}}$, and θ.

Lemma 4. Let $v = \texttt{C}^k(v_1, \ldots, v_k)$ be a value in terms of semantics of the source functional language. Then for arbitrary Σ, $\mathcal{S}$, θ, $\hat{v} \in [\![v]\!]_\theta$, and semantic variable $\mathfrak{s}$ such that $\mathfrak{s} \notin dom(\theta)$, (5) or (6) is true.

$$\langle \Sigma, \mathcal{S}, (\hat{v}\,\mathfrak{s}), (\theta, \varnothing)\rangle \rightsquigarrow^* \left\langle \Sigma', \mathcal{S}, \mathfrak{s} \equiv \texttt{C}^k(\mathfrak{s}'_1, \ldots, \mathfrak{s}'_k), (\theta', \varnothing)\right\rangle \text{ and } \theta'(\mathfrak{s}'_i) = v_i, \quad (5)$$

$$\langle \Sigma, \mathcal{S}, (\hat{v}\,\mathfrak{s}), (\theta, \varnothing)\rangle \rightsquigarrow \langle \Sigma, \mathcal{S}, \mathfrak{s} \equiv \mathfrak{s}', (\theta, \varnothing)\rangle \text{ and } \theta(\mathfrak{s}') = v. \quad (6)$$

This lemma is easily proven by induction in the height of v.

Lemma 5. Let $s = \langle \mathcal{S} = g_m \ldots g_1, e\rangle$ be a state in deferred semantics, g_i be atomic contexts, e be a first-order expression, θ be some substitution, and Σ be the set of selected semantic variables, $\mathfrak{s}_m$ be some semantic variable, $\hat{\mathcal{S}} \in [\![\mathcal{S}]\!]_\theta^{e, \mathfrak{s}_m}$, $\hat{e} \in [\![e]\!]_\theta$ and Σ contain all semantic variables from $\hat{\mathcal{S}}$ and θ. Then there is such a sequence of steps in the semantics of the relational extension that the following holds:

$$\left\langle \Sigma, \hat{\mathcal{S}}, (\hat{e}\,\mathfrak{s}_m), (\theta, \varnothing)\right\rangle \rightsquigarrow^* \hat{s},$$

where s and $\hat{s}$ are connected. This is true under the assumption that Σ contains all semantic variables from $\hat{\mathcal{S}}$ and θ.

The proof of this lemma can be easily conducted by considering the cases for expression e using Lemma 4.

Lemma 6. Let $s_1 \to s_2$ be a single evaluation step in deferred semantics, and $\hat{s_1}$ be such a state in relational semantics that s_1 and $\hat{s_1}$ are connected. Then there is a sequence of steps in relational semantics $\hat{s_1} \rightsquigarrow^* \hat{s_2}$ such that s_2 and $\hat{s_2}$ are connected.

This lemma is proven by analyzing the cases for s_1 and constructing the simulation relation [87; 88] using Lemmas 3, 4, and 5.

Lemma 7. Let $s_0 = \langle \varnothing, \varepsilon, \underline{\texttt{fresh}}(x)\ \ ([\![e]\!]^c\, x), \iota\rangle$ be an initial state in relational semantics. Then there exists a sequence of steps $s_0 \rightsquigarrow^* \hat{s}$ such that the initial state in the deferred semantics of $\langle \varepsilon, e\rangle$ and $\hat{s}$ are connected.

The proof of this lemma immediately follows from Lemma 5.

All the necessary definitions and lemmas are now formulated. Thus, we are now able to prove the partial dynamic correctness theorem. Let e be a first-order expression in the source language, which is evaluated as the value $v = \texttt{C}^k(v_1, \ldots, v_k)$

in the source semantics. Then, after evaluation of e in deferred semantics, the same value will be obtained as follows: $\langle \varepsilon, e\rangle \rightarrow^* \langle \varepsilon, v\rangle$.

Then by Lemma 7 we have

$$\langle \varnothing, \varepsilon, \underline{\texttt{fresh}}\ (x)\ (\llbracket e \rrbracket^c\ x), \iota\rangle \rightsquigarrow^* \hat{s},$$

in which $\langle \varepsilon, e\rangle$ and $\hat{s}$ are connected. By Lemma 6, there exists a state $\hat{s'}$ in relational semantics such that $\hat{s} \rightsquigarrow^* \hat{s'}$, in which $\langle \varepsilon, v\rangle$ and $\hat{s'}$ are connected. By definition of the simulation relation, $\hat{s'}$ can be represented as (7) or (8):

$$\left\langle \Sigma, \varepsilon, \mathfrak{s}_0 \equiv \uparrow\texttt{C}^k(\mathfrak{s}'_1, \ldots, \mathfrak{s}'_k), (\theta, \varnothing)\right\rangle, \theta(\mathfrak{s}'_i) = v_i; \tag{7}$$

$$\langle \Sigma, \varepsilon, \mathfrak{s}_0 \equiv \mathfrak{s}', (\theta, \varnothing)\rangle, \theta(\mathfrak{s}') = v. \tag{8}$$

Here $\mathfrak{s}_0$ is the first semantic variable introduced in Σ and $\mathfrak{s}_0 \notin dom(\theta)$. In both cases, it remains to take one last step in relational semantics, which completes this proof.

Chapter 4. The miniKanren Semantics with Dynamic Control of Conjunct Order

Dynamic control of operator order in a program is a classic approach to program optimization not only in logic programming. This approach simplifies development by eliminating the need to select the optimal order of operators manually. It also turns out to be useful in the absence of a static optimal order. For example, in case of multiple executions of a single function, the optimal order of operators in the body of this function may vary. Dynamic control of evaluation order helps to avoid creating multiple copies of this function that differ only in operator order.

In classic relational programming, the order of evaluation of disjuncts and conjuncts is determined by the semantics of disjunction and conjunction operators. And if disjunction determines the evaluation order of independent branches, effectively controlling the evaluation order of disjuncts by interleaving, conjunction, in its turn, is responsible for the operator evaluation order in each separate branch and strictly fixes the evaluation order.

This chapter presents the formal semantics of relational miniKanren with classic left-biased conjunction. We also describe angelic semantics which makes it possible to define the concept of fairness for a relational conjunction. We define an important from a practical point of view class of deterministic semantics of miniKanren, parameterized by the unfolding predicate for choosing the optimal order of conjuncts. Finally, a specific form of this predicate is defined, based on well quasi-ordering, which turns parameterized semantics into fair semantics.

4.1 Classic Left Biased Conjunction in miniKanren

In this section, we will analyze the features of the classic conjunction in miniKanren using several examples of relational definitions and specifications. Let us start with the definition of $\texttt{append}^o$, which is the concatenation of two lists.

$$\underline{\texttt{let}}\ \underline{\texttt{rec}}\ \texttt{append}^o = \lambda\ x\ y\ xy\ .$$
$$(x \equiv \uparrow\texttt{[]}\ \wedge\ xy \equiv y)\ \vee$$

$\underline{\texttt{fresh}}$ $(e\ xs\ xys)$

$x \ \equiv \uparrow(e\ ::\ xs)\ \wedge$

$xy \equiv \uparrow(e\ ::\ xys)\ \wedge$

$\texttt{append}^o\ t\ y\ ty$

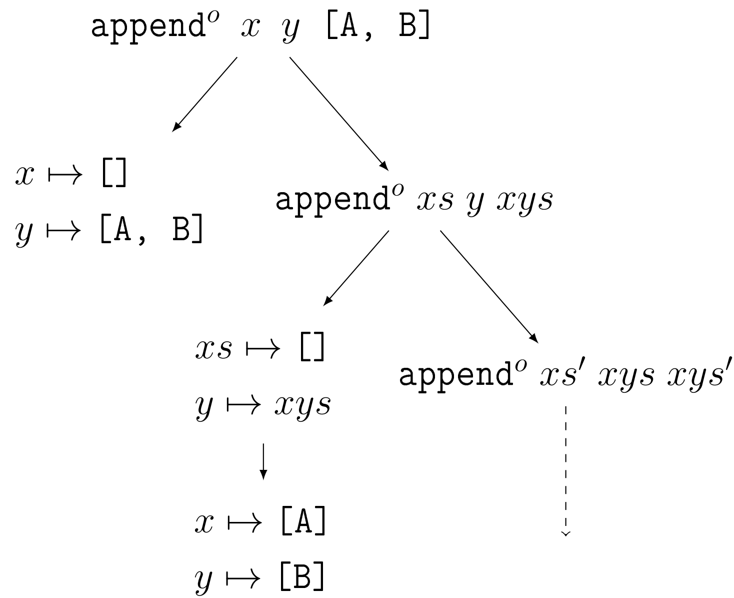
The three parameters x, y and xy correspond to three such lists that the concatenation of x and y is equal to xy. The relation is defined as follows: if x is empty (`[]`), then xy must be equal to y; otherwise we will split x into head e and tail xs, then xy will be equal to $e\ :\ xys$, where xys is a concatenation of xs and y obtained by recursively calling the same relation (here we use ‘`[]`"and ‘`:`"as constructor names).

In the context of this definition, the goal $\texttt{append}^o$ `[A]` `[B]` x evaluates to the substitution $x \mapsto$ `[A, B]`, which is the expected result. However, since concatenation works in both directions, the same relation can be used to solve other problems as well. For example, evaluating the goal $\texttt{append}^o$ `[A]`x `[A,B]` will return a substitution $x \mapsto$ `[B]`, which corresponds to getting the suffix of a list. At the same time, the goal $\texttt{append}^o$ x y `[A, B]` returns a set of the following three substitutions up to order:

$$\begin{array}{ll} x \mapsto \texttt{[]} & y \mapsto \texttt{[A, B]} \\ x \mapsto \texttt{[A]} & y \mapsto \texttt{[B]} \\ x \mapsto \texttt{[A, B]} & y \mapsto \texttt{[]}. \end{array}$$

In other words, the same definition of a relation can be used to search in different “directions” depending on the distribution of free variables.

In all the above examples, when evaluating the goal, a finite set of answers is given in a finite time. However, this is not always the case. For example, if we put a recursive call in the definition $\texttt{appendo}^o$ *first* in conjunction, then the goal $\texttt{append}^o xy$ `[A, B]` will *diverge* after returning all the answers. An illustration of this behavior is shown in the following figure:



Here, the left and right branches of the nodes correspond to the evaluation of the first and second `append`^{o} disjuncts, respectively. We start with the first disjunct of `append`^{o} , which immediately gives us the first answer $x \mapsto$ `[]`$, y \mapsto$ `[A,B]`. In the second disjunction, we encounter a conjunction and begin to evaluate its left conjunct, which is the application of `append`^{o} to *free* variables xs, y and xys. After substituting the body of the relation `append`^{o} to the position of applying this relation, we again encounter a disjunction. Its left disjunct gives us the substitution $xs \mapsto$ `[]`$, y \mapsto xys$, which, after returning from a recursive call and evaluating the remaining conjuncts, gives the second answer $x \mapsto$ `[A]`$, y \mapsto$ `[B]`. The second disjunction, however, is again a conjunction starting with the recursive application of `append`^{o} to new free variables. It is clear that this branch will never stop growing, and after restoring the third answer, the evaluation will diverge. This example demonstrates the well-known phenomenon of *refutational incompleteness* [40] of this particular implementation of `append`^{o}. It can be proven [89] that the initial implementation (with a recursive call placed last in conjunction) is refutational complete, but only for *linear* goals, i.e. such goals in which no variable occurs in arguments more than once.

Placing relation calls at the end of a set of conjuncts is a well-known trick in relational programming that often helps to improve the performance or even convergence of goal computation. This trick, however, does not help in all cases. For example, it does not help when there is more than one call in the set of conjuncts. Consider the `revers`^{o} relation, which matches an arbitrary list with a list containing the same elements in reverse order.

$$\texttt{revers}^o = \lambda\, x\, y \;\; . \;\; (x \equiv \uparrow\texttt{[]} \;\wedge\; y \equiv \uparrow\texttt{[]}) \;\vee$$
$$\underline{\texttt{fresh}}\;(e\; xs\; ys)$$
$$x \equiv \uparrow(\,e\;:\;xs) \;\wedge$$
$$\texttt{revers}^o\; xs\; ys \;\wedge$$
$$\texttt{append}^o\; ys\; \uparrow(e\;:\;\uparrow\texttt{[]})\; y$$

In this relation, there are two calls in the same set of conjuncts: $\texttt{revers}^o$ and $\texttt{append}^o$; thus, it is impossible to put them both last. With this particular order of calls, the goal $\texttt{revers}^o$ `[A, B, C]` x converges, but with the reverse order, the same goal diverges after finding the answer. Moreover, the reverse order negatively affects the effectiveness of the answer evaluation. At the same time, the goal $\texttt{revers}^o$ x `[A , B, C]` demonstrates symmetric behavior: diverges for a given order of conjuncts, and converges for the opposite.

An important purpose of miniKanren is to provide a purely declarative way to define executable relational specifications that would work equally well regardless of the "direction" of the search. In particular, since we expect conjunction and disjunction to be commutative and associative, the order of conjuncts and disjuncts should not have a noticeable effect on the behavior of the specification. As these examples show, this has not yet been fully achieved. What is even more interesting, all these cases are manifestations of the same drawback of the classical relational search strategy: *left bias* during conjunction execution. We can demonstrate this with the following artificial example. First, we define two relations: $\texttt{div}^o$ and $\texttt{fail}^o$.

$$\texttt{div}^o \;\; = \lambda\, x \;\; . \;\; \texttt{div}^o\; \texttt{x}$$
$$\texttt{fail}^o = \lambda\, x \;\; . \;\; \uparrow\texttt{A} \equiv \uparrow\texttt{B}$$

In both cases, x is a "non-essential" parameter that does not affect the result of evaluating these relations in any way. It is clear that $\texttt{div}^o$ diverges without answers, and $\texttt{fail}^o$ is evaluated as an empty set of answers in one step. Thus, one would expect that the conjunction of these relations would fail in one step. Indeed, the conjunction $\texttt{fail}^o$`_` $\wedge\texttt{div}^o$`_` completes in one step with an empty set of answers. However, the equivalent conjunction $\texttt{div}^o$`_` $\wedge\texttt{fail}^o$`_` diverges without an answer (here the underscore ("`_`") denotes an arbitrary non-essential parameter). The explanation of this is trivial: during the evaluation of a conjunction, we first evaluate its left conjunct until the first answer is found. In the first case, we immediately get an empty stream, and it is not possible to evaluate the entire conjunction. In the second case, we never get an answer; at the same time, we always have

the remaining steps to evaluate the left conjunct, and this means that the whole search diverges. This conjunction behavior not only leads to discrepancies in some important cases, but also greatly affects performance by clogging the search tree with infinite branches which do not produce any results. In some cases, this can be avoided by changing the order of conjuncts in the definition of the relation in accordance with a certain evaluation direction. However, as shown in the work [49], solving a number of applied problems requires that the same definition holds in different directions simultaneously.

In some cases, there is no static order of conjuncts at which the evaluation converges, however, with a dynamic change in the order of conjuncts, convergence can be achieved. For example, consider a query ($\underline{\texttt{repeat}}^o$ A q $\wedge$ $\texttt{repeato}^o$ B q) on the free variable q for the relation $\underline{\texttt{repeat}}^o$, which checks that all the elements of the list are l are equal to term e.

$$
\begin{aligned}
\underline{\texttt{repeat}}^o = \lambda\, e\, l\ .\ & (l \equiv \uparrow\texttt{[]}) \ \vee \\
& \underline{\texttt{fresh}}\ (ls) \\
& \quad l \equiv \uparrow(e\ :\ ls) \ \wedge \\
& \quad \underline{\texttt{repeat}}^o\ e\ ls
\end{aligned}
$$

This query requires that all elements of the list l, on the one hand, to be equal to the nullary constructor A, and on the other, to the constructor B, which differs from A. Only an empty list satisfies this requirement. It is easy to see that when evaluating this query in semantics with a left-biased conjunction, the evaluation will diverge with any order of arguments. However, if we partially evaluate the first conjunct $\underline{\texttt{repeat}}^o$ A q by postponing the nested call $\underline{\texttt{repeat}}^o$, and then evaluate the second conjunct $\underline{\texttt{repeat}}^o$ B q in the context of the data obtained from the first conjunct, then the evaluation will end in finding the only correct answer. In other words, in this case we will not only find a solution for a given query, but also prove its uniqueness.

The problems of left-biased conjunction described above are relevant for the relational conversion described in chapter 2. This is due to the fact that the result of this conversion is a relational program, and its effectiveness directly depends on the chosen conjunction execution strategy. Moreover, with relational conversion, the optimal order of conjuncts cannot be determined due to insufficient information contained in the original functional program. Therefore, “manual” editing of each relational query of this program is required for effective execution

of the automatically obtained resulting program in semantics with a left-biased conjunction.

Summing up, we note that in the general case, there is no optimal static order of conjuncts with a left-biased conjunction evaluation strategy. We propose another evaluation strategy that dynamically postpones the evaluation of some branches using dynamic analysis of the internal properties of the evaluated relations.

4.2 Angelic Semantics and Fairness

In this section, we introduce *angelic* operational semantics for the miniKanren language. Instead of a fixed conjunct evaluation order, this semantics selects the next conjunct to be evaluated nondeterministically, thereby enumerating all possible orders. Thus, if a relational query diverges in angelic semantics, then there is no convergent order of evaluation for it.

The semantics we propose is based on the *unfolding* of the application of relations. At each step, in the current state of the program, some application is nondeterministically selected and unfolded. This process continues as long as there are applications remaining in the current state. If the state is empty at some step, then evaluation is complete.

The results of this (and the following) section are based on the article [89], which presents two semantics for the miniKanren language: denotational and operational. Denotational semantics is set-theoretic and is close to the least Herbrand model [55]. Operational semantics defines a relational search for a solution with interleaving in the form of a transition system with labels [90], and is informally described in the previous section. The correctness and completeness of interleaving search in operational semantics have been proven using Coq [91] — an interactive tool for proving theorems. The article [89] also found that for the operational semantics of miniKanren, some syntactic transformations preserve the semantics of the transformed relational program and do not affect the set of evaluated answers. These transformations will play an important role in justifying the fairness of the semantics that we present. The next section is devoted to this issue.

As the first element of angelic semantics, we define a set of the following *semantic* variables:

$$\mathcal{A} = \{\alpha_0, \alpha_1, \ldots\}.$$

Note that semantic variables are linearly ordered, which makes it possible to use them in a deterministic way when executing a relational program. Then we will naturally define a set of *semantic terms* $\mathcal{T}_{\mathcal{A}}$ consisting of constructors and semantic variables. We will also use the following notation for the set of free variables in terms and goals: $\mathcal{FV}(\bullet)$. Substitution of a *semantic* term t in place of a *syntactic* variable x in terms and goals will be denoted as follows: $\bullet[x \leftarrow t]$.

Next, we define the set of *states of semantics* $\mathfrak{S}$ as follows:

$$\begin{array}{rcll} \mathfrak{S} & = & \times & \text{(final state)} \\ & | & \mathfrak{S}^o & \text{(intermediate state)} \\ \mathfrak{S}^o & = & \langle \theta, i, r^* \rangle & \text{(leaf state)} \\ & | & \mathfrak{S}^o \oplus \mathfrak{S}^o & \text{(disjunction)} \\ r & = & R^k(t_1, \ldots, t_k) & \text{(relation application)} \end{array}$$

The intermediate semantics state is a disjunction tree with leaves $\langle \theta, i, r^* \rangle$, where θ is a substitution of semantic variables into semantic terms, i is the number of the first unoccupied semantic variable, and r^* is list of uses of relations $R^k(t_1, \ldots, t_k)$ (possibly empty), where R^k is the name of the relation, and t_i are semantic terms. Informally, the state of semantics is some representation of the goal during its execution, when the results of evaluating conjuncts are collected in substitutions, and the residual goal is represented in disjunctive normal form (DNF). If the list of applications of relations in the leaf state is empty, then the substitution is the answer. The final state corresponds to the end of relational query execution.

We introduce two auxiliary functions for working with states. The **union** function combines two states:

$$\begin{array}{rcl} \textbf{union}(\times, S) & = & S, \\ \textbf{union}(S, \times) & = & S, \\ \textbf{union}(S_1, S_2) & = & S_1 \oplus S_2. \end{array}$$

The **push** function is used to restore the state after performing a step of one relation application:

$$\begin{array}{rcl} \textbf{push}(_, \times) & = & \times, \\ \textbf{push}(c, S_1 \oplus S_2) & = & \textbf{push}(c, S_1) \oplus \textbf{push}(c, S_2), \\ \textbf{push}(\rho \square \pi, \langle \theta, i, \sigma \rangle) & = & \langle \theta, i, \rho\sigma\pi \rangle. \end{array}$$

The first argument of the **push** function is a list of relation applications that contains a *hole* “□”. This hole marks the place where the unfolded relation application was located. The second parameter is the state representing the unfolding result. Thus, the **push** function propagates the unfolding context through the resulting local state.

Next, we are going to describe the semantics for a one-step unfolding of relation application. Informally, we take an application of the relation symbol to some semantic terms and replace these terms for the corresponding arguments in the body of the relation definition, and then perform all the unifications and transform the rest of the body into DNF. As a result, we get some semantics state. Note that we do not unfold relation applications inside the relation body. Let us give a formal definition of a one-step unfolding in terms of big step semantics “$\Rightarrow$” using the following rule:

$$\frac{R = \lambda \bar{x}.b \qquad \langle \theta, i, \varepsilon \rangle \vdash b\,[\bar{x} \leftarrow \bar{t}] \rightsquigarrow S}{\langle \theta, i \rangle \vdash R\,(\bar{t}) \Rightarrow S} \quad [\textsc{Unfold}]$$

Hereafter $\bar{\bullet}$ denotes a vector of terms or variables. In the conclusion of this rule, a pair consisting of a substitution and the number of the first unoccupied semantic variable is used as an environment. In the premise of the rule, small step semantics is used in the form of a relation “$\rightsquigarrow$”, which uses a previously defined state of semantics as an environment. This relation describes local evaluations inside the body of the unfolded application. The definition of this relation is given in Fig. 4.1.

The [End] rule extends the final state to all other evaluations. The rules [UnifyFail] and [UnifySucc] encode the unification stages: if there is a most general unifier in the θ substitution for these terms, then we will update this substitution. Otherwise, the evaluation will result in the final state. The [App] rule saves the relation application by adding it to the list of applications in the current leaf state. The [Fresh] rule corresponds to the allocation of a fresh semantic variable. We take the first unallocated semantic variable and replace it with a freshly linked syntactic one; we also increase the number of the first unallocated variable by one. The rules [DisjGoal] and [DisjState] describe the evalution of disjunction. The first rule evaluates both disjuncts and combines the resulting states using the **union** function. The second rule handles the case when the current environment is a disjunction. As in the first case, we perform independent evaluations and combine the results. Finally, the [Conj] rule describes the evaluation of conjunction. In this case, we first evaluate the left sub-goal, obtaining state S, and then evaluate the

right goal in the context of S, getting the final result S'. It should be noted that since no unfoldings are conducted, the process always converges to some state.

We define the angelic semantics itself using unfolding semantics, see Fig. 4.2. This semantics is defined in terms of a transition system with labels over a set of states. The set of semantics labels is defined as $\mathcal{L} = \circ \mid \theta$, where '$\circ$"denotes the absence of an answer, and θ is an answer in the form of a substitution.

The [Answer] rule corresponds to a situation when there is not a single relation application left in the considered state. In this case, we return the current substitution as an answer. If the current state is a list with a non-empty list of relation applications, then we nondeterministically select one for unfolding in accordance with the [ConjUnfold] rule. After unfolding we create a new state by placing the remaining list of applications in the resulting state. Finally, if the current state is a disjunction, then we perform the semantics step for the left disjunction S_1. If the result is the final state, then we set S_2 as the residual state according to the rule [Disj]. Otherwise, according to the [DisjStep] rule, we construct a new residual state by forming a new disjunction, in which the right initial substate is placed to the left, and the residual state of evaluating the left initial substate is

$$\times \vdash g \rightsquigarrow \times \qquad [\textsc{End}]$$

$$\frac{\nexists\, mgu\,(t_1, t_2, \theta)}{\langle \theta, i, c\rangle \vdash t_1 \equiv t_2 \rightsquigarrow \times} \qquad [\textsc{UnifyFail}]$$

$$\frac{\theta' = mgu\,(t_1, t_2, \theta)}{\langle \theta, i, c\rangle \vdash t_1 \equiv t_2 \rightsquigarrow \langle \theta', i, c\rangle} \qquad [\textsc{UnifySucc}]$$

$$\langle \theta, i, c\rangle \vdash R\,(\bar{t}) \rightsquigarrow \langle \theta, i, cR\,(\bar{t})\rangle \qquad [\textsc{App}]$$

$$\frac{\langle \theta, i+1, c\rangle \vdash g\,[x \leftarrow \alpha_i] \rightsquigarrow S}{\langle \theta, i, c\rangle \vdash \underline{\texttt{fresh}}(x)\; g \rightsquigarrow S} \qquad [\textsc{Fresh}]$$

$$\frac{\langle \theta, i, c\rangle \vdash g_1 \rightsquigarrow S_1 \qquad \langle \theta, i, c\rangle \vdash g_2 \rightsquigarrow S_2}{\langle \theta, i, c\rangle \vdash g_1 \vee g_2 \rightsquigarrow \mathbf{union}\,(S_1, S_2)} \qquad [\textsc{DisjGoal}]$$

$$\frac{S_1 \vdash g \rightsquigarrow S_3 \qquad S_2 \vdash g \rightsquigarrow S_4}{S_1 \oplus S_2 \vdash g \rightsquigarrow \mathbf{union}\,(S_3, S_4)} \qquad [\textsc{DisjState}]$$

$$\frac{\langle \theta, i, c\rangle \vdash g_1 \rightsquigarrow S \qquad S \vdash g_2 \rightsquigarrow S'}{\langle \theta, i, c\rangle \vdash g_1 \wedge g_2 \rightsquigarrow S'} \qquad [\textsc{Conj}]$$

Figure 4.1 — Small step semantics of local evaluations

$$\langle \theta, i, \varepsilon \rangle \xrightarrow{\theta} \times \qquad \text{[ANSWER]}$$

$$\frac{\langle \theta, i \rangle \vdash c \Rightarrow S}{\langle \theta, i, \rho c \pi \rangle \xrightarrow{\circ} \mathbf{push}\,(\rho \Box \pi, S)} \qquad \text{[CONJUNFOLD]}$$

$$\frac{S_1 \xrightarrow{\alpha} \times}{S_1 \oplus S_2 \xrightarrow{\alpha} S_2} \qquad \text{[DISJ]}$$

$$\frac{S_1 \xrightarrow{\alpha} S_1' \qquad S_1' \neq \times}{S_1 \oplus S_2 \xrightarrow{\alpha} S_2 \oplus S_1'} \qquad \text{[DISJSTEP]}$$

Figure 4.2 — miniKanren angelic semantics

placed to the right. With the help of this permutation, the interleaving method is implemented, which is an important feature of relational search. Note that these two disjunction rules repeat the corresponding rules in [89].

To apply angelic semantics to the specification $D_1 D_2 \ldots D_k \diamond g$, we need to replace all free variables in g with semantic variables and convert them to the initial state $S^0\,(g)$:

$$\langle \Lambda, n, \varepsilon \rangle \vdash g\,[x_0 \leftarrow \alpha_0, \ldots, x_{n-1} \leftarrow \alpha_{n-1}] \rightsquigarrow S^0\,(g).$$

Here we assume that $\mathcal{FV}\,(g) = \{x_0, \ldots, x_{n-1}\}$, Λ is an empty substitution. The definitions of the specifications are used later, at the stages of the unfolding of applications in the state $S^0\,(g)$.

Angelic semantics preserves deterministic left-to-right interleaving for classic miniKanren search and processes conjunction nondeterministically, describing all possible scenarios of execution. For example, the goal $(\texttt{fail}^o\ _\ \wedge\ \texttt{div}^o_)$ from the previous section will converge in accordance with angelic semantics if we consider the branch of its evaluation in which the left conjunct was unfolded:

$$\frac{\dfrac{\langle \Lambda,\ 1, \varepsilon \rangle \vdash A \equiv B \rightsquigarrow \times}{\langle \Lambda, 1 \rangle \vdash \texttt{fail}^o\,\alpha_0 \Rightarrow \times}}{\langle \Lambda, 1,\ \texttt{fail}^o\,\alpha_0 \wedge \texttt{div}^o\,\alpha_0 \rangle \xrightarrow{\circ} \times}$$

Indeed, the unfolding $\texttt{fail}^o$ will take us to the final state in one step. At the same time, if we decide to unfold only the right conjunct, then the evaluation will diverge:

$$\frac{\dfrac{\langle \Lambda,\ 1, \varepsilon \rangle \vdash \texttt{div}^o\,\alpha_0 \rightsquigarrow \langle \Lambda,\ 1, \texttt{div}^o\,\alpha_0 \rangle}{\langle \Lambda, 1 \rangle \vdash \texttt{div}^o\,\alpha_0 \Rightarrow \langle \Lambda,\ 1, \texttt{div}^o\,\alpha_0 \rangle}}{\langle \Lambda,\ 1,\ \texttt{fail}^o\,\alpha_0 \wedge \texttt{div}^o\,\alpha_0 \rangle \xrightarrow{\circ} \langle \Lambda,\ 1,\ \texttt{fail}^o\,\alpha_0 \wedge \texttt{div}^o\,\alpha_0 \rangle}$$

Since the states before and after the unfolding `div`^{o} are the same, this step will be repeated an infinite number of times.

The following lemma confirms the correspondence between the denotational semantics of the miniKanren language from [89] and angelic semantics.

Lemma 8 (Correctness and completeness)**.** Angelic semantics is correct and complete.

Its completeness follows directly from the completeness of deterministic semantics proven in the paper [89]. Indeed, any inference in deterministic semantics can be reproduced in angelic semantics. The proof of correctness repeats the proof of the correctness of operational semantics in [89].

Angelic semantics allows us to formally define what we can call the *fairness* of a conjunction evaluation strategy. First, we define the concept of *convergence.*

Definition 10 (Convergence)**.** Goal *gconverges* (denoted as $g \Downarrow$) if there is a inference sequence from the initial state g to the final state, i.e. $\langle S^0\,(g), \times \rangle \in \rightarrow^*$.

Now we have everything we need to formally define the concept of *fairness.*

Definition 11 (Fairness)**.** Correct and complete semantics in the form of a transition system of with labels is *fair* if the goal g converges to the final state whenever $g \Downarrow$.

In the next section, we will consider fair and deterministic semantics for the miniKanren language.

4.3 General miniKanren Semantics with Selection Predicate

In this section we present deterministic fair semantics for the miniKanren language. There is nothing complicated in this problem from a theoretical point of view: since the interleaving method ensures the completeness of the search during disjunction evaluation, interleaving during conjunction evaluation will ensure its fairness. In other words, it is enough to move the “focus” of evaluation from one conjunct to another after completing one step (or any finite number of steps) to achieve the goal. Alas, in practice, this approach leads to a critical decrease in

performance. The problem lies in guessing the right moment to pause conjunct evaluation: if the conjunct is "going to" to produce an answer, then its evaluation should be continued. Thus, we are looking for a certain dynamic criterion that would identify the right time to suspend the evaluation of a conjunct, taking into account the internal properties of the executed program.

Note that exactly the same problem occurs in the area of *metacomputing*. In supercompilation [92] symbolic execution of a program can lead to a potentially infinite unfolding. To cope with this difficulty, there is a technique of *generalization* [93], which is used to guarantee the convergence of the process. However, both premature and delayed generalizations are undesirable, because then the supercompilation method leads to the construction of inefficient programs. In practice, supercompilation generalized based on the concept of *well quasi-ordering* [94; 95] has confirmed its applicability.

Definition 12. A well quasi-ordering on Σ is such a pre-order "$\trianglelefteq$" that in an arbitrary infinite sequence of $x_1, x_2, \ldots$ elements of Σ there will be elements x_i and x_j such that $i < j$ and $x_i \trianglelefteq x_j$.

From a practical point of view, well quasi-ordering can help to detect the divergence of a certain sequence of computational steps. In our approach, we use this idea to get fair semantics. First, we define a very general deterministic semantics, equipped with a specific predicate that determines the order of switching between conjuncts. Then we prove that under certain predicate requirements, the semantics becomes fair. Finally, we present a specific predicate based on a specific well quasi-ordering and show that it satisfies these requirements.

Let us introduce a set of *extended* states $\mathfrak{R}$:

$$
\begin{array}{rcll}
\mathcal{H} & = & \langle \theta, r \rangle^* & \text{unfolding history} \\
\mathfrak{R} & = & \times & \text{final state} \\
 & | & \mathfrak{R}^o & \text{non-final state} \\
\mathfrak{R}^o & = & \langle \theta, i, \langle r, \mathcal{H} \rangle^* \rangle & \text{leaf state with unfolding history} \\
 & | & \mathfrak{R}^o \oplus \mathfrak{R}^o & \text{disjunction}
\end{array}
$$

This definition is supplemented by the concept of *unfolding history*. The unfolding history is a list of pairs of substitutions and applications of relations (possibly empty). In the leaf state, each relation application is now equipped with the unfolding history that led to this application.

$$\frac{\bigvee_{j=1}^{n} h_j \neq \varepsilon \qquad \bigwedge_{j=1}^{n} \neg\mathcal{P}\left(\theta, r_j, h_j\right)}{\langle \theta, i, (r_1, h_1) \ldots (r_n, h_n)\rangle \xrightarrow{\circ}_p \langle \theta, i, (r_1, \varepsilon) \ldots (r_n, \varepsilon)\rangle} \quad [\textsc{ConjClear}]$$

$$\frac{\bigwedge_{j=1}^{k-1} \neg\mathcal{P}\left(\theta, r_j, h_j\right) \qquad \mathcal{P}\left(\theta, r_k, h_k\right) \qquad \langle \theta, i\rangle \vdash r_k \Rightarrow R}{\begin{array}{l}\langle \theta, i, (r_1, h_1) \ldots (r_{k-1}, h_{k-1})(r_k, h_k)\pi\rangle \xrightarrow{\circ}_p \\ \qquad \mathbf{push}\left((r_1, h_1) \ldots (r_{k-1}, h_{k-1})\Box\pi, \mathbf{set}\left(R,\ (\theta, r_k) : h_k\right)\right)\end{array}} \quad [\textsc{ConjUnfold}^*]$$

Figure 4.3 — Generalized semantics of the miniKanren language, parameterized by choice predicate $\mathcal{P}$

The **union** and **push** functions introduced above retain their definition for extended states, since the presence of history does not change their behaviour. In addition, we need a new function **set**, which takes a normal state and a history and constructs an extended state by attaching the history to each relation application in this state:

$$\begin{array}{rcl}\mathbf{set}\left(\times, _\right) & = & \times, \\ \mathbf{set}\left(S_1 \oplus S_2, h\right) & = & \mathbf{set}\left(S_1, h\right) \oplus \mathbf{set}\left(S_2, h\right), \\ \mathbf{set}\left(\langle \theta, i, r_1 \ldots r_n\rangle, h\right) & = & \langle \theta, i, (r_1, h) \ldots (r_n, h)\rangle .\end{array}$$

Our semantics (see Fig. 4.3, the transition relation is denoted by "$\rightarrow_p$") is parameterized by the predicate $\mathcal{P}(\theta, r, h)$, where θ is a substitution, r is a relation application, and h is an unfolding history. Informally speaking, $\mathcal{P}(\theta, r, h)$ determines whether it is desirable to unfold the application of r. The semantics itself completely repeats the one-step unfolding relation "$\Rightarrow$" and the rules [Answer], [Disj] and [DisjStep] from the definition of angelic semantics. A one-step unfolding returns a usual, incomplete state, while the rules process the enriched states (but syntactically these rules retain their full form). However, the rule for handling the conjunction [ConjUnfold] is replaced by two other rules: [ConjClear] and [ConjUnfold*]. If $\mathcal{P}$ is true for at least one relation application, we apply [ConjUnfold*] and unfold the leftmost such application by setting the history of the obtained applications using the **set** function. If the predicate is false for all relation applications and there is at least one non-empty unfolding history, then we apply the rule [ConjClear], which deletes the history for each application, replacing it with an empty history ε.

Parameterizing with a choice predicate produces a whole family of semantics, and not all elements of this family are good. For example, if the predicate always remains false, then no steps can be performed from the state $\langle \Lambda, i, (r, \varepsilon) \rangle$, which endangers completeness. The following lemma provides the necessary completeness condition.

Lemma 9. If it is true that $\forall\, \theta, r \ :\ \mathcal{P}(\theta, r, \varepsilon)$, then for any intermediate state R there exists a state R' such that $R \xrightarrow{\alpha}_p R'$.

Proof. Let us prove this statement by structural induction in R.

The case of $R = R_1 \oplus R_2$ is covered by rules [Disj] and [DisjStep].

The case of $R = \langle \theta, i, \varepsilon \rangle$ is covered by the [Answer] rule.

The case of $R = \langle \theta, i, (r_1, h_1) \ldots (r_n, h_n) \rangle$ provided $\bigvee_{j=1}^{n} \mathcal{P}(\theta, r_j, h_j)$ is covered by [ConjUnfold*].

The case of $R = \langle \theta, i, (r_1, h_1) \ldots (r_n, h_n) \rangle$ under the conditions $\bigvee_{j=1}^{n} h_j \neq \varepsilon$ and $\bigwedge_{j=1}^{n} \neg\mathcal{P}(\theta, r_j, h_j)$ is covered by the [ConjClear] rule.

The only remaining case is $R = \langle \theta, i, (r_1, h_1) \ldots (r_n, h_n) \rangle$ provided $\bigwedge_{j=1}^{n} h_j = \varepsilon$ and $\bigwedge_{j=1}^{n} \neg\mathcal{P}(\theta, r_j, h_j)$. It follows, in particular, that $\neg\mathcal{P}(\theta, r_1, \varepsilon)$, but by the lemma $\forall\, \theta, r \ :\ \mathcal{P}(\theta, r, \varepsilon)$, which is impossible.

□

All the predicates that we will consider next trivially satisfy the conditions of Lemma 9.

We can get various existing semantics by selecting a predicate. For example, by choosing an identical truth as a predicate, we get semantics with a left-biased conjunction. If a predicate limits the size of history, i.e.

$$\mathcal{P}_N(\theta, r, h) = \mathit{length}(h) \leqslant N,$$

then we get a semantics that sequentially unfolds the application of relations from left to right to some depth of $N > 0$. This technique resembles *bottom-avoiding streams* [40], which are an implementation of potentially infinite lists based on a specific data structure called “ferns” [96], which is designed to postpone divergent evaluations.

In the context of this work, *well quasi-ordering predicates* are an important class of predicates, which we will consider in the next section.

4.4 Fair miniKanren Semantics by Well Quasi-Ordering

Let “$\trianglelefteq$” be a well quasi-ordering on the set of pairs of substitutions and relation applications. Then we define a well quasi-ordering predicate as follows:

$$\mathcal{P}_{\trianglelefteq}(\theta, r, h) = \nexists \langle \theta_0, r_0 \rangle \in h \; : \; \langle \theta_0, r_0 \rangle \trianglelefteq \langle \theta, r \rangle \, .$$

This class of predicates ensures that every relation application contained in the state will eventually be unfolded after a finite number of steps. This statement is proven in chapter 5.

As discussed earlier, creating a fair semantics is not difficult if performance issues are not taken into account. From this point of view, the whole construction of fair semantics with the help of a well quasi-ordering does not provide any benefits unless a specific predicate of a well quasi-ordering which will help obtain an effective implementation is presented. This well quasi-ordering, firstly, should not require computational costs and, secondly, it should provide a good prediction of conjunct divergence. For example, *homeomorphic embedding* [97], often used as a criterion for stopping the program unfolding in the supercompilation method, in our case both requires significant computational costs and ineffectively predicts conjunct divergence.

Now we can present a well quasi-ordering predicate which will provide us with a practically important version of fair semantics.

First, we define the “$\leqslant_h$” relation on tuples of semantic terms.

Definition 13. Let $\langle t_1^1, \ldots, t_n^1 \rangle$ and $\langle t_1^2, \ldots, t_n^2 \rangle$ be tuples of semantic terms. Then the following statements are equivalent:

$$\langle t_1^1, \ldots, t_n^1 \rangle \leqslant_h \langle t_1^2, \ldots t_n^2 \rangle \quad \Leftrightarrow \quad \forall i \; : \; \mathit{height}\,(t_i^1) \leqslant \mathit{height}\,(t_i^2)$$

The “$\leqslant_h$” relation compares terms by their height. It requires at least one term of the left tuple to be strictly smaller than the corresponding term from the right tuple. The remaining left terms should not be longer than the corresponding terms in the right tuple.

Lemma 10. The “$\leqslant_h$” relation is a well quasi-ordering.

Proven by induction in the sum of term heights.

Now we can define "$\leqslant_{sr}$" on the set of pairs of substitutions and relation applications. Preliminarily, for each relation, we identify the set of its structural-recursive arguments. For some relation R, an argument is structurally recursive if the value of the argument decreases structurally with each recursive call. This is a simple syntactic property of a relation that can be checked statically for each relation of a relational program before evaluation.

Definition 14. Let θ_1, θ_2 be substitutions, $r_1 = R\,(t^1_1, \ldots, t^1_n)$ and $r_2 = R\,(t^2_1, \ldots, t^2_n)$ be arbitrary applications of relation R, and $j_1, \ldots, j_k$ be numbers of structurally recursive arguments in R. Then we define relation "$\leqslant_{sr}$" for applications of relation R in the context of their corresponding substitutions. Applications r_1 and r_2 satisfy $(\theta_1, r_1) \leqslant_{sr} (\theta_2, r_2)$ if their structurally recursive arguments satisfy the condition. $(t^1_{j_1}\theta_1, \ldots, t^1_{j_k}\theta_1) \leqslant_h (t^2_{j_1}\theta_2, \ldots, t^2_{j_k}\theta_2)$

Lemma 11. The "$\leqslant_{sr}$" relation is a well quasi-ordering.

The proof follows directly from Lemma 10.

It is easy to see that structural recursion was not actually used in this proof; in fact, "$\leqslant_{sr}$" remains well quasi-ordered, even if we select all parameters (or arbitrary ones). The choice of such a definition lies in the fact that structural recursion has demonstrated the best results in practice; in other words, it is a good heuristic for choosing a well quasi-ordering.

Chapter 5. Conjunction Fairness in miniKanren Semantics Equipped with a Well Quasi-Ordering Choice Predicate

This chapter presents a formal proof of conjunction fairness in the semantics of the miniKanren language with a procedure for dynamically controlling the order of conjuncts in the case of using a well quasi-ordering choice predicate.

First of all, we will simplify further reasoning, noting the dependence of convergence of a state and its leaves and replenishment of the substitution during unfolding of relation applications.

5.1 Convergence of State Leaves of Angelic Semantics and Semantics with a Quasi-Ordering Choice Predicate

We need to connect two semantics: angelic semantics, presented in Section 4.2, and fair semantics equipped with a well quasi-ordering choice predicate, presented in Section 4.3. We also intend to prove that in both semantics an arbitrary goal converges or diverges simultaneously.

Both semantics are represented as transition systems of with the same alphabet of labels and tree-like states, in which internal nodes correspond to disjunctions, and leaves contain ordered conjunctions of applications of relations, a substitution and the number of the first fresh semantic variable. Moreover, both semantics use an identical set of rules corresponding to the case of disjunction execution. Also, both semantics are correct and complete with respect to the denotational semantics of the miniKanren [89] language. Completeness means that in a state no leaf can be excluded from consideration — each leaf will be considered in a finite number of steps and, consequently, some relation application will be unfolded in each leaf. This allows us to abstract from specific forms of states and instead, treat them as sets of leaves. The following lemma justifies this transition.

Lemma 12 (leaf convergence)**.** Let s be a state with a set of leaves $\{\omega_i\}$. Then s converges (in angelic and fair semantics) if and only if every leaf ω_i converges.

Proof. The lemma follows from a more general fact. Let $s \to s_1 \to s_2 \to \ldots$ be a (finite or infinite) sequence of inference in angelic or fair semantics (we may consider only angelic semantics, since semantics equipped with a well quasi-ordering predicate is a special case of angelic semantics). Then the following is true:

- each state s_i can be represented as $s_i\,[w_1, \ldots, w_k]$, where $\{w_j\}$ is a disjunct set of subtrees of s_i and contains all its leaves in the order corresponding to left-to-right traversal of s_i;
- each angelic semantics step can be represented as $s_i\,[\ldots w_{j-1}, w_j, w_{j+1}, \ldots] \to s_{i+1}\,[\ldots w'_{j-1}, w_j^1, \ldots, w_j^m, w'_{j+1} \ldots]$, where a set of leaves (possibly empty) $\{w_j^1, \ldots, w_j^m\}$ is obtained from the unfolding of some relation application contained in leaf w_j and other relation applications from leaf w_j using **push**, and the set of leaves $\{w'_1, \ldots, w'_{j-1}, w'_{j+1}, \ldots, w'_k\}$ is a permutation of the set $\{w_1, \ldots, w_{j-1}, w_{j+1}, \ldots, w_k\}$;
- for each leaf w_j, you can select the sequence of its inference $\omega_k \to \cdots \to w_j$, the beginning of which will be some leaf ω_k of state s.

Let us prove this fact by induction in the path length between states s and s_i. For the induction base case, all leaves of state s are exactly equal to the set $\{\omega_i\}$, the elements of which satisfy all the requirements.

Let by induction assumption there be an intermediate state $s_i\,[w_1, \ldots, w_k]$, all leaves of which satisfy the requirements. The application of the disjunction rules will pass the state s_i from the root to some leaf (if the state does not contain disjunctions, then the whole case degenerates into $s_i = w_1$ and the statement becomes trivial) and, therefore, will come to a particular leaf w_j. One of the uses in the w_j leaf will be unfolded according to the rule for conjunction. Then the following state can be represented as

$$s_{i+1}\,[\ldots w'_{j-1}, w_j^1, \ldots, w_j^m, w'_{j+1} \ldots],$$

where the leaf w_j will be replaced by the leaves of the w_j unfolded result, supplemented by the rest of the relation applications from the leaf w_j that have not been unfolded.

In other words, any state evaluation can be decomposed into independent evaluations of its leaves, and vice versa. Therefore, the convergence of a state guarantees the convergence of each leaf, just as the convergence of each leaf guarantees the convergence of the entire state. $\square$

5.2 Preserving Convergence of Angelic Semantics

The following observation concerns the "commutativity" of sequential unifications. Let $p_{1,2}, q_{1,2}$ be some terms. Then the following statement holds:

$$mgu\,(p_1, p_2) \cdot mgu\,(q_1, q_2) = mgu\,(q_1, q_2) \cdot mgu\,(p_1, p_2).$$

Indeed, the unification sequences $mgu\,(p_1, p_2) \cdot mgu\,(q_1, q_2)$ and $mgu\,(q_1, q_2) \cdot mgu\,(p_1, p_2)$ are the most general unifier for pairs of terms (p_1, p_2) and (q_1, q_2). If they are different, then the terms $C\,(p_1, q_1)$ and $C\,(p_2, q_2)$, where C is some binary constructor, will have *different* most general unifiers, which is impossible[1].

With this in mind, we come to the following property. Let $\theta \xrightarrow{u_1;\, u_2;\ldots u_k} \theta'$ is a sequence of unifications of $u_1, \ldots,', u_k$ that transform substitution θ into substitution θ'. Then for any permutation π it is true that

$$\theta \xrightarrow{u_{\pi(1)};\, u_{\pi(2)};\ldots u_{\pi(k)}} \theta'.$$

Another nuance concerns the order of introduction of fresh semantic variables. Say we have a leaf state $\langle \theta, i, ab \rangle$, where θ is a substitution, a and b are applications of relations, and i is the next free semantic variable. In angelic semantics, we can unfold a and b in any order. Since the relations used in the a and b applications may contain `fresh` constructs, these constructs will be evaluated in different order, providing different specific semantic variables and different substitutions. However, these substitutions will be α-equivalent. This observation reflects the following intuitively trivial fact: the order of introduction of semantic variables is not important, as long as no variable is allocated more than once during evaluation in the same branch. Thus, we can ignore `fresh` and consider all substitutions up to α-equivalence. In further reasoning, this will allow us to exclude all the operators of introducing a fresh variable from states.

Lemma 13 (replenishment of substitution during unfolding)**.** Let r be an application of relation and θ be a substitution. Unfolding transforms the state $\langle \theta, r \rangle$ into a set of states $\{\langle \theta_i, \rho_i \rangle\}$, where ρ_i is a conjunction of applications, θ_i is some

[1]Strictly speaking, the validity of these statements depends on the representation of the substitution. For example, in a particular representation, the substitutions $[x \mapsto y]$ and $[y \mapsto x]$ may not be equal, but equivalent.

substitution. Then for each i there is a sequence of unifications $u_1^i \dots u_{k_i}^i$ such that the following is true:

$$\theta \xrightarrow{u_1^i \dots u_{k_i}^i} \theta_i.$$

This lemma can be easily proven by induction in the height of unfolding semantics.

Thus, we can introduce the $[r \to \rho_i]$ notation to perform a sequence of unifications during the unfolding of application r, which leads to the leaf state of ρ_i.

Lemma 14 (weakening in angelic semantics)**.** Let θ be a substitution, ω be a set of applications of relations, and let $\langle\theta, \omega\rangle$ converge to $\{\}$ (empty state) in angelic semantics. Then for an arbitrary substitution σ and an arbitrary set of applications of relations ψ, it is true that $\langle\theta \cdot \sigma, \omega\psi\rangle$ also converges to $\{\}$.

This lemma can be easily proved by induction on the derivation $\langle\theta, \omega\rangle \to^* \{\}$.

This lemma proves the fact that imposing additional constraints (a more specific substitution and introducing additional conjuncts) cannot make a convergent program divergent. This fact will be important in proving the following lemma.

Lemma 15 (preserving the convergence of angelic semantics)**.** Let $\varphi a \varphi'$ be a set of applications of relations with a selected element "a" and θ be a substitution. Let $\{\langle\theta \cdot [a \to \alpha_i], \alpha_i\rangle\}$ be the result of the state unfolding of $\langle\theta, a\rangle$. Then $\langle\theta, \varphi a \varphi'\rangle$ converges if and only if each state $\{\langle\theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi'\rangle\}$ converges.

Proof.

- $\Leftarrow$ Let all states from the set $\{\langle\theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi'\rangle\}$ converge. Then the unfolding of the application of a in the state $\langle\theta, \varphi a \varphi'\rangle$ leads us to a set of conditionally convergent states $\{\langle\theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi'\rangle\}$.
- $\Rightarrow$ Let $\langle\theta, \varphi a \varphi'\rangle$ converge. Then, two cases are possible.
  Case 1. During convergent inference application a was not unfolded. Then $\langle\theta, \varphi a \varphi'\rangle$ converges to $\{\}$ (since each intermediate state contains an application of a). Moreover, the state $\langle\theta, \varphi \varphi'\rangle$ also converges to $\{\}$ (we can repeat the convergent inference $\langle\theta, \varphi a \varphi'\rangle$ for the state $\langle\theta, \varphi \varphi'\rangle$). By Lemma 14, it follows from the convergence of inference for the state $\langle\theta, \varphi \varphi'\rangle$ that all states $\langle\theta \cdot [a \to \alpha_i], \varphi \alpha_i \varphi'\rangle$ converge to $\{\}$.

Case 2. During convergent inference, application a was unfolded at some step. Then for the state $\langle\theta, \varphi a \varphi'\rangle$ there is an inference

$$\langle\theta, \varphi a \varphi'\rangle \to^* \langle\theta \cdot \sigma, \pi a \rho\rangle\,, \qquad (*)$$

where σ is some set of unifications and the next step of this sequence will be the unfolding of a. Then the result of unfolding a will be a set of states $\{\langle\theta \cdot \sigma \cdot [a \to \alpha_i], \pi\alpha_i\rho\rangle\}$. Repeating unfoldings from inference $(*)$ for states $\langle\theta \cdot [a \to \alpha_i], \varphi\alpha_i\varphi'\rangle$, we get the state $\langle\theta \cdot [a \to \alpha_i] \cdot \sigma, \pi\alpha_i\rho\rangle$. This state converges because $\theta \cdot [a \to \alpha_i] \cdot \sigma = \theta \cdot \sigma \cdot [a \to \alpha_i]$.

□

Corollary 1. Let $\langle\theta, \omega\rangle$ be a certain state, and $\{\langle\theta', \omega'\rangle\}$ be a set of states reachable from the state $\langle\theta, \omega\rangle$ for a finite number of unfoldings. Then the state $\langle\theta, \omega\rangle$ converges if and only if all states from the set $\{\langle\theta', \omega'\rangle\}$ converge.

5.3 Conjunction Fairness in Semantics with a Well Quasi-Ordering Choice Predicate

Now we have everything we need to prove the validity of generalized semantics equipped with a well quasi-ordering predicate of choice.

Theorem 3. Let $\mathcal{P}_{\trianglelefteq}$ be a well quasi-ordering predicate. Then "$\to_{\mathcal{P}_{\trianglelefteq}}$" is a fair semantics.

Proof. Let state g converge in angelic semantics. Consider application of relation r, which will be unfolded first in the convergent inference of generalized semantics equipped with a well quasi-ordering choice predicate. There are two possible cases.

Case 1. The semantics "$\to_{\mathcal{P}_{\trianglelefteq}}$" will perform exactly the same unfolding of application of relation r. In this case, both semantics performed an identical step.

Case 2. The "$\to_{\mathcal{P}_{\trianglelefteq}}$" semantics chose some other relation application. By the property of well quasi-ordering, a finite number of unfoldings will be performed before the application of r is unfolded. Let us repeat this finite number of unfoldings in fair semantics. By Corollary 1, the result of the unfolding will belong to the set of converging states.

Thus, the "$\rightarrow_{\mathcal{P}_{\trianglelefteq}}$" semantics will perform all the steps of a converging inference in angelic semantics and, consequently, the inference in "$\rightarrow_{\mathcal{P}_{\trianglelefteq}}$" will also converge. □

This theorem confirms that using a well quasi-ordering predicate as a parameter of the generalized semantics of miniKanren, we obtain a fair semantics.

Corollary 2. The "$\rightarrow_{\mathcal{P}_{\leqslant_{sr}}}$" semantics is a fair semantics.

Proof. Predicate $\leqslant_{sr}$ is a well quasi-ordering predicate by Lemma 11. Therefore, Theorem 3 holds for the "$\rightarrow_{\mathcal{P}_{\leqslant_{sr}}}$" semantics. □

To sum up: semantics equipped with a well quasi-ordering predicate of height of structurally recursive arguments is fair and, as we will see in the next chapter, turns out to be effective in practice.

Chapter 6. Implementation and Experiments

This chapter is dedicated to the software implementation of the proposed relational conversion and fair semantics of miniKanren, as well as the results of experiments.

6.1 Implementation

Relational conversion is implemented as a translator of functional programs into relational ones. The implementation of the operational semantics of miniKanren, including the procedure for dynamic control of the order of conjuncts, is a relational extension for OCaml. OCaml version 4.13 [98] was chosen as the implementation language.

6.1.1 Functional Program Translator

A subset of the OCaml language was chosen as the source language for translated programs, which contains all the components of the functional language described in chapter 2: λ-calculus, recursive and non-recursive `let`-bindings, constructors of expressions of algebraic data types, the pattern matching operator, predefined logical constants `true` and `false` and polymorphic comparison of first-order expressions. OCanren [82] is selected as the target language, which includes all the relational programming components necessary for translation.

The choice of OCaml as the implementation language and the source language of the translator and OCanren (a relational extension of the OCaml language) as the target language is due to the possibility of using the infrastructure components of the standard OCaml compiler [99; 100] in the development of the translator. First of all, we used intermediate representations of programs for the OCaml compiler — *abstract syntax tree* (AST) and *typed abstract syntax tree* (TAST). Both representations are described as a set of OCaml algebraic data types and are a classic way of describing

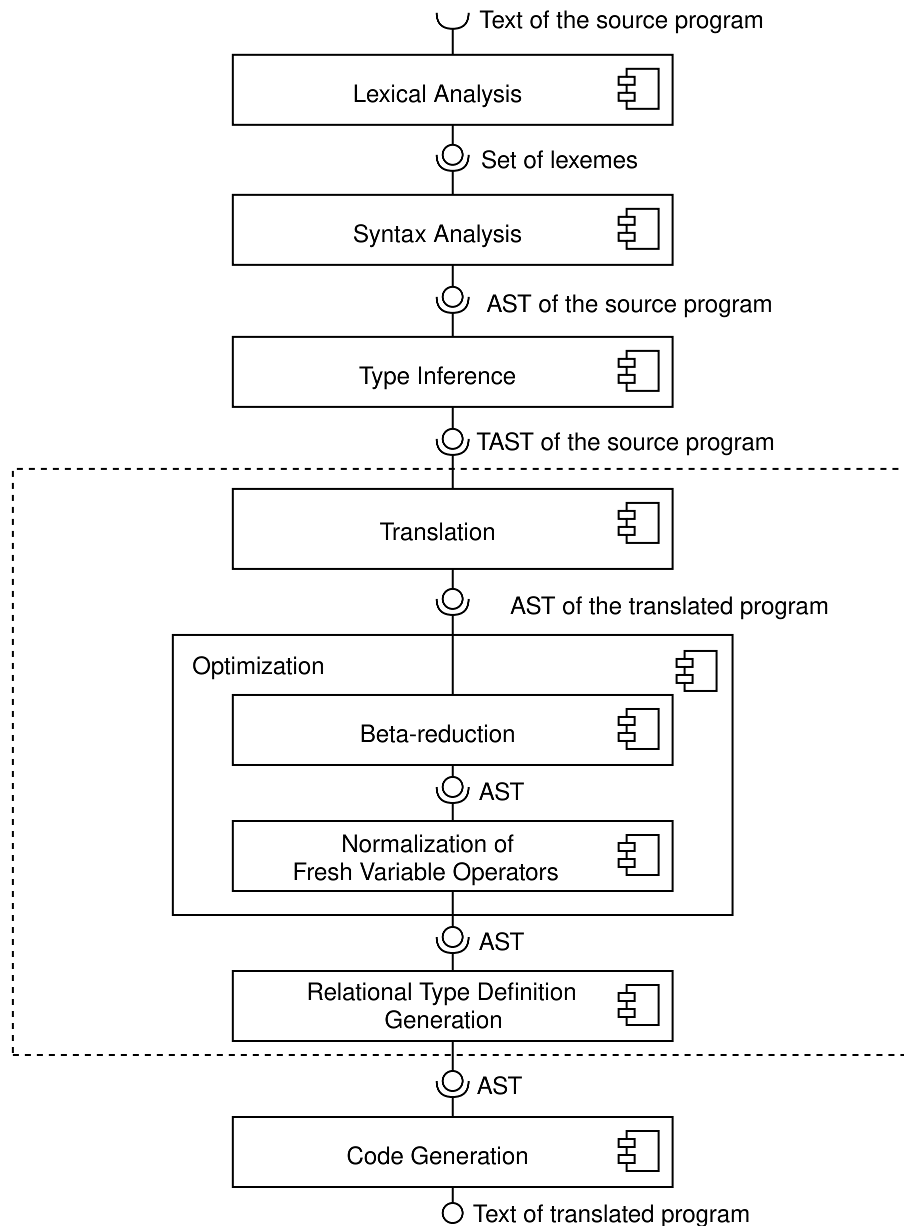


Figure 6.1 — UML diagram of translator components

an intermediate representation of a program. In the development of the translator, several ready-made components of the OCaml compiler were used: lexical analysis, syntactic analysis, type inference and code generation.

The architecture of the translator is shown in Fig. 6.1 as an UML component diagram, in which the components developed by the author of this work are highlighted with a dotted line. Note that the architecture of the translator is a classic optimizing translator [101] and contains traditional components of lexical and syntax analysis, inference of source program types, optimization and code generation. *Lexical Analysis* is responsible for highlighting the lexemes of the language in the text of the source program. *Syntax Analysis* generates the AST of the source program

using the set of its lexemes. *Type Inference* evaluates types for all expressions of the program using the Remy algorithm [102; 103]; this component is based on the Hindley-Milner algorithm [80]. The result of its work is a TAST — a representation of the program corresponding to the AST obtained at the previous stage, in which the inferred type is associated with each expression.

Translation is the main component which constructs a relational program based on a functional one. This component implements the relational conversion presented in chapter 2. Due to the fact that the target language is an extension of the source language, there is no need to introduce another kind of intermediate representation of the program, and therefore the translation result can be represented as an AST for the OCaml language. The *Optimization* component performs several equivalent AST transformations to optimize the final program in terms of its size and efficiency. The β*-reduction* component, which is part of *Optimization*, is responsible for simplifying the constructed program by applying all λ-abstractions to arguments. This is necessary due to a set of λ-abstractions that can be applied statically occurring during translation. This component significantly improves the quality of the resulting programs both in terms of brevity and readability, and performance. The *Normalization of Fresh Variable Operators*, which is also part of *Optimization*, is intended for clustering the introduced logical variables in order to reduce the size of the final program and the overhead associated with the introduction of fresh variables. Note that the translator contains an additional optimization component that ranks conjuncts depending on the assessment of the complexity of their execution and the degree of non-determinism that occurs after their execution. For example, it is reasonable to move the unification and disequality constraint operators to the beginning of the set of conjuncts, since they are deterministic and easier to execute in comparison with relation applications. However, this optimization is necessary only if a relational program is executed using a classical left-biased conjunction. In the case of using fair conjunction, there is no need for this optimization component.

The *Relational Type Definition Generation* component generalizes the algebraic data types of the source program for the correct inference of the types of the final relational program.

The last component, *Code Generation*, transforms the resulting AST obtained via the work of the previous components into a relational program that can be

executed using both classical left-biased conjunction and fair conjunction, and the implementation of which will be discussed in the next section.

This translator is implemented in 1800 lines of OCaml code, which is available here [104].

6.1.2 Relational Extension of the OCaml Language with a Well Quasi-Ordering Choice Predicate

We used the OCaml language to implement a relational extension with a choice predicate for the following reasons. Firstly, this language makes it possible to use the representation of relational terms, unification operators and disequality constraints implemented in the existing OCanren relational extension with left-biased conjunction. Secondly, in order to integrate the relational extension and the translator described in the previous section, it is necessary to match the target language of the translator and the extension being implemented. For the same reason, the syntax of a relational extension with a choice predicate coincides with the syntax of the OCanren extension.

The embedding of the relational extension into the OCaml language was conducted as follows. The structure of a relational program state has been described as an algebraic data type. Further, all extension operators were expressed by means of the OCaml source language as higher-order functions. Thus, any relational program is a combination of relational operators and, thus, a function of the OCaml language. Since the result of a relational program is a potentially infinite set of answers, the `run` function was developed to extract a finite part of the answers as a list.

The architecture of the relational extension is shown in Fig. 6.2 in the form of a UML diagram. The components developed by the author of this work are highlighted with a dotted line.

First of all, consider the auxiliary components that are part of the OCanren relational extension. The *Representation of Logical Terms* component is responsible for the construction and correct typing of logical terms containing both constructors and logical variables. The details of this representation are described in the work [67]. This representation is used in all components that process terms, namely in

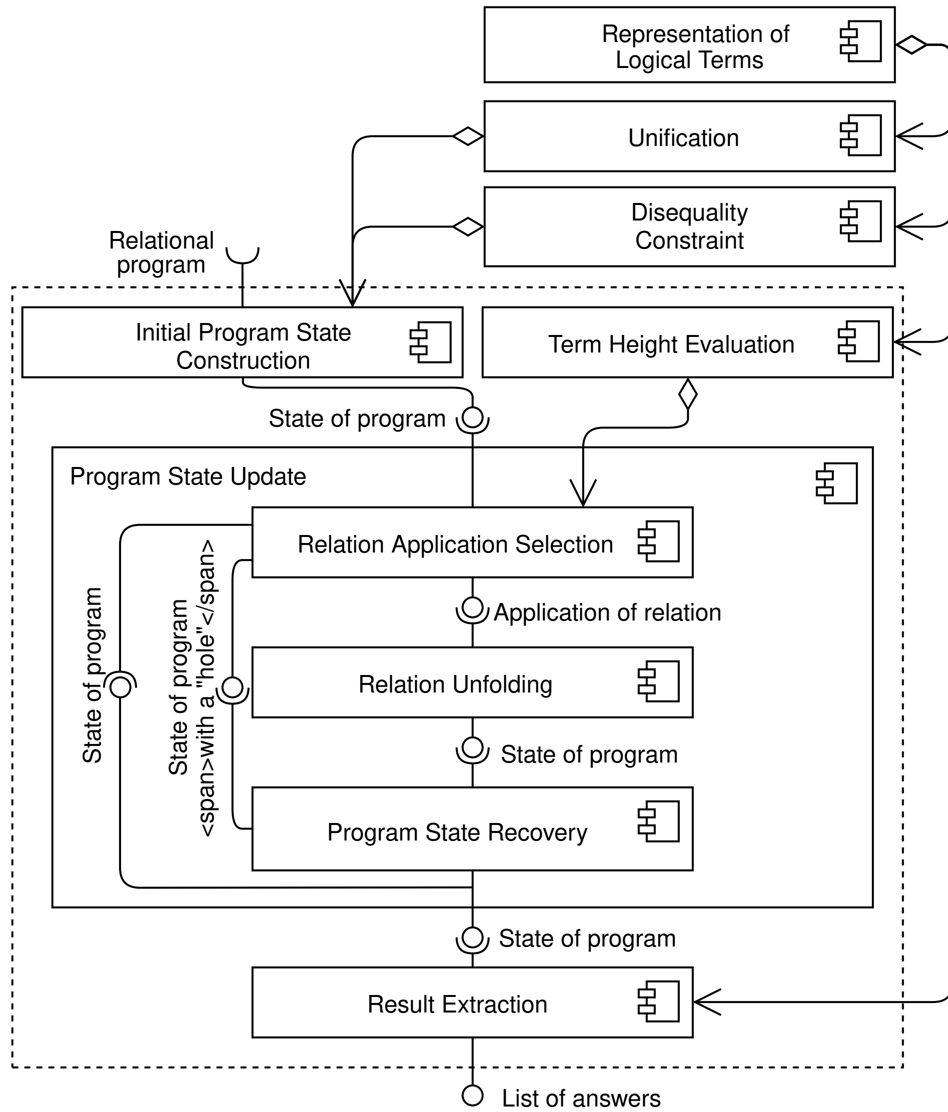


Figure 6.2 — UML diagram of components of relational extension with a well quasi-ordering choice predicate

the unification component, the disequality constraint component, the term height calculation component, and the result extraction component.

Unification and *Disequality Constraint* are responsible for executing relational unification and disequality constraint operators, respectively. The principle of their operation is described in detail in Section 2.2.

The *Initial Program State Construction* component contains implementations of all relational extension operators. Thanks to this component, the stage of interpretation of relational programs is excluded from implementation, since all relational operators are functions of the OCaml language. The initial state is a relational query containing query variables — variables whose final values will be the result of program execution. Also, this component contains an algorithm

for automatic detection of structural-recursive relation arguments. In the absence of such arguments, an argument is found that is structurally decreasing in the maximum number of disjuncts of the relation. Finally, in this component, the developer sets the maximum number of results to be evaluated. If the number is not set, execution will continue until the program state is completely exhausted.

The main component of the conversion is *Program State Update*, which is responsible for step-by-step rebuilding of the state by evaluating its individual parts. It includes the *Relation Application Selection* component, which defines the program state leaf required for the update, selects the application of the relation using the predicate and separates it from the state. The result of this component is a separated application and the state of the program with a “hole” in the place of this application. The selection predicate is based on a comparison of the heights of the arguments, for which the *Term Height Evaluation* component is responsible, which in turn relies on the representation of logical terms. *Relation Unfolding* replaces an application of a relation with the body of the relation and substitutes arguments in the body of the relation. The result of the unfolding is the program state which must be substituted in place of the unfolded relation application. The *Program State Recovery* component is responsible for this. Next, there are two possible cases. If the program state contains the required number of leaves with the resulting substitutions, the update process is completed. Otherwise, the new state of the program has to be updated again.

The last component, *Result Extraction*, selects from the resulting substitutions the terms corresponding to the variables of the original relational query. To extract the results, the term refinement function `walk`, contained in the logical term representation component, is used.

The relational extension with a choice predicate is implemented in 1200 lines of code, available here [105].

6.2 Experiments

The correctness of the proposed relational conversion and semantics of miniKanren is formally proved in chapters 3 and 5, respectively. However, their performance requires experimental evaluation. In the case of a relational conversion,

it is necessary to evaluate the effectiveness of the converted programs. To do this, we optimize the converted programs manually and compare the performance of the optimized and non-optimized versions. In the case of semantics implemented as an extension of the OCaml language, it is necessary to compare the efficiency of fair conjunction in comparison with the classical conjunction in the existing OCanren implementation.

Thus, the *goal* of this experimental study is to compare the efficiency of fair and classical conjunction in execution of manually optimized and non-optimized converted programs.

To achieve this goal, we need to answer the following *questions*.

[Q1] What are the overhead costs of fair conjunction?

[Q2] What is the performance of fair conjunction compared to the classical one?

[Q3] What is the stability of fair conjunction compared to the classical one?

[Q4] What is the performance of optimized programs compared to the non-optimized ones using a fair conjunction?

To answer these questions, it is necessary to calculate the following two *metrics*:

[M1] run time of a set of manually optimized programs that employ classical and fair conjunction;

[M2] run time of a set of non-optimized programs that employ classical and fair conjunction.

We selected two simple programs for experiments, which were executed both in the forward and reverse directions: list reversal relation `revers`^{o} and list sort relation `sort`^{o}. We have selected three other, more complex programs: the first, `hanoi`^{o}, solves the Tower of Hanoi puzzle [1], the second, `bridge`^{o}, solves the bridge and torch puzzle[2], and the third program, `water`^{o}, solves the water pouring puzzle[3]. All of these programs were implemented in a subset of the OCaml functional language for their subsequent relational conversion. After that, relational images of these programs were obtained and, then, manually optimized by reordering the conjuncts.

Both the optimized and non-optimized programs were compiled using both the OCanren extension and the developed relational extension. To reduce the

[1] `https://en.wikipedia.org/wiki/Tower_of_Hanoi`

[2] `https://en.wikipedia.org/wiki/Bridge_and_torch_problem`

[3] `https://en.wikipedia.org/wiki/Water_pouring_puzzle`

Program	Input size	Classical conjunction	Fair conjunction
revers^o in forward direction	30	0.002	0.002
	60	0.012	0.012
	90	0.044	0.045
revers^o in reverse direction	30	0.002	0.002
	60	0.014	0.014
	90	0.054	0.055
sort^o in forward direction	30	0.031	0.032
	60	0.357	0.362
	30	1.324	1.388
sort^o in reverse direction	3	0.001	0.001
	4	0.001	0.001
	5	0.012	0.012
	6	0.150	0.158
hanoi^o	-	0.597	0.620
bridge^o	-	0.074	0.075
water^o	-	1.247	1.317

Table 1 — Run time (in seconds) of an array of manually optimized programs using classical and fair conjunction

measurement error, each program was run 20 times in order to calculate its average run time.

Table 1 in accordance with **[M1]** shows the results of the manually optimized programs using classical and fair conjunction. As one can see, the run time of programs using classical conjunction is slightly smaller compared to the run time of programs using fair conjunction. Since these programs are optimized for a specific query by selecting the optimal order of conjuncts, fair conjunction does not lead to an increase in performance, but its use adds minor overhead, which is less than 6% of the total run time. Thus, the answer to **[Q1]** is as follows: the overhead is insignificant and does not exceed 6% of the total execution time.

Table 2, in accordance with **[M2]**, shows the run time results of non-optimized programs that use classical and fair conjunction. In the case of using a classical conjunction with a non-optimal order of conjuncts, a program has exponential asymptotic time complexity. The run time of the program was limited to 300 seconds,

Program	Input size	Classical conjunction	Fair conjunction
revers^o	30	0.002	0.002
in forward	60	0.012	0.012
direction	90	0.044	0.046
revers^o	30	0.016	0.003
in reverse	60	0.176	0.015
direction	90	0.860	0.060
sort^o	30	0.034	0.035
in forward	60	0.363	0.377
direction	90	1.350	1.417
sort^o in reverse direction	3	0.001	0.001
	4	6.785	0.001
	5	>300	0.013
	6	>300	0.161
hanoi^o	-	>300	0.644
bridge^o	-	34.165	0.075
water^o	-	>300	1.326

Table 2 — Run time (in seconds) of a set of non-optimized programs using classical and fair conjunction

so the exact value is unknown. In the table, “>300” denotes this case. These programs are obtained automatically and are not optimized for a specific query, so the run time of programs using classical conjunction is significantly larger than the run time of programs using fair conjunction. In the best case, for non-optimized programs, the performance of fair conjunction exceeds the classical one by more than 450 times. Thus, the answer to the question **[Q2]** is as follows: in the case of non-optimized relational programs, the performance of the fair conjunction is higher compared to the classical one, and in the peak case it exceeds by more than 450 times.

To get an answer to the questions **[Q3]** and **[Q4]**, let us compare fair conjunction run times for manually optimized and non-optimized programs. As can be seen from Tables 1 and 2, the run time of programs using fair conjunction has no significant differences and answers the question **[Q3]** as follows: fair conjunction demonstrates high stability and does not depend on the order of conjuncts in the program. At the same time, we can answer the question **[Q4]** as follows: when using

fair conjunction, the performance of non-optimized programs corresponds to the performance of optimized programs.

Summing up, we draw the following conclusion: fair conjunction based on structural recursion is comparable in efficiency with the classical directed conjunction in the case of optimized programs, and also demonstrates higher performance in the case of non-optimized programs. At the same time, when using fair conjunction, the performance of non-optimized programs is comparable to the performance of optimized programs.

Conclusion

As a result, we formulate the main results achieved in this work.

1. New approach to relational conversion of general-form functional programs was developed and its static and dynamic correctness was proven.
2. Formal angelic semantics of the miniKanren was impleneted and their equivalence to declarative miniKanren semantics was proven. The notion of conjunction fairness as property of angelic semantics was defined.
3. Formal semantics of relational miniKanren with dynamic control of the conjunct calculation order was implemented and conjunction fairness was proven.
4. Relational conversion for a subset of OCaml was implemented on OCaml, and an experimental study was conducted that shows high effectiveness of automatically obtained relational programs.
5. An embedding of miniKanren with dynamic control of conjunct calculation order in functional language OCaml was implemented on OCaml, and experimental study was conducted that shows high efficiency in comparison to the classic implementation of miniKanren with a non-optimal order of conjuncts and the insignificance of overhead with an optimal order of conjuncts.

As **recommendations for the use of the obtained results** in industry and scientific research, we indicate the following. At the moment, the development of relational programs is a complex task that requires developers to have deep knowledge in this area. As a result, relational programming is used mainly in scientific research by individual experts to solve a narrow class of problems of finding solutions, validating and synthesizing programs [38; 46; 60]. However, thanks to the methods proposed in this work, relational programming becomes more accessible and, as a result, more attractive to a wider range of users. In addition, automated conversion of relational programs according to functional ones will allow developing complex target solutions that combine functional and relational behaviors within the infrastructure of one language.

The following can be noted as **prospects for further development of the topic**:

- further researches of the features of the fair execution of a relational conjunction, including the search for effective choice predicates for relational programs of a special type, the research of choice predicates in addition to the completely quasi-ordering predicates considered in this dissertation research;
- a generalization of the fairness property for relational disjunction;
- extension of the source language for relational conversion, in particular, the addition of a pattern matching operator with non-disjunct patterns.

References

1. *Prawitz, D.* An Improved Proof Procedure [Text] / D. Prawitz // Theoria. — 1960. — Vol. 26, no. 2. — P. 102—139.
2. *Gilmore, P. C.* A Proof Method for Quantification Theory: Its Justification and Realization [Text] / P. C. Gilmore // IBM Journal of Research and Development. — 1960. — Vol. 4, no. 1. — P. 28—35.
3. *Davis, M.* A Computing Procedure for Quantification Theory [Text] / M. Davis, H. Putnam // J. ACM. — New York, NY, USA, 1960. — Vol. 7, no. 3. — P. 201—215.
4. *Hewitt, C.* PLANNER: A Language for Proving Theorems in Robots [Text] / C. Hewitt // Proceedings of the 1st International Joint Conference on Artificial Intelligence. — Washington, DC : Morgan Kaufmann Publishers Inc., 1969. — P. 295—301.
5. *Kowalski, R.* Predicate Logic as a Programming Language [Text] / R. Kowalski // Information Processing. — Stockholm, North Holland, 1974. — P. 569—574.
6. Un système de communication homme-machine en Français [Text] / A. Colmerauer [et al.] // Groupe de recherche en Intelligence Artificielle. — Marseille, France, 1973.
7. Information Technology — Programming Languages — Prolog — Part 1: General Core [Text] : Standard / International Organization for Standardization. — 1995.
8. Information Technology — Programming Languages — Prolog — Part 2: Modules [Text] : Standard / International Organization for Standardization. — Geneva, Switzerland, 2000.
9. *Colmerauer, A.* The Birth of Prolog [Text] / A. Colmerauer, P. Roussel. — 1996.
10. *Kowalski, R.* Algorithm = Logic + Control [Text] / R. Kowalski // Commun. ACM. — New York, NY, USA, 1979. — Vol. 22, no. 7. — P. 424—436.

11. *Merritt, D.* Building Expert Systems in Prolog [Text] / D. Merritt. — Berlin, Heidelberg : Springer-Verlag, 1989.
12. *Beckert, B.* leanTAP: Lean Tableau-based Deduction [Text] / B. Beckert, J. Posegga // Journal of Automated Reasoning. — 1995. — Vol. 15. — P. 339—358.
13. *Clocksin, W. F.* Clause and Effect: Prolog Programming for the Working Programmer [Text] / W. F. Clocksin. — Secaucus, New Jersey, USA : Springer, 1997.
14. *Weijland, W.* Semantics for Logic Programs without Occur Check [Text] / W. Weijland // Theoretical Computer Science. — 1990. — Vol. 71, no. 1. — P. 155—174.
15. *Knuth, D. E.* The Art of Computer Programming: Combinatorial Algorithms, Part 1 [Text] / D. E. Knuth. — 3rd. — Addison-Wesley Professional, 1997.
16. *Arbab, B.* Operational and Denotational Semantics of Prolog [Text] / B. Arbab, D. M. Berry // The Journal of Logic Programming. — 1987. — Vol. 4, no. 4. — P. 309—329.
17. *Stroetmann, K.* A Declarative Semantics for the Prolog Cut Operator [Text] / K. Stroetmann, T. Glaß // Proceedings of the 5th International Workshop on Extensions of Logic Programming. — Berlin, Heidelberg : Springer-Verlag, 1996. — P. 255—271.
18. *Ceri, S.* What You Always Wanted to Know About Datalog (And Never Dared to Ask) [Text] / S. Ceri, G. Gottlob, L. Tanca // IEEE Transactions on Knowledge and Data Engineering. — 1989. — Vol. 1, no. 1. — P. 146—166.
19. *Cheney, J.* Nominal Logic Programming [Text] : PhD thesis / Cheney J. — Valencia, Spain : Cornell University, 2004.
20. *Cheney, J.* αProlog: A Logic Programming Language with Names, Binding, and α-Equivalence [Text] / J. Cheney, C. Urban // Proceedings of the 20th Intl. Conf. on Logic Programming. — Saint-Malo, France, 2004.
21. *Nadathur, G.* An Overview of λprolog [Text] / G. Nadathur, D. Miller // Logic Programming, Proceedings of the Fifth International Conference and Symposium / ed. by R. Kowalski, K. Bowen. — Seattle, Washington, USA : MIT Press, 1988. — P. 810—827.

22. *Nadathur, G.* The Metalanguage λprolog and Its Implementation [Text] / G. Nadathur // Proceedings of the 5th International Symposium on Functional and Logic Programming. — Berlin, Heidelberg : Springer-Verlag, 2001. — P. 1—20.

23. *Hanus, M.* Curry: A Truly Functional Logic Language [Text] / M. Hanus, H. Kuchen, J. J. Moreno-Navarro //. — 1995.

24. *Hanus, M.* Search Strategies for Functional Logic Programming [Text] / M. Hanus, B. Peemöller, F. Reck // Software Engineering 2012. Workshopband / ed. by S. Jähnichen, B. Rumpe, H. Schlingloff. — Bonn : Gesellschaft für Informatik e.V., 2012. — P. 61—74.

25. *Hanus, M.* A Generic Analysis Environment for Declarative Programs [Text] / M. Hanus // Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming. — Tallinn, Estonia : Association for Computing Machinery, 2005. — P. 43—48.

26. *Hanus, M.* A Typeful Integration of SQL into Curry [Text] / M. Hanus, J. Krone // Electronic Proceedings in Theoretical Computer Science. — 2016. — Vol. 234. — P. 104—119.

27. *Hanus, M.* Declarative Programming of User Interfaces [Text] / M. Hanus, C. Kluß //. — 2009. — P. 16—30.

28. *Somogyi, Z.* The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language [Text] / Z. Somogyi, F. Henderson, T. Conway // The Journal of Logic Programming. — 1996. — Vol. 29, no. 1. — P. 17—64. — High-Performance Implementations of Logic Programming Systems.

29. *Overton, D.* Precise and Expressive Mode Systems for Typed Logic Programming Languages [Text] : PhD thesis / Overton David. — Department of Computer Science, Software Engineering, The University of Melbourne, 2004.

30. *Hanus, M.* Ontology Driven Software Engineering for Real Life Applications [Text] / M. Hanus, J. Krone //. — Innsbruck, Austria, 2007.

31. *Hill, P.* The Gödel Programming Language [Text] / P. Hill, J. W. Lloyd. — The MIT Press, 1994.

32. *Pfenning, F.* System Description: Twelf — A Meta-Logical Framework for Deductive Systems [Text] / F. Pfenning, C. Schürmann // Proceedings of 16th International Conference on Automated Deduction. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1999. — P. 202—206.

33. *Crary, K.* Higher-Order Representation of Substructural Logics [Text] / K. Crary // Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming. — Baltimore, Maryland, USA : Association for Computing Machinery, 2010. — P. 131—142.

34. *Lee, D. K.* Towards a Mechanized Metatheory of Standard ML [Text] / D. K. Lee, K. Crary, R. Harper // Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — Nice, France : Association for Computing Machinery, 2007. — P. 173—184.

35. The Reasoned Schemer [Text] / D. P. Friedman [et al.]. — 2nd. — The MIT Press, 2005.

36. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl) [Text] / O. Kiselyov [et al.] // SIGPLAN Not. — New York, NY, USA, 2005. — Vol. 40, no. 9. — P. 192—203.

37. *Rozplokhas, D.* Scheduling Complexity of Interleaving Search [Text] / D. Rozplokhas, D. Boulytchev // Proceedings of the 16th International Symposium on Functional and Logic Programming. — 2022.

38. *Byrd, W. E.* MiniKanren, Live and Untagged: Quine Generation via Relational Interpreters (Programming Pearl) [Text] / W. E. Byrd, E. Holk, D. P. Friedman // Proceedings of the Annual Workshop on Scheme and Functional Programming. — Copenhagen, Denmark : Association for Computing Machinery, 2012. — P. 8—29.

39. *Near, J. P.* αleanTAP: A Declarative Theorem Prover for First-Order Classical Logic [Text] / J. P. Near, W. E. Byrd, D. P. Friedman // Logic Programming / ed. by M. Garcia de la Banda, E. Pontelli. — Berlin, Heidelberg, 2008. — P. 238—252.

40. *Byrd, W. E.* Relational Programming in miniKanren: Techniques, Applications, and Implementations [Text] : PhD thesis / Byrd William E. — Indiana University, 2009.

41. *Hemann, J.* μKanren: A Minimal Functional Core for Relational Programming [Text] / J. Hemann, D. P. Friedman // Proceedings of the 2013 Workshop on Scheme and Functional Programming. — Alexandria, VA, 2013.

42. *Swords, C.* rKanren: Guided Search in miniKanren [Text] / C. Swords, D. P. Friedman // In Proceedings of the 2013 Workshop on Scheme and Functional Programming. — Alexandria, VA, USA, 2013.

43. *Hemann, J.* A Framework for Extending microKanren with Constraints [Text] / J. Hemann, D. P. Friedman // In Proceedings of the 2015 Workshop on Scheme and Functional Programming. — 2015.

44. *Byrd, W. E.* αKanren A Fresh Name in Nominal Logic Programming [Text] / W. E. Byrd, D. P. Friedman // In Proceedings of the 2007 Workshop on Scheme and Functional Programming. — 2007. — P. 79—90.

45. *Moiseenko, E.* Constructive Negation for miniKanren [Text] / E. Moiseenko // Proceedings of the 2019 miniKanren and Relational Programming Workshop. — 2019. — P. 58—78.

46. A Unified Approach to Solving Seven Programming Problems (Functional Pearl) [Text] / W. E. Byrd [et al.] // Proceedings of ACM Program. Lang. — New York, NY, USA, 2017. — 8:1—8:26.

47. Guarded Fresh Goals: Dependency-Directed Introduction of Fresh Logic Variables [Text] / C. E. Alvis [et al.] // Proceedings of the 2021 miniKanren and Relational Programming Workshop. — 2021.

48. *Lu, K.-C.* Towards a miniKanren with Fair Search Strategies [Text] / K.-C. Lu, W. Ma, D. P. Friedman // Proceedings of the 2019 miniKanren and Relational Programming Workshop. — 2019. — P. 1—15.

49. *Rozplokhas, D.* Improving Refutational Completeness of Relational Search via Divergence Test [Text] / D. Rozplokhas, D. Boulytchev // Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming. — New York, NY, USA : Association for Computing Machinery, 2018. — 18:1—18:13.

50. *Schrijvers, T.* Tor: Extensible Search with Hookable Disjunction [Text] / T. Schrijvers, M. Triska, B. Demoen // Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. — Leuven, Belgium : Association for Computing Machinery, 2012. — P. 103—114.

51. *S.M.A.*, *P.* Tabling in contextual abduction with answer subsumption [Text] / P. S.M.A., S. A., P. L.M. // 2017 International Conference on Advanced Computer Science and Information Systems, ICACSIS 2017. — 2018. — P. 459—464.

52. *J.*, *A.* Evaluation of the Implementation of an Abstract Interpretation Algorithm using Tabled CLP [Text] / A. J., C. M. // Theory and Practice of Logic Programming. — 2019. — Vol. 19. — P. 1107—1123.

53. *Tamaki*, *H.* OLD Resolution with Tabulation [Text] / H. Tamaki, T. Sato // Proceedings of the Third International Conference on Logic Programming. — 1986. — P. 84—98.

54. *Naish*, *L.* Automating Control for Logic Programs [Text] / L. Naish // J. Log. Program. — 1985. — Vol. 2. — P. 167—183.

55. *Lloyd*, *J. W.* Foundations of Logic Programming [Text] / J. W. Lloyd. — Berlin, Heidelberg : Springer-Verlag, 1984.

56. Parallel Execution of Prolog Programs: A Survey [Text] / G. Gupta [et al.] // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 2001. — Vol. 23, no. 4. — P. 472—602.

57. *Lozov*, *P.* Typed relational conversion [Text] / P. Lozov, A. Vyatkin, D. Boulytchev // Trends in Functional Programming. — Springer International Publishing, 2018. — P. 39—58.

58. *Lozov*, *P.* Conversion Typed Functions into Relational Form [Text] / P. Lozov, D. Boulytchev // Proceedings of the Institute for System Programming of the RAS. — 2018. — Vol. 30, no. 2. — 45—64 (In Russ.)

59. *Minsky*, *Y.* Real World OCaml: Functional Programming for the Masses [Text] / Y. Minsky, A. Madhavapeddy, J. Hickey. — O'Reilly Media, 2013.

60. *Lozov*, *P.* Relational Interpreters for Search Problems [Text] / P. Lozov, E. Verbitskaia, D. Boulytchev // Proceedings of the 2019 miniKanren and Relational Programming Workshop. — 2019. — P. 43—57.

61. *Lozov*, *P.* On Fair Relational Conjunction [Text] / P. Lozov, D. Boulytchev // Proceedings of the 2020 miniKanren and Relational Programming Workshop. — 2020. — P. 1—12.

62. *Lozov, P.* Efficient fair conjunction for structurally-recursive relations [Text] / P. Lozov, D. Boulytchev // Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. — Association for Computing Machinery, 2021. — P. 58—73.

63. *Kosarev, D.* Relational Synthesis for Pattern Matching [Text] / D. Kosarev, P. Lozov, D. Boulytchev // Programming Languages and Systems / ed. by B. C. d. S. Oliveira. — Springer International Publishing, 2020. — P. 293—310.

64. *Wright, A.* A Syntactic Approach to Type Soundness [Text] / A. Wright, M. Felleisen // Inf. Comput. — Duluth, MN, USA, 1994. — Vol. 115, no. 1. — P. 38—94.

65. *Barendregt, H. P.* Handbook of Logic in Computer Science (Vol. 2) [Text] / H. P. Barendregt ; ed. by S. Abramsky, D. M. Gabbay, S. E. Maibaum. — Oxford University Press, Inc., 1992. — Chap. Lambda Calculi with Types. P. 117—309.

66. *Urzyczyn, P.* Inhabitation in Typed Lambda-Calculi (A Syntactic Approach) [Text] / P. Urzyczyn // Proceedings of the Third International Conference on Typed Lambda Calculi and Applications. — Berlin, Heidelberg : Springer-Verlag, 1997. — P. 373—389.

67. *Kosarev, D.* Typed Embedding of a Relational Language in OCaml [Text] / D. Kosarev, D. Boulytchev // Proceedings Electronic Proceedings in Theoretical Computer Science. Vol. 285 / ed. by K. Asai, M. R. Shinwell. — Nara, Japan, 2016. — P. 1—22.

68. A Small Embedding of Logic Programming with a Simple Complete Search [Text] / J. Hemann [et al.] // SIGPLAN Not. — New York, NY, USA, 2016. — Vol. 52, no. 2. — P. 96—107.

69. cKanren: miniKanren with Constraints [Text] / C. E. Alvis [et al.] // Proceedings of the 2011 Annual Workshop on Scheme and Functional Programming. — 2011.

70. *Baader, F.* Handbook of Automated Reasoning [Text] / F. Baader, W. Snyder ; ed. by A. Robinson, A. Voronkov. — Amsterdam, The Netherlands, The Netherlands : Elsevier Science Publishers B. V., 2001. — Chap. Unification Theory.

71. *Cardelli*, *L.* On Understanding Types, Data Abstraction, and Polymorphism [Text] / L. Cardelli, P. Wegner // ACM Comput. Surv. — New York, NY, USA, 1985. — Vol. 17, no. 4. — P. 471—523.

72. *Schmidt*, *D. A.* Denotational Semantics: A Methodology for Language Development [Text] / D. A. Schmidt. — USA : William C. Brown Publishers, 1986.

73. *Winskel*, *G.* The Formal Semantics of Programming Languages: An Introduction [Text] / G. Winskel. — MIT Press, 1993.

74. *Fernández*, *M.* Programming Languages and Operational Semantics: An Introduction [Text] / M. Fernández. — King's College Publications, 2004.

75. A Simple Applicative Language: Mini-ML [Text] / D. Clément [et al.] // Proceedings of the 1986 ACM Conference on LISP and Functional Programming. — New York, NY, USA : Association for Computing Machinery, 1986. — P. 13—27.

76. *Plotkin*, *G.* Call-by-name, call-by-value and the λ-calculus [Text] / G. Plotkin // Theoretical Computer Science. — 1975. — Vol. 1, no. 2. — P. 125—159.

77. *Plotkin*, *G.* A Structural Approach to Operational Semantics [Text] : tech. rep. / G. Plotkin. — University of Aarhus, 1981. — DAIMI FN—19.

78. *Felleisen*, *M.* The Calculi of lambda-nu-cs Conversion: a Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages [Text] : PhD thesis / Felleisen Matthias. — Indiana University, 1987.

79. Oficial miniKanren web-page [Electronic Resource]. — URL: http://minikanren.org (visited on 06/01/2022).

80. *Hindley*, *R.* The Principal Type-Scheme of an Object in Combinatory Logic [Text] / R. Hindley // Transactions of the American Mathematical Society. — 1969. — Vol. 146. — P. 29—60.

81. *Milner*, *R.* A Theory of Type Polymorphism in Programming [Text] / R. Milner // Journal of Computer and System Sciences. — 1978. — Vol. 17, no. 3. — P. 348—375.

82. OCanren web-page [Electronic Resource]. — URL: https://github.com/JetBrains-Research/OCanren (visited on 06/01/2022).

83. *Lassez, J.-L.* Foundations of Deductive Databases and Logic Programming [Text] / J.-L. Lassez, M. J. Maher, K. Marriott ; ed. by J. Minker. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1988. — Chap. Unification Revisited. P. 587—625.

84. *Pierce, B.* Types and Programming Languages [Text] / B. Pierce. — MIT Press, 2002. — Chap. 8.3 Safety = Progress + Preservation.

85. *Hodges, W.* Formal Features of Compositionality [Text] / W. Hodges // Journal of Logic, Language, and Information. — 2001. — Vol. 10, no. 1. — P. 7—28.

86. *Church, A.* Some Properties of Conversion [Text] / A. Church, J. B. Rosser // Transactions of the American Mathematical Society. — 1936. — Vol. 39, no. 3. — P. 472—482.

87. *N. A. Lynch, F. W. V.* Forward and Backward Simulations: I. Untimed Systems [Text] / F. W. V. N. A. Lynch // Information and Computation. — 1995. — Vol. 121. — P. 214—233.

88. *N. A. Lynch, F. W. V.* Forward and Backward Simulations, II: Timing-Based Systems [Text] / F. W. V. N. A. Lynch // Information and Computation. — 1996. — Vol. 128. — P. 1—25.

89. *Rozplokhas, D.* Certified Semantics for Relational Programming [Text] / D. Rozplokhas, A. Vyatkin, D. Boulytchev // Programming Languages and Systems / ed. by B. C. d. S. Oliveira. — Springer International Publishing, 2020. — P. 167—185.

90. *Keller, R. M.* Formal Verification of Parallel Programs [Text] / R. M. Keller // Commun. ACM. — 1976. — Vol. 19, no. 7. — P. 371—384.

91. *Bertot, Y.* Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions [Text] / Y. Bertot, P. Castéran. — Springer, 2004. — (Texts in Theoretical Computer Science. An EATCS Series).

92. *Turchin, V. F.* The Concept of a Supercompiler [Text] / V. F. Turchin // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 1986. — Vol. 8, no. 3. — P. 292—325.

93. *Sørensen*, *M. H.* An Algorithm of Generalization in Positive Supercompilation [Text] / M. H. Sørensen, R. Glück // Proceedings of ILPS'95, the International Logic Programming Symposium. — MIT Press, 1995. — P. 465—479.

94. *Kruskal*, *J. B.* Well-Quasi-Ordering, The Tree Theorem, and Vazsonyi's Conjecture [Text] / J. B. Kruskal // Transactions of the American Mathematical Society. — 1960. — Vol. 95, no. 2. — P. 210—225.

95. *Higman*, *G.* Ordering by Divisibility in Abstract Algebras [Text] / G. Higman // Proceedings of the London Mathematical Society. — 1952. — No. 1. — P. 326—336.

96. *Friedman*, *D. P.* Fancy Ferns Require Little Care [Text] / D. P. Friedman, D. S. Wise // Symposium on Functional Languages and Computer Architecture. — 1981. — P. 124—156.

97. *Leuschel*, *M.* On the Power of Homeomorphic Embedding for Online Termination [Text] / M. Leuschel // Static Analysis / ed. by G. Levi. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1998. — P. 230—245.

98. OCaml 4.13.0 Release Notes [Electronic Resource]. — URL: https://ocaml.org/releases/4.13.0 (visited on 06/01/2022).

99. The core OCaml system web-page [Electronic Resource]. — URL: https://github.com/ocaml/ocaml (visited on 06/01/2022).

100. *Rémy*, *D.* Objective ML: A Simple Object-Oriented Extension of ML. [Text] / D. Rémy, J. Vouillon // Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 1997. — P. 40—53.

101. *Aho*, *A. V.* The Theory of Parsing, Translation, and Compiling [Text] / A. V. Aho, J. D. Ullman. — USA : Prentice-Hall, Inc., 1972.

102. *Wright*, *A. K.* A Syntactic Approach to Type Soundness [Text] / A. K. Wright, M. Felleisen // Inf. Comput. — 1994. — Vol. 115. — P. 38—94.

103. *Rémy*, *D.* Using, Understanding, and Unraveling the OCaml Language From Practice to Theory and Vice Versa [Text] / D. Rémy // Applied Semantics / ed. by G. Barthe [et al.]. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. — P. 413—536.

104. noCanren web-page [Electronic Resource]. — URL: https://github.com/Lozov-Petr/noCanren (visited on 06/01/2022).

105. OCanren with fair conjunction web-page [Electronic Resource]. — URL: https://github.com/JetBrains-Research/OCanren/tree/fair (visited on 06/01/2022).

List of Figures

1.1 Example of Unnesting conversion 17
1.2 An invalid case of Unnesting conversion 18
1.3 Big-step semantics for the language of binary arithmetic expressions . . 22
1.4 Small-step semantics for the language of binary arithmetic expressions . 23
1.5 Semantics in Felleisen notation for the language of binary arithmetic expressions 25

2.1 Source language syntax 28
2.2 Typing rules of the source language 29
2.3 Semantics of the source language 31
2.4 Relational extension syntax 32
2.5 Typing rules of the relational extension 33
2.6 Semantics of the relational extension 34

4.1 Small step semantics of local evaluations 61
4.2 miniKanren angelic semantics 62
4.3 Generalized semantics of the miniKanren language, parameterized by choice predicate $\mathcal{P}$ 65

6.1 UML diagram of translator components 76
6.2 UML diagram of components of relational extension with a well quasi-ordering choice predicate 79

List of Tables

1 Run time (in seconds) of an array of manually optimized programs using classical and fair conjunction 82

2 Run time (in seconds) of a set of non-optimized programs using classical and fair conjunction 83