ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Факультет «Школа информатики, физики и технологий»

Трилис Алексей Андреевич

ЭВРИСТИЧЕСКИЕ СТРАТЕГИИ РЕДУКЦИИ ТЕРМОВ В ЗАДАЧАХ ФОРМАЛЬНОЙ ВЕРИФИКАЦИИ

Выпускная квалификационная работа
по направлению подготовки 01.04.02 Прикладная математика и информатика
образовательная программа «Программирование и анализ данных»

Рецензент
А.А. Трунов

Научный руководитель
канд. физ.-мат. наук, доц.
Д.Н. Москвин

Соруководитель
А.М. Ляшин

Санкт-Петербург 2025

Оглавление

Аннотация **3**

Введение **5**

1. Обзор предметной области **8**

1.1. Редукции термов в лямбда-исчислении 8

1.2. Классические порядки редукций 10

1.3. Проблемы производительности в Coq 12

1.4. Система Ursus 14

1.5. Выводы и результаты по главе 17

2. Разработка стратегий **18**

2.1. Анализ входных данных 18

2.2. Базовые стратегии 20

2.3. Графовые оптимизации 22

2.4. Оптимизации, основанные на типах данных 25

2.5. Работа с условными операторами 27

2.6. Выводы и результаты по главе 29

3. Сравнение стратегий **30**

3.1. Подготовка тестовых данных 30

3.2. Эксперименты на линейном и рекурсивном коде 31

3.3. Эксперименты на коде с условными операторами 36

3.4. Выбор внутренней редукции 37

3.5. Эксперименты на реальных данных 38

3.6. Выводы и результаты по главе 40

Заключение **42**

Список литературы **43**

Аннотация

Интерактивное средство доказательств теорем Coq/Rocq используется во многих работах по формальной верификации программ. Однако, средства Coq по вычислениям, необходимым для формальной верификации, зачастую оказываются недостаточно производительными для решения практических задач. В системе Ursus для верификации смарт-контрактов на Coq ключевой задачей является эффективное символьное вычисление результатов функций. Данная работа посвящена оптимизации этого процесса. Разработаны несколько стратегий вычисления, основанные на вызове по значению и вызове по необходимости, применяющие техники декомпозиции и эвристически учитывающие структуру и закономерности во входных данных. В том числе, исследуются эвристики, связанные с типами данных и графовыми характеристиками задачи. Разработанные стратегии сравниваются друг с другом как на синтетических, так и на реальных данных, и в результате для практического использования выбраны несколько стратегий, показавших наилучший результат.

Ключевые слова: формальная верификация, стратегии редукции, оптимизация, Coq, Rocq, Ursus.

The interactive theorem prover Coq/Rocq is widely used in many works on formal software verification. However, Coq's computational mechanisms, which are essential for formal verification, often lack the performance needed to solve practical problems. In the Ursus framework, designed for verifying smart contracts in Coq, a key challenge is the efficient symbolic evaluation of function results. This work is dedicated to optimizing that process. Several evaluation strategies have been developed, based on call-by-value and call-by-need approaches, employing decomposition techniques and heuristically taking into account structural patterns in the input data, including heuristics related to data types and graph-based characteristics of the problem. The developed strategies are compared against one another on both synthetic and real-world data, leading to the selection of several strategies that demonstrated the best results for practical use.

Keywords: formal verification, reduction strategies, optimization, Coq, Rocq, Ursus.

Введение

Формальная верификация является важной и активно развивающейся областью исследований. В программных продуктах нередко встречаются ошибки, которые трудно заметить даже опытному программисту. Традиционные методы нахождения программных ошибок, такие как тестирование, хоть и зачастую являются эффективными, не могут гарантировать полное отсутствие ошибок, так как проверка происходит лишь на ограниченном множестве входных данных. В то же время в некоторых областях, таких как авиация, медицина, финансы, цена ошибки может оказаться крайне высокой, и уровень корректности, обеспечиваемый тестированием, перестаёт являться приемлемым. В таких областях требуется математически доказать корректность системы, то есть её соответствие формальной спецификации.

Интерактивное средство доказательств теорем Coq, также известное как Rocq [1] используется во многих работах по математике и формальной верификации. В частности, знаменитая теорема о четырёх красках была доказана именно в этой системе [2], а среди крупных проектов по формальной верификации на Coq можно выделить доказанно корректный оптимизирующий компилятор для подмножества языка C CompCert [3] и фреймворк для верификации распределённых систем IronFleet [4].

Coq предоставляет мощный и выразительный язык для формальной верификации. Этот язык содержит богатую систему типов, предлагает удобные возможности для расширения, в том числе разработку стратегий доказательства на языке Ltac. Вместе с этим, производительность языка может вызывать практические проблемы. На некоторых практических задачах вычисления могут показывать экспоненциальное время работы [5] [6].

Смарт-контракты, то есть программы, запускаемые в рамках блокчейн-сетей — особенно подходящее применение для формальных методов [7], поскольку эти программы, с одной стороны, являются относительно простыми, а с другой зачастую оперируют большими денежными суммами, то есть цена ошибки может быть высокой. Вдобавок, изменение таких программ с целью исправления ошибок после их запуска в силу особенностей блокчейн-сетей является сложным или даже невозможным, поэтому необходимо найти ошибки до запуска. Язык Ursus [8], улучшению которого посвящена эта работа, использует возможности расширения Coq для создания системы верификации смарт-контрактов.

В рамках проекта Ursus были достигнуты значительные результаты по поддержке языков программирования, используемых в разработке смарт-контрактов, таких как Solidity и Rust, и по автоматизации процесса верификации, от трансляции исходного кода во внутренний язык Ursus до непосредственно доказательства свойств кода. Однако, поскольку Ursus расширяет и использует средства Coq, во многих практических задачах непреодолимым препятствием становятся вышеобозначенные проблемы Coq

с производительностью.

При всех возможностях расширения Coq, порядки вычисления и унификации, приводящие к экспоненциальным проблемам, остаются "чёрным ящиком", и возможности пользователя повлиять на эти алгоритмы крайне ограничены. Даже нахождение причины экспоненциального поведения может потребовать глубокого исследования внутренних алгоритмов Coq [6]. Поэтому актуальной является задача по разработке более гибких стратегий, которые, используя нативные методы вычисления в Coq для небольших элементов, будут композироваться в итоговое вычисление специфичными для стратегии методами, позволяя тем самым пользователю влиять на производительность посредством выбора подходящей стратегии.

Центральной гипотезой этой работы заключается в том, что разработанные таким способом стратегии вычислений будут по крайней мере в некоторых случаях более производительными, чем нативные стратегии Coq. Эта работа фокусируется на практической задаче оптимизации для конкретного процесса в системе Ursus, на символьном вычислении результата функции относительно необходимого свойства. Этот процесс является центральным для доказательств — по окончании работы этого процесса, верификатор может работать с минимализированными формулами и утверждениями, относящимся непосредственно к доказываемому свойству программы, что качественно более удобно, чем доказательства с нуля. Вместе с этим, символьное вычисление на данный момент является самым вычислительно затратным процессом в Ursus. Таким образом, оптимизация этого процесса является ключевой практической задачей для системы Ursus.

Отдельный интерес представляет исследование эвристик, позволяющих тонкой настройкой улучшить производительность стратегий для повторяющихся закономерностей во входных данных для конкретной практической задачи. В этой работе будут исследованы эвристики, связанные с типами и с графовыми характеристиками данных.

Цель и задачи

Целью данной работы является оптимизация символьного вычисления результата функции в контексте системы Ursus. Для этого ставятся следующие задачи:

- Разработать несколько стратегий вычисления, используя классические порядки редукций и эвристики, продиктованные структурой специфических для системы Ursus данных.
- Подготовить набор программ на Ursus, включающий в себя фундаментальные конструкции языков программирования, для использования в качестве тестовых данных.

- Сравнить разработанные стратегии на тестовом наборе и выявить среди них наиболее производительные.

Обзор последующих глав

Глава 1 посвящена обзору теоретических основ, относящихся к области исследования, а также подробному описанию контекста работы.

В главе 2 описываются предложенные стратегии вычисления с различными эвристическими модификациями.

Глава 3 содержит описание тестового набора программ и сравнение предложенных стратегий с точки зрения производительности.

1. Обзор предметной области

В данной главе будет даны необходимые определения таких ключевых понятий, как лямбда-исчисление, подстановки и редукции, рассмотрены некоторые классические порядки редукций, такие как нормальный порядок и аппликативный порядок, а также поставлены вопросы относительно оптимальности этих порядков. Также в этой главе будет рассмотрен непосредственный контекст работы, а именно язык Coq, система Ursus и входные данные для рассматриваемой задачи.

1.1. Редукции термов в лямбда-исчислении

Лямбда-исчисление [9] — это математическая система, позволяющая формализовать вычислимость в функциональных языках программирования. Лямбда-исчисление определяется как множество *термов* T вместе с множеством *редукций* $\to$, описывающих допустимые преобразования термов. В нетипизированном случае множество термов можно определить как

$$T \ni t ::= \lambda x.\, t \mid x \mid t_1\ t_2 \tag{1}$$

Первый случай в этой записи называется *абстракцией* и описывает функцию, принимающую как аргумент некое выражение (переменную) x, второй случай описывает обращение к ранее объявленной в абстракции переменной, а третий случай называется *аппликацией* и описывает применение функции к аргументу. Чтобы формально описать семантику лямбда-исчисления, вводятся различные редукции, центральной из которых является β*-редукция* $\to_\beta$, формализующая вычисления. Чтобы определить β-редукцию, введём несколько вспомогательных понятий.

Определим *множество свободных переменных* $FV(t)$ как множество всех переменных в t, не имеющих соответствующей абстракции:

$$FV(x) = \{x\},\ FV(t_1\ t_2) = FV(t_1) \cup FV(t_2),\ FV(\lambda x.\, t) = FV(t) \setminus \{x\} \tag{2}$$

Заметим, что переменные в этой системе функционируют как способ связать абстракцию с её аргументом, а конкретные имена этих переменных нам не важны. Чтобы формализовать это наблюдение, определим α*-конверсию* как переименование переменной в абстракции на произвольную другую. Согласно конвенции Барендрегта [9], будем неявно применять α-конверсию, так чтобы все переменные имели различные имена, а α-эквивалентные (т.е. получаемые друг из друга с помощью α-конверсии) термы будем считать равными.

Определим *подстановку* $t[x := u]$ как замену всех вхождений переменной x в терме t на терм u:

$$
\begin{gathered}
x[x := u] = u \qquad y[x := u] = y \ (x \neq y) \\
t_1 \ t_2[x := u] = t_1[x := u] \ t_2[x := u] \\
\lambda x.\, t[x := u] = \lambda x.\, t \qquad (\lambda y.\, t)[x := u] = \lambda y.\, (t[x := u]) \ (x \neq y, y \notin FV(u))
\end{gathered} \tag{3}
$$

Заметим, что подстановка для некоторых аргументов может быть не определена из-за условия $y \notin FV(u)$ в последнем случае, но предварительное применение α-конверсии для аргумента u с целью использовать в нём только не используемые в t свободные переменные, позволяет доопределить подстановку.

Определим множество *редексов* терма t как множество всех подтермов t вида $(\lambda x.\, t_1)t_2$. Наличие редекса в терме показывает, что в терме есть функция, которую можно применить к аргументу. Наконец, мы можем определить β-редукцию как подстановку аргумента в произвольном редексе терма, которая оставляет остальную часть терма без изменений:

$$
(\lambda x.\, t_1)t_2 \to_\beta t_2[x := t_1] \tag{4}
$$

Определим $\to_\beta^*$ как транзитивное замыкание $\to_\beta$. Если $t_1 \to_\beta^* t_2$, то будем говорить, что t_1 *редуцируется* к t_2. Поскольку в терме может быть несколько редексов, *порядок* (или *стратегия*) *редукции*, то есть последовательность редексов в последовательном применении β-редукции, может быть различным. Этот факт порождает несколько ключевых вопросов: может ли от выбора стратегии зависеть результат, конечность процесса, и, наконец, оптимальность (т.е. количество шагов редукции).

Будем говорить, что терм t находится в *нормальной форме*, если к нему нельзя применить β-редукцию, то есть он не содержит редексов. Нормальная форма представляет собой конечный результат вычислений, подобно упрощенной до конкретного числа арифметической формуле. Важным свойством любой системы, основанной на лямбда-исчислении, является *конфлюэнтность*, то есть независимость результата от порядка редукций:

$$
t \to_\beta^* t_1, \ t \to_\beta^* t_2 \ \Rightarrow \ \exists t'.\, t_1 \to_\beta^* t', \ t_2 \to_\beta^* t' \tag{5}
$$

Напрямую из определения конфлюэнтности следует, что если система является конфлюэнтной, то терм не может редуцироваться к разным нормальным формам. Описанная выше простейшая система лямбда-исчисления является конфлюэнтной по теореме Черча-Россера, как и многие другие более сложные и применимые на практике системы, поскольку недетерминированные в этом смысле системы являются менее желательными на практике. Будем называть терм t' *нормальной формой* терма t, если t' находится в нормальной форме, а t редуцируется к t'. Задача поиска нормальной формы является ключевой при разработке семантики любого функционального

языка программирования, поскольку, в практических терминах, решение этой задачи эквивалентно написанию интерпретатора языка.

Для ответа на вопрос о завершимости введём понятия *сильной* и *слабой нормализуемости*. Терм t является слабо нормализуемым, если у него есть нормальная форма, и сильно нормализуемым, если любая стратегия редукции редуцирует его к нормальной форме за конечное число шагов β-редукции. Система является сильно (слабо) нормализуемой, если все термы в ней являются сильно (слабо) нормализуемыми.

Описанная выше простейшая система лямбда-исчисления не является ни сильно, ни слабо нормализуемой [9]. На практике это означает, что вычисление кода, соответствующего некоторым термам, может не завершиться, что в некоторых случаях нежелательно (а в слабом случае — завершимость этого вычисления может зависеть от порядка редукций). Данную проблему можно решать *типизацией* термов, т.е. рассмотрением только подмножества в каком-то смысле корректных термов. Так, просто типизированное лямбда-исчисление является сильно нормализуемым [9], однако в такой системе становится невозможно описать рекурсию. Более того, любой полный по Тьюрингу язык не может быть сильно нормализуемым, ведь завершимость всех программ на таком языке противоречила бы проблеме остановки.

Данная работа находится в контексте системы, которая является и конфлюэнтной, и сильно нормализуемой (см. ниже в разделе 1.3), то есть независимо от выбора стратегии редукции, любое вычисление придёт к единственно верному результату за конечное число шагов. Нас же интересует вопросы *оптимальности* стратегий, поскольку даже в системе, удовлетворяющим описанным выше теоретическим свойствам, термы могут редуцироваться к начальной форме за существенно отличающееся и значимое на практике число шагов.

1.2. Классические порядки редукций

Мы рассмотрим два классических порядка редукций — нормальный и аппликативный.

Аппликативный порядок редукции заключается в выборе наиболее вложенного редекса (а из редексов с равной вложенностью — самого левого). При таком порядке все аргументы абстракции будут вычислены до того, как подставлены. *Нормальный* порядок редукции заключается в выборе наименее вложенного редекса (аналогично, из редексов с равной вложенностью выбирается самый левый).

У этих стратегий есть *слабые* версии, отличающиеся ограничением на β-редукцию и соответственно определением нормальной формы. Для слабой стратегии запрещены редукции внутри абстракции, то есть, например, слабая стратегия рассматривает терм $(\lambda x.\,(\lambda y.\,y)x)$ как находящийся в нормальной форме. Слабая версия аппликатив-

ного порядка называется *вызов по значению* (*call-by-value*), а нормального — *вызов по имени* (*call-by-name*). Дальнейшие соображения высказаны для аппликативных и нормальных порядков, однако их можно также применить к вызову по значению и вызову по имени соответственно.

При условии наличия у терма нормальной формы нормальный порядок всегда редуцирует терм к ней, чего нельзя сказать про аппликативный порядок. Например, для терма $(\lambda x.\, y)(\omega\omega)$, где y — свободная переменная, а $\omega = \lambda z.\, zz$, нормальный порядок редуцирует внешний редекс, придя к нормальной форме y, а аппликативный порядок будет бесконечно редуцировать редекс $\omega\omega$, который редуцируется к себе же. Но даже в типизированных сильно нормализуемых системах, где невозможны бесконечные вычисления, производительность этих порядков на некоторых примерах может существенно отличаться. Например, при редукции терма $(\lambda x.\, T)M$, где T — произвольный корректно типизирующийся терм, а M — корректно типизирующийся терм большого размера, в аппликативный порядке терм M будет редуцирован, а в нормальном — нет. Термы, находящиеся в аналогичном M положении, будем называть *"мёртвым кодом"*.

В целом, преимущество нормального порядка по сравнению с аппликативным заключается в отбрасывании веток вычисления, которые не будут использованы в итоговом вычислении, "мёртвый код". Интуитивно, нормальный порядок является ленивым, то есть не будет вычислять аргументы абстракций до того, как они непосредственно понадобятся. Это позволяет ему не вычислять ветки вычисления, которые не используются в итоге, как и было проиллюстрировано в примере выше. На практике такие неиспользуемые ветки вычисления могут соответствовать, к примеру, проекциям структуры (в этом случае вычисления в полях, не задействованных в проекции, становятся бесполезными), или условным операторам с константным условием.

Недостатком же нормального порядка является возможное дублирование подтермов при подстановке, и, соответственно, дубликация работы при редукции этих подтермов. Для примера расширим лямбда-исчисление парами (t_1, t_2) и возьмём терм $P = (\lambda x.\, (x, x))$ типа $A \to A \times A$, строящий пару из двух одинаковых элементов. Рассмотрим редукцию терма $P((\lambda y.\, y)x)$:

$$\begin{gathered} P((\lambda y.\, y)x) \to_{\beta_{appl}} Px \to_{\beta_{appl}} (x, x) \\ P((\lambda y.\, y)x) \to_{\beta_{norm}} ((\lambda y.\, y)x, (\lambda y.\, y)x) \to_{\beta_{norm}} (x, (\lambda y.\, y)x) \to_{\beta_{norm}} (x, x) \end{gathered} \tag{6}$$

В нормальном порядке из-за преждевременной подстановки не приведённого к нормальной форме подтерма $((\lambda y.\, y)x)$ оказались удвоенными вычисления, направленные на редукцию этого терма. А если рассмотреть редукцию терма $\underbrace{PP\cdots P}_{n \text{ раз}}((\lambda y.\, y)x)$, то редукция в нормальном порядке повторит редукцию этого подтерма 2^n раз, то есть в таких случаях нормальный порядок будет экспоненциально менее производитель-

ным, чем аппликативный.

Чтобы решить эту проблему, в некоторых языках, например в Haskell [10], вводится мемоизация для результатов редукции подтермов, то есть результат однажды редуцированного подтерма используется при редукции его копий. Такая модификация вызова по имени называется *вызов по необходимости* (*call-by-need*). Такие техники могут быть реализованы достаточно сложными алгоритмами, добавляющими определённые накладные расходы на производительность и использование памяти, поэтому они используются не всегда, однако они решают описанное выше экспоненциальное замедление.

Haskell является одним из немногих языков с порядком вычислений, основанным на нормальном порядке. Большинство других языков программирования, как императивных, так и функциональных используют вызов по значению. В целом, наличие любых императивных примитивов в языке, которые потенциально могут приводить к побочным эффектам, делает применение нормального порядка намного менее привлекательным, поскольку при таком порядке трудно отследить последовательность побочных эффектов. Однако, многие языки, особенно функциональные, позволяют симулировать ленивые вычисления в нормальном порядке для лучшей выразительности и эффективности в отдельных случаях, тогда как вызов по значению используется по умолчанию. Примеры включают модуль `Lazy` в OCaml, конструкция `lazy val` в Scala или генераторы списков в Python.

1.3. Проблемы производительности в Coq

Интерактивное средство доказательств теорем Coq используется для доказательства математических теорем и формальной верификации программного кода. Теоретическая основа внутреннего языка Coq — *исчисление индуктивных конструкций* (*calculus of inductive constructions*), расширяющее *исчисление конструкций* индуктивно определёнными типами данных. Исчисление конструкций, в свою очередь, является типизированным лямбда-исчислением, лежащем в вершине $\lambda P\omega$ лямбда-куба [11], то есть система типов этого исчисления включает в себя зависимые типы, полиморфные типы и операторы над типами.

Исчисление индуктивных конструкций является конфлюэнтным, то есть результат редукции не зависит от порядка редукции, и сильно нормализуемым, то есть любой порядок редукции приведёт к нормальной форме за конечное число шагов [12]. В отличие от многих языков общего назначения, Coq не является Тьюринг-полным языком, но при этом позволяет использовать рекурсивные функции при условии, что будет доказана их завершаемость. Это необычное свойство языка продиктовано необходимостью консистентности внутренней логики, однако оно также даёт максимальную свободу при выборе стратегии редукции — поскольку любая стратегия приведёт к

результату за конечное время, можно разработать какие угодно стратегии и выбрать из них наиболее производительную.

Coq предоставляет возможность использовать вызов по необходимости (тактика `lazy`) или вызов по значению (тактика `cbv`). Разработчики языка утверждают, что стратегия `cbv` будет эффективнее в чисто вычислительных случаях, то есть почти не содержащих "мёртвый код", а стратегия `lazy` может демонстрировать лучшую производительность, когда результат можно получить без полного вычисления всех подвыражений, например, при наличии "мёртвого кода" или ветвлений, вычисление которых можно отложить [1], что согласуется с теоретическими соображениями, описанными в разделе 1.2. Во внутренних для системы редукциях при финальной проверки типов доказательства (команда `Qed`) используется call-by-need стратегия.

Однако, к сожалению, ограничиться выбором подходящей для входных данных стратегии из этих двух вариантов недостаточно. При разработке практических и масштабируемых проектов на Coq нередко можно наткнуться на критические проблемы производительности, вызванные ошибками или особенностями в коде Coq. Например, хотя в Coq применяется вызов по необходимости, мемоизированный вариант вызова по имени, в некоторых ситуациях эквивалентность дублирующихся термов не распознаётся, что приводит к повторной редукции одних и тех же выражений [13]. Выше уже было обозначено, к каким нежелательным последствиям может привести отсутствие мемоизации для вызова по имени. В другом случае, из-за недостаточной информации о типах вычисления приобретают экспоненциальную сложность [14].

Более подробному исследованию проблем с производительностью в Coq посвящена диссертация Гросса [6]. В ней, помимо описания множества проблем, подобным двум описанным выше, утверждается, что диагностика таких проблем и их исправление требуют значительного погружения в детали реализации Coq и экспертного знания относительно устройства этой реализации. Такое погружение выходит за рамки данной работы, поэтому в этой работе во многом внутренняя реализация Coq будет восприниматься как "чёрный ящик". Тем не менее будет продемонстрировано, что даже при таком упрощённом подходе, различные алгоритмы композиции внутренних механизмов Coq показывают результаты, на данной задаче значительно превосходящие эти механизмы по производительности.

Отметим также работы по оптимизации редукций, которые добавили в Coq редукции `vm_compute` [15] и `native_compute` [16]. Эти реализации вызова по значению, ориентированные на эффективность и работу с высоконагруженными вычислениями, значительно превосходят по производительности редукцию `cbv`. Однако, эти редукции было бы затруднительно использовать для символьного вычисления, то есть в контексте этой работы, поскольку для символьного вычисления критически важно объявлять некоторые определения как *непрозрачные* (*opaque*), которые не будут раскрываться в процессе вычисления. Например, на некоторых этапах намного полезнее

работать со сложением как с абстрактным символом `+`, чем с буквальным определением сложения, а при раскрытии таких конструкций крайне затруднена автоматизация. Редукции `vm_compute` и `native_compute` однако игнорируют непрозрачность определений и редуцируют термы до конца, а потенциальное расширение этих реализаций поддержкой непрозрачности может сильно замедлить эти алгоритмы [17]. Поэтому в рамках этой работы не будут использоваться эти тактики, несмотря на их эффективность.

Язык Ltac позволяет расширять набор тактик Coq, то есть создавать программы для автоматической верификации. При этом на него не накладываются ограничения на завершимость, поэтому разработка алгоритмов на нём более напоминает разработку на традиционных языках. В этой работе Ltac будет использован для разработки специализированных стратегий редукции.

1.4. Система Ursus

Система Ursus разработана как расширение системы Coq и предназначена для верификации смарт-контрактов на различных языках. Она включает в себя:

- внутренний монадический язык;
- транслятор, позволяющий автоматически получать из исходного кода смарт-контрактов на языках Solidity, Rust, FunC аналогичный код на внутреннем языке Ursus;
- аналоги стандартных библиотек для этих языков, сформулированных на Coq или Ursus, с заранее доказанными фактами о содержимом этих библиотек;
- интерпретатор для языка Ursus;
- язык для написания спецификаций;
- *генератор вычислений* (*exec generator*), позволяющий преобразовывать функции на языке Ursus в набор уравнений, описывающих изменения в системе на каждом шагу (с автоматически доказанной эквивалентностью результату интерпретации функции);
- алгоритм, который позволяет упростить систему уравнений, полученную из генератора вычислений, относительно конкретного элемента спецификации, тем самым упростив работу верификатора (символьное вычисление).

Эта работа посвящена оптимизации последнего шага из списка выше, а поскольку этот шаг принимает как входные данные систему уравнений из генератора вычислений, необходимо остановиться на его описании подробнее. Проиллюстрируем работу

генератора вычислений на простом примере. Возьмём небольшую функцию на исходном языке Solidity:

```
uint64 result;
function f(uint64 a, uint64 b) {
    uint64 x = result;
    x = x + a;
    x = x - b;
    result = x;
}
```

После того как эта функция будет транслирована на язык Ursus, генератор вычислений создаст на её основе следующую систему уравнений (конкретные выражения заменены текстовыми описаниями для краткости):

```
y1 = стартовое состояние системы
y2 = указатель на глобальную переменную result
y3 = y1, в котором создана новая переменная x со значением y2
y4 = указатель на состояние переменной x в y3
y5 = значение указателя y4
y6 = y5 + a[1]
y7 = y3, в котором значение переменной x заменено на y6
y8 = указатель на состояние переменной x в y7
y9 = значение указателя y8
y10 = y9 - b[1]
y11 = y7, в котором значение переменной x заменено на y10
y12 = указатель на состояние переменной x в y11
y13 = значение указателя y12
y14 = y11, в котором значение глобальной переменной result заменено на y13
y15 = y14 (итоговое состояние системы)
```

Такая низкоуровневая декомпозиция исходной функции отдалённо напоминает код на языках ассемблера. В отличие от них, здесь отсутствует какой-либо стек или хранилище данных, а каждая очередная вводимая формула может использовать значения всех формул, введённых выше.

Теперь перейдём к верификации корректности этой функции. Единственный наблюдаемый эффект от запуска этой функции — изменение значения глобальной переменной `result`. То есть, полная спецификация корректности этой функции заключается в следующем факте: если значение `result` в стартовом состоянии `y1` равно

[1] в целях упрощения примера будем считать, что переполнение численных переменных невозможно

r, то значение `result` в финальном состоянии `y15` должно равняться $r + a - b$. Для того чтобы доказать этот факт, необходимо упростить систему путём подстановки определений y_i и редукции соответствующих термов.

Формальная постановка задачи

Входные данные:

1. Система уравнений Y вида $y_i = T_i(y_1, y_2, \ldots, y_{i-1})$, где T_i — некоторый терм, который может содержать переменные y_j для $j < i$.
2. Спецификация $P(y_1, y_2, \ldots, y_n)$, где P это произвольный терм, который может содержать любые переменные y_i.

Задача упрощения системы: выполнить редукцию терма спецификации $P(y_1, y_2, \ldots, y_n)$ с использованием определений y_i.

Зачастую в реальных задачах, как и в рассматриваемом примере, спецификация имеет вид $P(y_1, y_n)$, то есть является отношением между стартовым и финальным состоянием, описывающим поведение функции. Однако, в некоторых случаях, если для спецификации важно промежуточное состояние, например при вызове внутренней функции после изменения глобального состояния, в спецификации также могут использоваться переменные y_i для $1 < i < n$. Руководствуясь этими практическими соображениями, будем решать задачу в общем виде.

Заметим, что спецификация для этой функции является достаточно тривиальной, и её верность окажется очевидной, как только будут упрощена система уравнений. Действительно, в этом случае после упрощения достаточно будет применить тактику для доказательства тривиальных фактов `auto`, которая доказывает цели, выводимые из текущих гипотез и стандартных лемм по правилам логики первого порядка и простым уравнениям. Тем не менее даже если спецификация является более сложной, упрощение системы уравнений значительно упростит задачу верификатора, так как специализирует код функции относительно необходимого среза состояния. Например, если свойство спецификации описывает изменение в функции только одной переменной, в результате упрощения все изменения других переменных, не влияющих на специфицированную, будут исключены.

Одно из преимуществ описанной декомпозиции заключается в возможности тонко и точечно настраивать и применять редукцию и вычисления, что и будет продемонстрировано в этой работе. А именно, при разработке стратегий вычисления в следующей главе, нативные стратегии редукции (`lazy` и `cbv`) будут применяться лишь для конкретных уравнений и для вычисления итогового результата, в то время как разработанные стратегии будут задавать порядок вычислений, руководствуясь структурой данных, размером входа и другой информацией о конкретной задаче.

1.5. Выводы и результаты по главе

Определено нетипизированное лямбда-исчисление, на его примере определены подстановки и β-редукция. Сформулированы следующие порядки редукций: нормальный порядок и вызов по имени, аппликативный порядок и вызов по значению. Аппликативный порядок может быть неэффективен в случае "мёртвого кода", а нормальный — в случае дубликации аргументов. Для частичного решения проблем нормального порядка применяется вызов по необходимости. В языке Coq используются оба подхода. Из-за многочисленных и сложных в диагностике и исправлении проблем в производительности Coq было выбрано относиться к нативным редукциям как к "чёрному ящику". Описана система Ursus в целом и входные данные для задачи упрощения системы формул в частности. Для оптимизации этой задачи будем использовать нативные редукции на декомпозированых элементах, а результаты композировать различными, разработанными в следующей главе способами.

2. Разработка стратегий

Данная глава, центральная часть этой работы, посвящена алгоритмическому описанию различных стратегий вычисления, которые будут применяться для упрощения системы уравнений. Сначала будут описаны простейшие нативные стратегии, затем они будут усложняться добавлением новых эвристик и оптимизаций. Отдельно будет описана работа с условными операторами, порождающими несколько веток вычисления.

2.1. Анализ входных данных

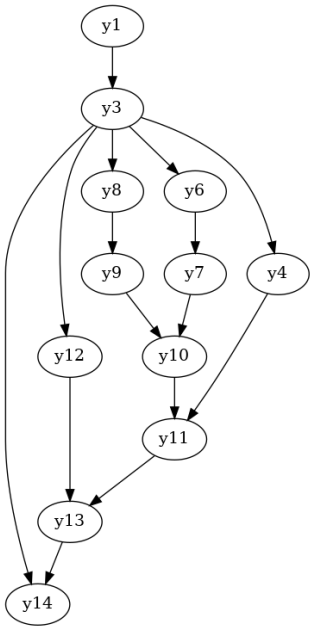


Рис. 1: Графовый вид простейшей системы

В предыдущей главе было описаны два принципиально отличающихся порядка вычисления: нормальный (в Coq тактика `lazy`) и аппликативный (в Coq тактика `cbv`) порядки, а также приведены их существенные недостатки. Рассмотрим, применимы ли эти недостатки к изучаемой задаче.

Основной недостаток аппликативного порядка — неэффективная работа с "мёртвым кодом", блоками кода, результат редукции которых не влияет на итоговое вычисление. Могут ли такие блоки кода присутствовать во входных данных? Напомним, что входные данные состоят из набора уравнений $y_i = T_i(y_1, \ldots, y_{i-1})$ и спецификации $P(y_1, \ldots, y_n)$. Спецификация P формально может быть любым термом, однако на практике в каждом конкретном доказательстве верификатора интересуют только некоторый срез системы, поведение отдельных частей программы. Из этого следует, что те части программы, которые не относятся к верифицируемому свойству, в терме $P(y_1, \ldots, y_n)$ как раз и окажутся "мёртвым кодом". Вдобавок, при определённых стартовых состояниях y_1 некоторые ветки кода, отражённые в последующих уравнениях $y_i = T_i$, могут оказаться нерелевантными. К примеру, верификатора интересуют только стартовые состояние, при которых некоторое булево поле равно `true`, тогда `else`-ветки соответствующих условных операторов окажутся мёртвым кодом. Таким образом, недостатки аппликативного порядка применимы к этой задаче в полной мере.

Основным недостатком же нормального порядка (особенно версии без мемоизации) является дублирование работы при преждевременной подстановке аргументов. Поскольку определение рассматриваемой системы уравнений разрешает в T_i использовать любые y_j для $j < i$, такое дублирование в теории может быть значительным. Чтобы изучить свойства реально рассматриваемых систем, был разработан графовый визуализатор. На рисунке 1 представлен графовый вид системы уравнений, постро-

енной по простейшему линейному блоку кода, а именно по тесту из множества Simple при n = 1 (в этой главе будут приводиться в качестве примеров тесты из множеств Simple, Recursion, If и IfAndRecursion, подробнее эти множества будут описаны в разделе 3.1). Ребро $y_i \to y_j$ означает, что в T_j используется y_i. Заметим, что поскольку T_j не может ссылаться на y_i, если $j \leq i$, то граф является ациклическим ориентированным графом.

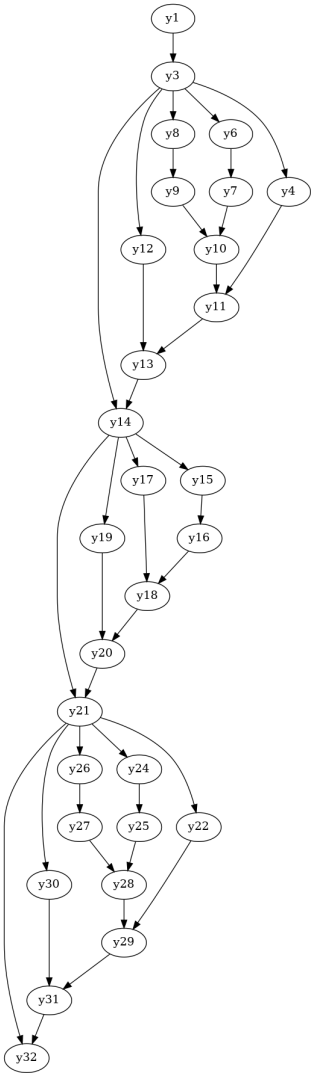


Рис. 2: Графовый вид системы из двух блоков кода

Если бы этот граф имел вид "бамбука", то есть если бы исходящая и входящая степень каждой вершины была бы не больше одного, то можно было бы заключить, что нормальный порядок подходит отлично, так как подстановка не дублирует работу и мемоизация не требуется. Однако даже на простейшем линейном коде наблюдается достаточно ветвистый граф. А именно, существуют пять различных путей из y_3 в итоговое состояние y_{14}, то есть в худшем случае работа по редукции y_3 будет продублирована пять раз. Заметим также, что y_3 соответствует глобальному состоянию системы, то есть является относительно большим термом, и дублирование его редукции даже несколько раз нежелательно.

На рисунке 2 представлен граф, построенный на основе двух блоков кода из предыдущего примера, с небольшой соединительной конструкцией между ними (тест из множества Simple при n = 2). Этот граф иллюстрирует, что число путей из стартовой вершины в итоговую, а значит, и потенциальное дублирование работы, становится экспоненциальным при увеличении размеров кода. Напомним, что код для этих иллюстраций взят упрощенный, на реальных примерах, особенно с разветвлением кода по условным операторам дублирование может быть ещё сильнее. Таким образом, при выборе нормального порядка необходима мемоизация, иначе замедление будет экспоненциальным, что недопустимо.

Итак, оба подхода имеют свои недостатки, и предварительный анализ входных данных показывает, что для рассматриваемой задачи оба недостатка — "мёртвый код" и дублирование при подстановке — применимы в полной мере. Дублирование при подстановке частично решается мемоизацией, однако эта техника создаёт определённые накладные расходы, и до перехода к экспериментам неочевидно, будет ли эти расходы стоить оптимизации по "мёртвому коду" по сравнению с аппликативным порядком. Соответственно, при дальнейшей разработке стратегий будут использоваться оба порядка и идеи, связанные с каждым из них.

2.2. Базовые стратегии

Прежде чем приступить к разработке стратегий, заметим, что можно исключить из рассмотрения все уравнения, которые в итоге не используются в P, тем самым избавившись от "мёртвого кода" на уровне уравнений. Это можно сделать эффективно (по отношению к остальной работе), применив следующий трюк: поскольку встроенная тактика `clear` завершится с ошибкой, если удаляемый элемент контекста используется в типах или значениях других элементов, последовательным применением этой тактики можно найти все неиспользуемые элементы. В дальнейшем будем считать, что все y_i транзитивно задействованы в P.

Первой рассматриваемой стратегией является простая стратегии `native`, включающая в себя только один запуск редукции. Приведём набор уравнению к *let-форме*, то есть воспользуемся конструкцией `let` внутреннего языка Coq и преобразуем каждое уравнение $y_i = T_i$ в часть единого терма `let` $y_i = T_i$ `in` $\ldots$, а после всех `let-in` конструкций вставим верифицируемое свойство P. Получив из входных данных единый терм, редуцируем его с помощью нативных редукций `lazy` или `cbv`.

Стратегия 1
native-lazy

$(Y, P) \leftarrow$ input
$F(x) \leftarrow x$
for $i \leftarrow 1$ to $|Y|$ **do**
    $F(x) \leftarrow F(\texttt{let}\ x_i = T_i\ \texttt{in}\ x)$
    for $j \leftarrow i + 1$ to $|Y|$ **do**
        $T_j \leftarrow T_j[y_i := x_i]$
return $lazy(F(P))$

Стратегия 2
native-cbv

$(Y, P) \leftarrow$ input
$F(x) \leftarrow x$
for $i \leftarrow 1$ to $|Y|$ **do**
    $F(x) \leftarrow F(\texttt{let}\ x_i = T_i\ \texttt{in}\ x)$
    for $j \leftarrow i + 1$ to $|Y|$ **do**
        $T_j \leftarrow T_j[y_i := x_i]$
return $cbv(F(P))$

Как было аргументированно в разделе 2.1, трудно решить без экспериментальных данных, какая из двух нативных редукций покажет лучший результат на практике.

Поэтому все рассматриваемые стратегии будут иметь две версии и называться либо `*-lazy`, либо `*-cbv`. Для краткости в дальнейшем будет определяться только одна версия каждой стратегии, а в псевдокоде использоваться вызов $reduce$, который может соответствовать `lazy` или `cbv` в зависимости от версии.

Стратегия `native` является самым простым и общим решением задачи. Однако, дальнейшая оптимизация этого решения затруднительна, поскольку возможности пользователя повлиять на производительность нативной редукции ограничены.

Попробуем использовать идеи вызова по значению и вызова по необходимости на уровне уравнений. Следующая стратегия `bottomup`, подражая вызову по необходимости, начинает с конечного терма P и подставляет в него все значения y_i, а затем вычисляет получившийся терм. При этом подставляться будут только те y_i, которые не используются в других уравнениях, то есть начиная с y_n и до y_1 (отсюда название стратегии, если предположить, что система уравнений записана сверху вниз).

Стратегия 3
bottomup

$(Y, P) \leftarrow$ input
for $i \leftarrow |Y|$ to 1 **do**
    $P \leftarrow P[y_i := T_i]$
return $reduce(P)$

Такая стратегия вряд ли будет применима на практике, поскольку не показывая никаких улучшений относительно стратегии `native`, она вносит экспоненциальное разрастание терма за счёт подстановок до вычислений. А поскольку на каждом шаге происходит подстановка терма, это приведёт к экспоненциальному замедлению алгоритма. Частично эту проблему можно решить редукцией перед каждой подстановкой:

Стратегия 4
bottomup-reductions

$(Y, P) \leftarrow$ input
for $i \leftarrow |Y|$ to 1 **do**
    $T_i \leftarrow reduce(T_i)$
    $P \leftarrow P[y_i := T_i]$
return $reduce(P)$

Однако, это не соответствует идее мемоизации в вызову по необходимости, а больше напоминает вызов по значению. В итоге такая стратегия подвержена минусам обоих подходов. Поскольку редуцируемые термы всё ещё содержат свободные переменные, которые могут находиться, например, в голове конструкций `if` или `match`, то в терме может остаться "мёртвый код". С другой стороны, из-за этого "мёртвого

кода" терм может увеличиться в размерах, что при его дубликации при подстановке может замедлить дальнейшие подстановки.

Попробуем теперь производить вычисления в обратном порядке, двигаясь "сверху вниз", так что все вычисляемые уравнения не имеют свободных переменных к моменту вычисления, то есть могут быть вычислены до конца. Это крайне похоже на вызов по значению — в такой стратегии все "аргументы" полностью вычисляются до подстановки.

Стратегия 5
topdown

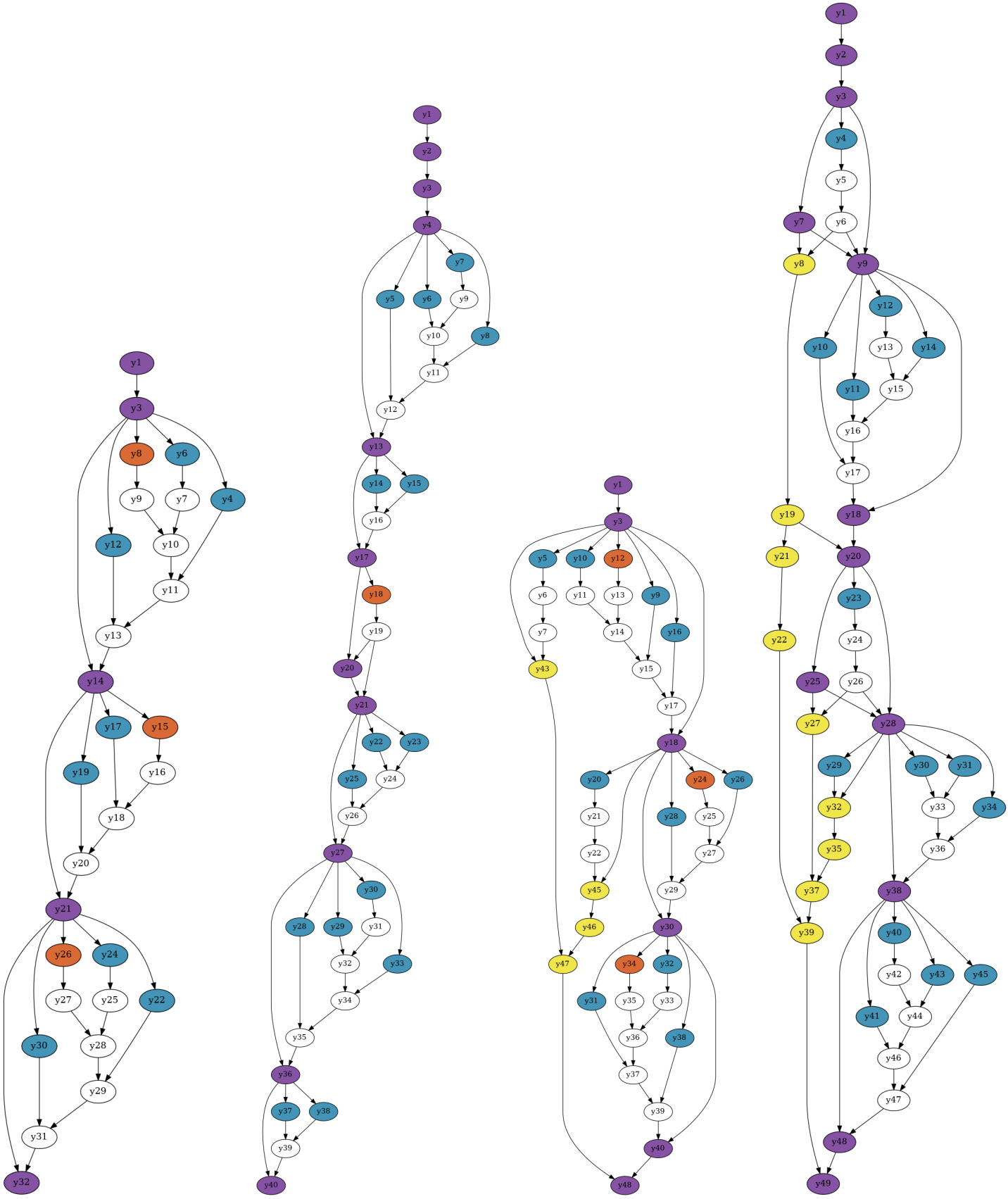
$(Y, P) \leftarrow$ input
for $i \leftarrow 1$ to $|Y|$ **do**
  $T_i \leftarrow reduce(T_i)$
  for $j \leftarrow i + 1$ to $|Y|$ **do**
    $T_j \leftarrow T_j[y_i := T_i]$
  $P \leftarrow P[y_i := T_i]$
return $reduce(P)$

Такая стратегия интересна тем, что на каждом шаге промежуточное вычисление носит окончательный характер из-за отсутствия свободных переменных, а разбиение единой редукции на множество небольших может позволить уменьшить эффект от неэффективности в реализации редукций.

2.3. Графовые оптимизации

До сих пор разрабатываемые алгоритмы имели достаточно общий характер, хоть и иногда сопровождались комментариями и аргументацией относительно конкретных данных. В этом и следующем разделе будут исследованы оптимизации, основанные непосредственно на строении данных. Эти оптимизации будут сфокусированы на предварительном упрощении задачи перед запуском одной из базовых стратегий. А именно, описываемые далее алгоритмы преобразовывают исходные данных (Y, P) в эквивалентные (с точки зрения результата редукции) (Y', P'), а затем решают задачу окончательно с помощью одной из стратегий из предыдущего раздела $basic\_strategy(Y', P')$.

Для того чтобы разработать метод для упрощения задачи, вернёмся к анализу данных на основе графовой визуализации, подобно описанному в разделе 2.1. На рисунке 3 представлены графовые виды систем уравнений, полученных из разных программ (тесты из всех множеств, описанных в разделе 3.1 при n = 2). Цвета вершин соответствуют типам данных и будут рассмотрены ниже в разделе 2.4.



(a) Simple (b) Recursion (c) If (d) IfAndRecursion

Рис. 3: Графовые виды различных систем уравнений

С помощью изучения подобных графов было найдено несколько интересных с точки зрения оптимизации свойств. Так, оказалось, что около половины вершин имеют и входящую, и исходящую степень равную единице, то есть соответствующие y_i используются только в одном T_j, а T_i содержит только один y_k. В области графовых алгоритмов для уменьшения задачи применяют *стягивание* таких вершин, то есть рёбра $u \to w$ и $w \to v$ заменяются на одно ребро $u \to v$, а вершина w удаляется. Интерпретируя этот подход для системы уравнений, получим оптимизацию, которая заранее подставляет тела все стягиваемых уравнений, не вычисляя их, и удаляет соответствующие уравнения:

Стратегия 6
contractions

$(Y, P) \leftarrow$ input
$T \leftarrow P(y_0, y_n)$
for $i \leftarrow 1$ to $|Y|$ **do**
  if $indeg(y_i) = 1$ and $outdeg(y_i) = 1$ **then**
    for $j \leftarrow i + 1$ to n **do**
      $T_j \leftarrow T_j[y_i := T_i]$
    $P \leftarrow P[y_i := T_i]$
    $Y \leftarrow Y \setminus (y_i, T_i)$
return $basic\_strategy(Y, P)$

В разделе 2.1 было установлено, что граф является достаточно ветвистым, и есть экспоненциальное относительно числа вершин число путей из начальной вершины в конечную. Однако, при дальнейшем анализе графов оказывается, что эта ветвистость достигается за счёт относительно небольшого числа вершин с большой исходящей степенью. При этом подавляющее большинство вершин имеют исходящую степень, равную единице, то есть для большинства y_i верно, что y_i используется только в одном T_j. При подстановке таких y_i может возникнуть дубликация, однако дальнейший анализ графов показывает, что эта дубликация не будет носить экспоненциальный характер. Вершины, имеющие исходящую степень, большую чем один, разбивают граф на несколько блоков, при этом каждый блок содержит небольшое количество вершин. Таким образом, при стягивании вершин из такого блока, хоть дубликация и может возникнуть, но она не будет распространяться между блоками и приумножаться. Согласно этим соображениям, интересной для исследования является также оптимизация `contractions-strong`, заключающаяся в предварительной подстановке без редукции всех вершин, имеющих исходящую степень один.

Стратегия 7
contractions-strong

$(Y, P) \leftarrow$ input
for $i \leftarrow 1$ to $|Y|$ **do**
  if $outdeg(y_i) = 1$ **then**
    for $j \leftarrow i + 1$ to n **do**
      $T_j \leftarrow T_j[y_i := T_i]$
    $P \leftarrow P[y_i := T_i]$
    $Y \leftarrow Y \setminus (y_i, T_i)$
return $basic\_strategy(Y, P)$

Такие оптимизации могут уменьшить размер задачи и тем самым способствовать построению более эффективных стратегий, однако процедура определения графовых свойств, то есть подсчёта функций *outdeg* и *indeg*, затруднительна и может занимать значительное время. Ясно, что вычислять явно степени вершин на каждом шаге неэффективно, но даже предподсчёт с явным построением графа вносит в алгоритм определённые накладные расходы, усложнённые программным окружением алгоритма, а именно отсутствием памяти с произвольным доступом как таковой — единственное подобием памяти при разработке на Ltac является интерактивный контекст доказательства, поиск по которому может быть долгим при больших размерах этого контекста. Не исключено, что эти накладные расходы могут перекрыть выигрыш от оптимизации.

2.4. Оптимизации, основанные на типах данных

В предыдущем разделе представлены две оптимизации, потенциально способные ускорить вычисления на данных, имеющих определённую структуру, и поставлен вопрос относительно их практической применимости в связи с накладными расходами на явное построение графа и вычисление степеней вершин. В этом разделе будут разработаны оптимизации, которые используют информацию о типах данных как приближение к графовым свойствам, необходимым для принятия решений о стягивании вершин, что позволяет избежать явного построения графа.

Рассмотрим снова рисунок 3, обратив внимание на окраску вершин. Она отражает информацию о типах данных y_i, а именно:

- Фиолетовые вершины имеют тип `Ledger`, описывающий глобальное состояние системы. Этот тип включает в себя, помимо состояния глобальных переменных, также данные о других смарт-контрактах, балансах и конфигурации сети, поэтому данные такого типа, как правило, являются относительно большими термами.

- Жёлтые вершины имеют тип `ControlResult` и связаны с порядком выполнения (control flow) программы, в том числе с такими конструкциями, как обработка ошибок и досрочный выход из функции.

- Голубые вершины имеют тип `FieldType(F)` и описывают проекции структур, то есть чтение отдельного поля `F` у структур данных (в том числе, но не только, у глобального состояния типа `Ledger`).

- Оранжевые вершины имеют тип `LocalStateMapping(T)` и соответствуют получению информации о состоянии локальных переменных типа `T`.

- Нераскрашенные вершины имеют пользовательские типы, соответствующие типам в исходной программе, например, численные или булевы.

В результате анализа графов были сформулированы следующие утверждения о соответствии типов вершин и их графовых свойствах:

1. Все вершины типа `LocalStateMapping` (оранжевые) и `FieldType` (голубые) имеют входящую и исходящую степень, равную единице.

2. Почти все вершины типа `ControlResult` (жёлтые) имеют исходящую степень, равную единице.

3. Вершины типа `Ledger` (фиолетовые), как правило, имеют большую исходящую степень.

4. Почти все вершины, имеющие исходящую степень, большую чем один, являются вершинами типа `Ledger` (фиолетовыми).

Утверждение (1) было установлено верным в результате анализа алгоритма, строящего систему уравнений, и его реализации. Остальные утверждения не носят точный характер, только приблизительный, однако анализ алгоритма построения показал, что исключения из этих наблюдений встречаются только в отдельных и достаточно редких случаях.

Из утверждения (1) напрямую следует аппроксимированная версия оптимизации `contractions` под названием `contractions-typebased`, которая будет стягивать не все вершины с входящей и исходящей степенью, равной единице, а только вершины с типами `LocalStateMapping` и `FieldType`. Несмотря на то, что будет стянуто меньше вершин, такая оптимизация все равно может оказаться эффективной, и не будет тратить время на явный анализ графовых свойств.

Стратегия 8
contractions-typebased

$(Y, P) \leftarrow$ input
for $i \leftarrow 1$ to $|Y|$ **do**
  if $typeof(y_i)$ is $LocalStateMapping$ or $FieldType$ **then**
    for $j \leftarrow i + 1$ to n **do**
      $T_j \leftarrow T_j[y_i := T_i]$
    $P \leftarrow P[y_i := T_i]$
    $Y \leftarrow Y \setminus (y_i, T_i)$
return $basic\_strategy(Y, P)$

Из утверждений (3) и (4) также следует аппроксимированная версия сильной оптимизации `contractions-strong` под названием `contractions-strong-typebased`, которая будет стягивать все вершины, кроме вершин типа `Ledger`. Несмотря на то, что эта аппроксимация может ошибаться с двух сторон, так как могут существовать как вершины типа `Ledger` с исходящей степенью, равной единице, так и вершины других типов, имеющие большую исходящую степень, утверждается, что таких вершин достаточно мало в реальных данных, и что аппроксимация будет достаточно эффективна. Вдобавок, термы типа `Ledger`, как уже отмечалось выше, являются относительно большими, и не подставлять их разумно ещё и из-за этого соображения.

Стратегия 9
contractions-strong-typebased

$(Y, P) \leftarrow$ input
for $i \leftarrow 1$ to $|Y|$ **do**
  if $typeof(y_i)$ is not $Ledger$ **then**
    for $j \leftarrow i + 1$ to n **do**
      $T_j \leftarrow T_j[y_i := T_i]$
    $P \leftarrow P[y_i := T_i]$
    $Y \leftarrow Y \setminus (y_i, T_i)$
return $basic\_strategy(Y, P)$

Таким образом, стратегии, описанные в этом разделе, также используют графовые свойства системы, но косвенно, по типовым признакам, сопряжённым с графовыми свойствами, в целях эффективного определения этих свойств.

2.5. Работа с условными операторами

При наличии в коде условных операторов `if` или аналогичных конструкций, вычисление немного меняется. Следует отдельно рассматривать ветки исполнения, в

которых соответствующие булевы термы равняются `true` или `false`. В Coq это можно сделать с помощью встроенной тактики `destruct`, разбивающей текущее вычисление на две копии, отличающиеся только предположением относительно значения рассматриваемого булева терма.

Поскольку вычисление дублируется для каждого булева оператора, неминуемо порождается значительная экспоненциальная дубликация работы. Разумеется, в конкретных случаях, когда число веток вычисления крайне велико, возможно избежать экспоненциального взрыва такими техниками, как удаление невозможных веток или разбиение функции на подфункции с доказательством промежуточных инвариантов. Однако, такие техники, как правило, являются ручной ad-hoc работой, специфичной для каждой рассматриваемой функции, поэтому в рамках этой работы будет предложено общее решение, которое пусть и является экспоненциальным от количества условных операторов, но избегает излишней дубликации, то есть не вызывает `destruct` до того, как это необходимо для сохранения инвариантов стратегии.

Для начала отметим, что эвристики, основанные на стягивании, не должны подставлять термы, содержащие условный оператор, чтобы не допустить даже минимальной дубликации этих операторов (в случае сильных версий) и для упрощения дальнейшего нахождения этих операторов. То есть, эти оптимизации будут модифицированы добавлением в условие подстановки утверждения "не является условным оператором". Базовые стратегии же будут модифицированы следующим образом:

- В стратегиях `topdown` и `bottomup-reductions` необходимо сохранить инвариант о том, что на момент редукции определённого уравнения все уравнения выше уже полностью обработаны. Поэтому если стратегия доходит до уравнения, содержащего условный оператор, оно вычисляется, вызывается `destruct` и в каждой из веток вычисления подставляется, а дальнейшие вычисления происходят в каждой ветке отдельно.

- В стратегиях `native` и `bottomup` такого инварианта нет, поэтому можно вызвать `destruct` позже. В этих стратегиях вначале будет вызван стандартный проход алгоритма, построение терма с помощью let-конструкции или подстановки соответственно, за тем исключением, что уравнения, содержащие условные операторы, будут пропущены и останутся в контексте. Далее, второй проход для каждого оставшегося уравнения редуцирует его и вызывает `destruct`, и, наконец, итоговый терм вычисляется в каждой ветке вычислений. Промежуточные редукции, которых нет в базовой версии этих стратегий, здесь необходимы из-за того, что для корректной работы `destruct` термы в уравнениях и в итоговом терме должны быть приведены к единой форме (в данном случае, к нормальной форме).

Поскольку описываемые модификации оказывают минимальное воздействие на поведение и производительность стратегий в случаях, когда данные не содержат условных операторов, не является необходимым реализовывать отдельные модификации стратегий для данных с условными операторами, а в экспериментальной части будут измеряться результаты только для стратегий с описанными в этом разделе модификациями.

2.6. Выводы и результаты по главе

Произведён анализ входных данных, и на основе этого анализа разработаны стратегии вычисления. Представлено четыре базовые стратегии:

- `native`, строит единый терм с помощью let-конструкций и затем вычисляет его единой редукцией;
- `bottomup`, подставляет определения используемых переменных снизу вверх и затем вычисляет единой редукцией;
- `bottomup-reductions`, подставляет определения используемых переменных снизу вверх и вычисляет промежуточный результат на каждом шаге;
- `topdown`, подставляет определения переменных сверху вниз и вычисляет промежуточный результат на каждом шаге.

Каждая из этих стратегий может применяться либо сама по себе, либо после предварительного использования одного из четырёх алгоритмов эвристического упрощения:

- `contractions`, подставляет определения переменных, использующихся только в одном уравнении и использующие в своём определении только одну переменную;
- `contractions-strong`, подставляет определения переменных, использующихся только в одном уравнении;
- `contractions-typebased`, подставляет определения переменных типов `LocalStateMapping` и `FieldType`;
- `contractions-strong-typebased`, подставляет определения всех переменных, кроме переменных типа `Ledger`.

Вдобавок, каждая стратегия может использовать одну из двух нативных редукций, `cbv` или `lazy`. Таким образом, всего представлено $4 \cdot 5 \cdot 2 = 40$ стратегий. Также описаны модификации стратегий, необходимые для поддержки условных операторов. Все эти стратегии были реализованы на языке Ltac.

3. Сравнение стратегий

Данная глава посвящена сравнению описанных выше стратегий. В ней описывается подготовка необходимых тестовых данных, приведены результаты экспериментов с применением стратегий на этих данных, и сделаны соответствующие выводы.

3.1. Подготовка тестовых данных

Тестовые данные для этой работы состоят из двух частей. Первая, синтетическая, была написана специально для этой работы и представляет собой несколько наборов функций, в каждом из которой размер и сложность функций равномерно увеличивается по заданному шаблону. Вторая часть взята из реальной практики верификации смарт-контрактов.

Для того чтобы синтетические данные подходили для численных экспериментов, необходимо было создать логически несложный код, который может повторяться с незначительными изменениями неограниченное число раз. Вместе с этим необходимо было протестировать такие фундаментальные элементы языков программирования, как условные переходы и рекурсия. В то же время циклы не были включены в этот код, поскольку верификация кода с циклами является отдельной областью исследований, ставящих множество уникальных вопросов [18], и выходит за рамки этой работы.

Для построения синтетических тестовых данных был выбран классический алгоритм подсчёта полиномиального хеша строки. Этот алгоритм был реализован на языке Ursus в различных вариациях. При этом для этих алгоритмов верифицируется соответствие реализации референсной реализации на языке Coq. Синтетический набор состоит из следующих наборов функций:

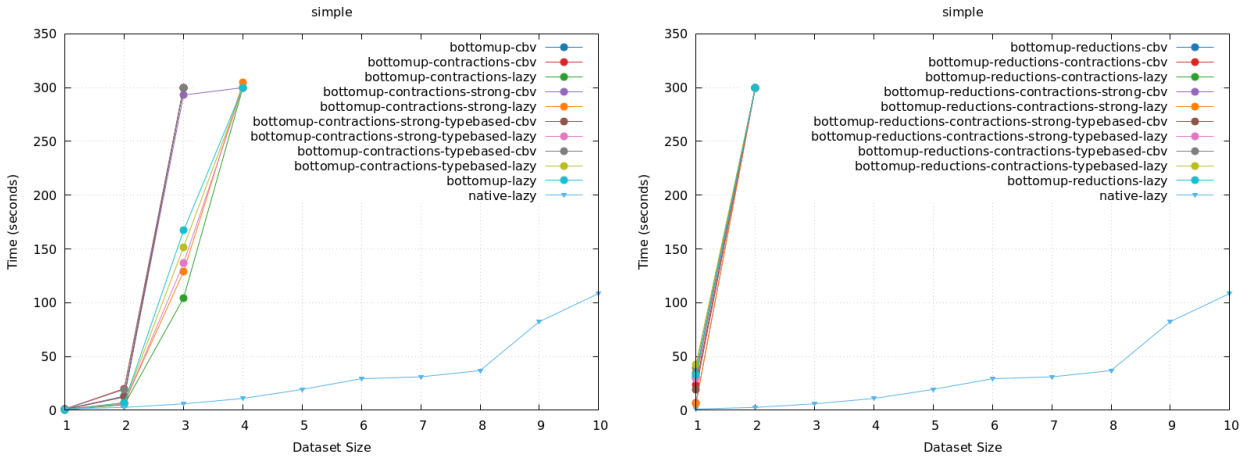
- **Simple**. i-я функция вычисляет хеш первых i символов строки, при этом код является линейным (не использует циклы или рекурсию), то есть размер кода линейно зависит от порядкового номера функции.
- **If**. Расширение набора **Simple**, в котором симулируется работа с нуль-терминированными строками. В случае, если программа дошла до нулевого символа, вычисление должно завершиться. Этот набор позволяет протестировать условный оператор и досрочный выход из функции.
- **Recursion**. Рекурсивная версия набора **Simple**. В ней функция с номером $i+1$ сначала вызывает функцию i, а затем дополняет вычисление $(i+1)$-м символом.
- **IfAndRecursion**. Рекурсивная версия набора **If**.

Для реальной части тестовых данных был выбран кошелек с мультиподписью из экосистемы TON [19]. Кошельки с мультиподписью используются в блокчейн-сетях повсеместно [20] и являются критическим элементом для безопасности многих систем, поэтому верификация таких кошельков имеет практическую значимость. Материалы для верификации этого смарт-контракта были взяты из практики компании Pruvendo, в том числе трансляция кода с исходного языка Solidity на Ursus и формальная спецификация этой системы. Эксперименты заключались в применении разных стратегий для доказательства имеющейся формальной спецификации.

Все эксперименты поставлены на процессоре Intel Xeon E5-2687W v4 @ 3.00GHz с 512 Гб оперативной памяти и 48 ядрами, использовалась версия Coq 8.16.1.

3.2. Эксперименты на линейном и рекурсивном коде

Начнём сравнение стратегий с выявления наименее производительных стратегий, которые определённо не подходят к применению на практике, поскольку работают относительно долго даже на тестах небольшое размеров. Такими стратегиями являются все стратегии, основанные на базовых стратегиях `bottomup` и `bottomup-reductions`. На рисунке 4 приведены результаты этих стратегий на наборе тестов Simple. Тогда как стратегия `native-lazy`, приведённая на этих графиках для сравнения, вычисляет систему уравнений размера 10 за минуту, вычисления для `bottomup` во всех вариациях и со всеми оптимизациями не укладываются в пять минут на тесте размера 4, а для `bottomup-reductions` — уже на тесте размера 2. На этих и дальнейших графиках вычисления ограничены пятью минутами, то есть измерения в 300 секунд означают превышение этого лимита.



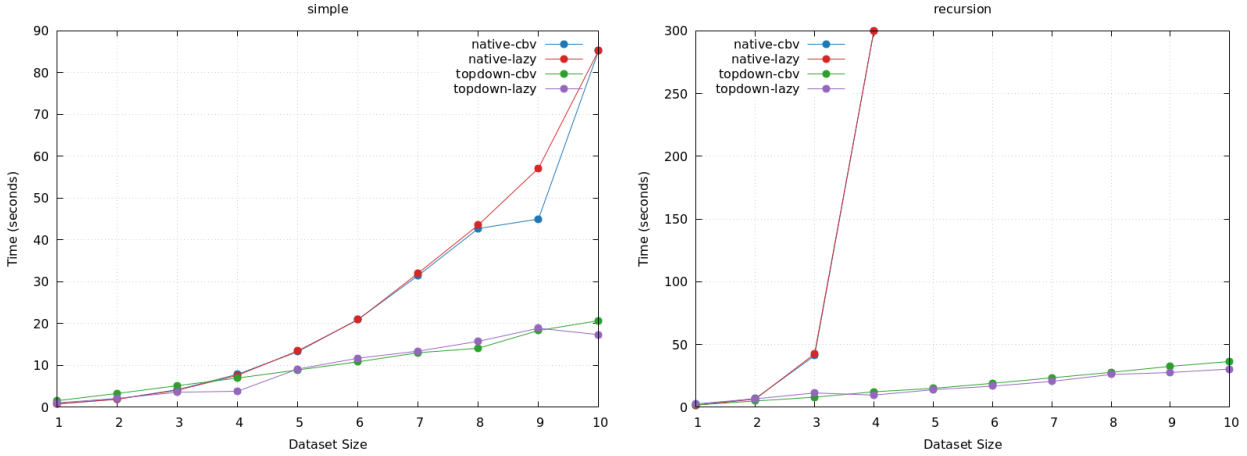
(a) Семейство стратегий `bottomup` (b) Семейство стратегий `bottomup-reductions`

Рис. 4: Результаты сравнения стратегий `bottomup` на наборе данных Simple

Такой резкий рост времени работы для этих стратегий, вероятно, является экспоненциальным, что несложно объяснить. Действительно, стратегия `bottomup` подстав-

ляет все термы не редуцируя их, что приводит к значительной дубликации кода, и даже мемоизация в ленивом вычислении не решает проблему, поскольку ещё до запуска внутренней редукции процедура подстановки термов с дублицированным кодом становится экспоненциальной. То, что стратегия `bottomup-reductions`, включающей в себя редукции на каждом шаге, является ещё медленнее наивной версии, может показаться контринтуитивным. Однако, этот факт объясняется тем, что при преждевременной редукции термов в стратегии `bottomup-reductions`, то есть при редукции термов, которые содержат неподставленные свободные переменные y_i, терм может значительно увеличиваться в размерах из-за "мёртвого кода", например если такая переменная стоит в голове условного оператора или сопоставления с образцом.

В более общем смысле, рисунок 4 иллюстрирует, насколько важно выбрать правильный порядок вычислений и подстановок, так как в результате выбора неверного порядка легко получить стратегию, не применимую на практике из-за крайней неэффективности.



(a) Набор данных Simple

(b) Набор данных Recursion

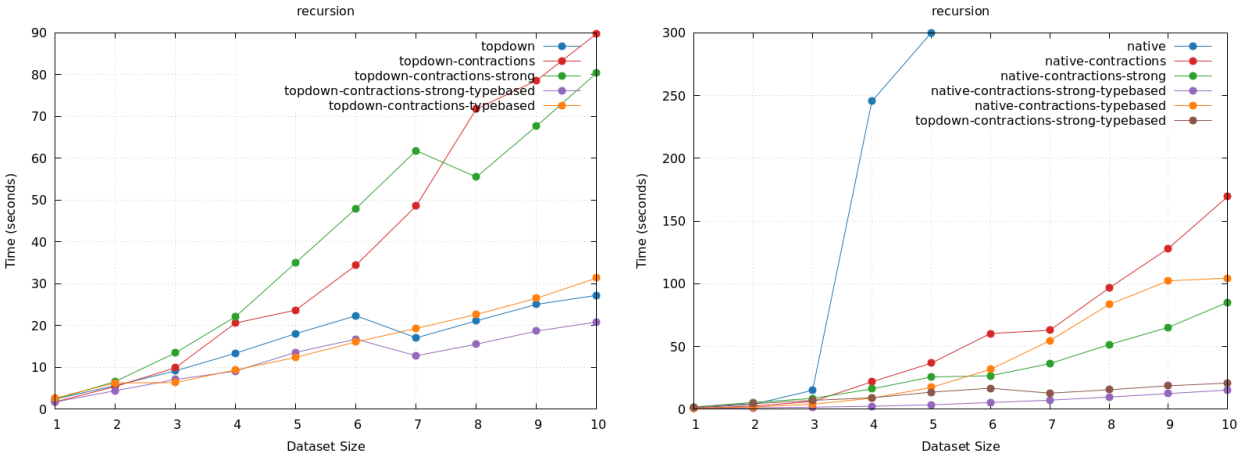
Рис. 5: Результаты сравнения базовых стратегий

Перейдём к сравнению двух оставшихся базовых стратегий, `native` и `topdown`. Результаты сравнения этих стратегий на наборах данных Simple и Recursion приведены на рисунке 5. Оказывается, что уже базовая стратегия `topdown` показывает результаты, значительно лучше, чем `native`. На наборе данных Recursion стратегия `native` показывает экспоненциальный взрыв, похожий на наблюдаемый со стратегиями `bottomup`, и вероятно объясняемый неэффективностями в реализации редукций. Однако, стратегия `topdown` даже без дальнейших эвристических оптимизаций, использующая множество редукций для решения небольших подзадач, не демонстрирует экспоненциального поведения, и даже выглядит линейно на рассматриваемых объёмах данных. Таким образом, уже правильный выбор базовой стратегии, использующую естественную декомпозицию данных на уравнения и обрабатывающих их в

необходимом порядке, позволяет значительно улучшить производительность вычислений.

Другим наблюдением, отражённым на рисунке 5, является статистическая неразличимость на рассматриваемых программах при использовании внутренних редукций `cbv` или `lazy`. Из-за этой неразличимости в приведённых далее графиках и рассуждениях вариации стратегий, отличающиеся только выбором внутренней редукции, будут опущены, и для всех стратегий будет использоваться вариант с редукцией `lazy`. Однако, было установлено, что на некоторых программах выбор редукции всё же имеет существенное влияние, это будет описано в разделе 3.4.

Итак, стратегия `topdown` уже показывает значительное, асимптотическое улучшение относительно стратегии `native`, что является центральным практическим результатом работы. Однако, можно получить ещё более эффективные стратегии, рассмотрев дальнейшие эвристики, описанные в разделах 2.3 и 2.4. Результаты сравнения этих стратегий приведены на рисунке 6.



(a) Семейство стратегий `topdown`

(b) Семейство стратегий `native`

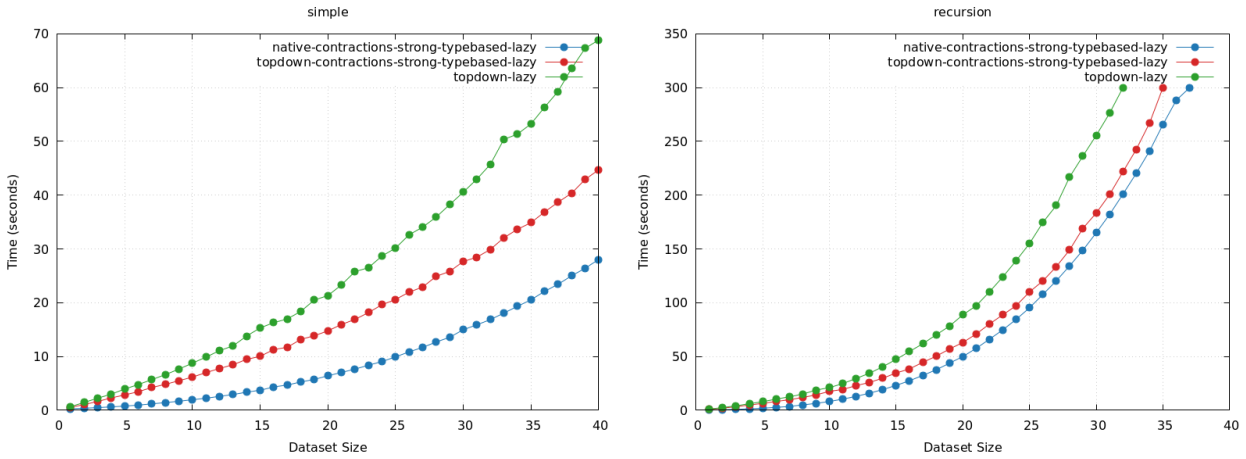
Рис. 6: Результаты сравнения стратегий с эвристическими оптимизациями на наборе данных Recursion

Рассматривая результаты различных эвристических оптимизаций стратегии `topdown` (рис. 6a), можно заметить, что почти все из них оказывают негативное влияние на производительность. Так, хуже всего результат показывают оптимизации, основанные на чистых графовых свойствах (`topdown-contractions` и `topdown-contractions-strong`), из-за накладных расходов на явное построение графа они проигрывают базовой стратегии. Не показывает значительного улучшения и слабая версия оптимизации, основанной на типах данных `topdown-contractions-typebased`. Однако сильная версия этой оптимизации `topdown-contractions-strong-typebased` немного быстрее, чем базовая версия `topdown`.

Что же касается тех же оптимизаций, применённых для стратегии `native` (рис. 6b), то они все улучшают эффективность базовой стратегии и не показывают тако-

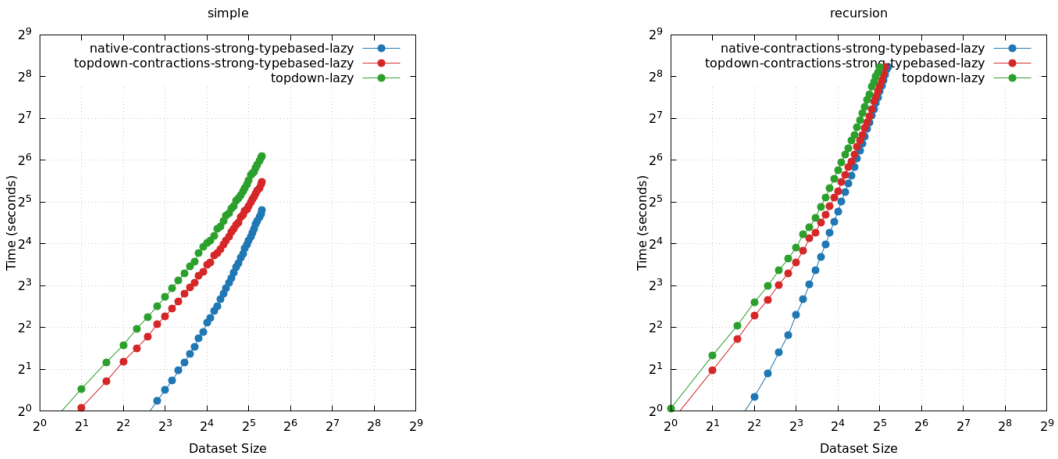
го резкого роста. Однако, их относительный порядок совпадает с изложенным для `topdown`, а именно графовые эвристики и слабая версия типовой эвристики показывают не такой эффективный результат, как сильная версия типовой эвристики `native-contractions-strong-typebased`. Интересно, что несмотря на то, что базовая версия `native` значительно медленнее базовой версии `topdown`, если применить к обеим стратегиям оптимизацию `contractions-strong-typebased`, то `native` станет даже немного, но статистически заметно, эффективнее `topdown`.

Итак, выявлены три стратегии, показывающие наилучшие результаты: стратегия `topdown` в базовой версии и стратегии `topdown` и `native` с оптимизацией `contractions-strong-typebased`. На рисунке 7 приведены результаты этих трёх стратегий на больших объёмах данных (до n = 40, тогда как предыдущие графики включали данные до n = 10).



(a) Набор данных Simple (b) Набор данных Recursion

Рис. 7: Результаты сравнения лучших стратегий на больших данных



(a) Набор данных Simple (b) Набор данных Recursion

Рис. 8: Результаты сравнения лучших стратегий на больших данных в log-log шкале

На наборе данных Simple (рис. 7a) рост времени работы стратегий выглядит почти линейным, кривизна графика не очень заметна. Однако, на наборе данных Recursion (рис. 7b), который является в несколько раз более вычислительно затратным по размеру системы уравнений, чем набор Simple, отчётливо видно, что рост времени работы всех трёх стратегий имеет нелинейный характер. На рисунке 8 приведены версии тех же графиков с рисунка 7, но использована log-log шкала. Видно, что рост времени работы всех стратегий является полиномиальным. При этом обе версии стратегии `topdown` на наборе Simple завершаются за $\mathcal{O}(n^{1.5})$, а на наборе Recursion за $\mathcal{O}(n^2)$. Стратегия `native-contractions-strong-typebased` на наборе Simple завершается за $\mathcal{O}(n^2)$, а на наборе Recursion за $\mathcal{O}(n^{2.5})$. Интересно, что стратегия, показывающая на рассматриваемых данных лучшие результаты, имеет менее эффективную асимптотику. Из этого следует, что при достаточно больших размерах данных стратегия `topdown-contractions-strong-typebased`, имеющая асимптотику лучше, будет показывать самые эффективные результаты. Однако, похоже, что это произойдёт лишь при данных крайне большого размера, представляющим меньший практический интерес.

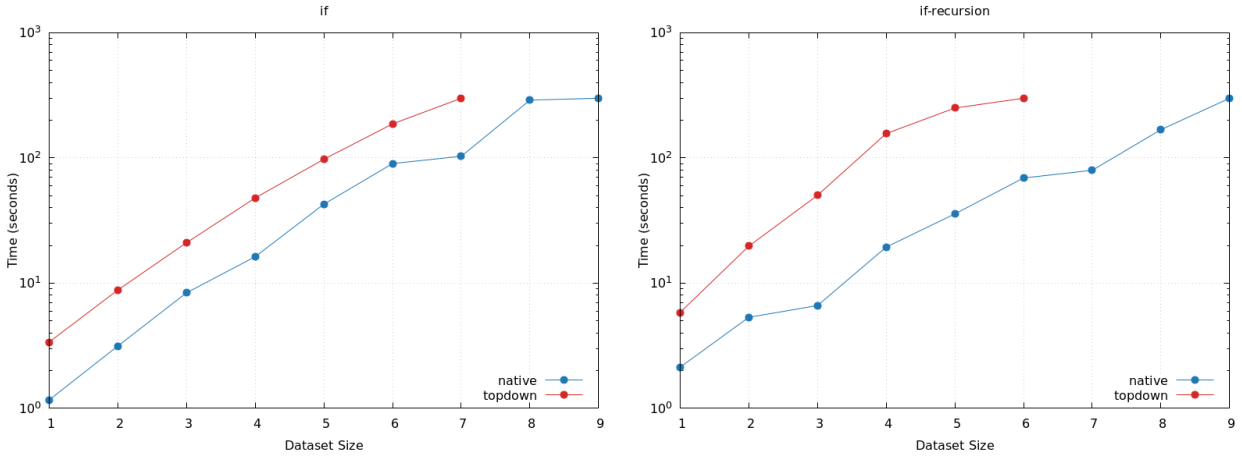
Несмотря на то, что было бы предпочтительнее получить линейный алгоритм, коэффициенты полиномиальных функций, описывающих время работы лучших стратегий, достаточно малы и приемлемы на практике. Так, за пять минут лучшая стратегия может обработать код из нескольких сотен строк кода с глубиной рекурсии до 35, что соответствует реалиям разработки смарт-контрактов. Отметим, что стратегия `native` без оптимизаций за пять минут может обработать только код из нескольких десятков строк кода с глубиной рекурсии до 4, то есть существенно меньше, и не может обрабатывать за адекватное время большую часть реальных смарт-контрактов.

На всех рассматриваемых наборах данных эффективнее всего проявляет себя стратегия `native-contractions-strong-typebased`, хотя её преимущество относительно стратегии `topdown-contractions-strong-typebased` уменьшается с увеличением размеров теста, а на крайне больших тестах вторая стратегия превзойдёт первую. Что же касается преимущества относительно базовой стратегии `topdown`, то есть преимущества стратегии с эвристическими оптимизациями перед лучшей базовой стратегией, оно может показаться незначительным на больших тестах, поскольку асимптотического выигрыша эвристические оптимизации не привносят и на больших тестах разница между ними не кажется существенной. Однако, на вполне реалистичных размерах функции в 70 строк (n = 35 в наборе данных Simple) и глубине рекурсии в 10, эвристические оптимизации позволяют ускорить вычисление в 3 раза. Такие оптимизации могут быть особенно значимы в случаях, когда символьное вычисление используется в композитных процессах, например при доказательстве свойств сложных сценариев, где этот множитель 3 будет возведён в степень. Таким образом, несмотря на то, что уже базовая стратегия `topdown` показывает хороший результат, результат дальней-

ших оптимизаций также значим на практике.

3.3. Эксперименты на коде с условными операторами

Перейдём к экспериментам с использованием программ, содержащих условные операторы. Поскольку, как описано в разделе 2.5, работа с условными операторами в общем случае является экспоненциально сложной относительно числа условных операторов, все графики в этом разделе будут построены на логарифмической шкале. Также из рассмотрения в этом разделе будут исключены стратегии из семейства `bottomup`, показывающие экспоненциальный рост уже на линейном коде.



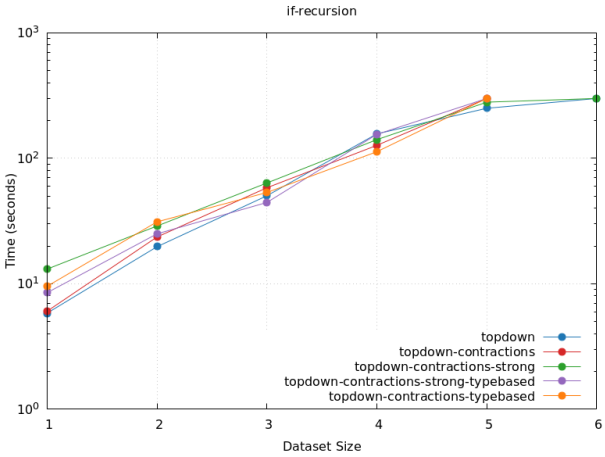
(a) Набор данных If

(b) Набор данных IfAndRecursion

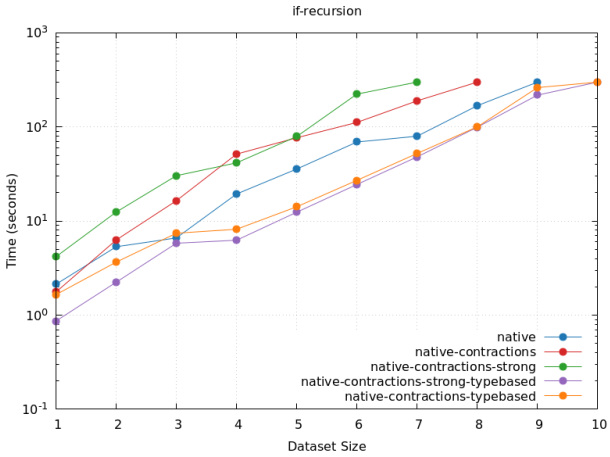
Рис. 9: Результаты сравнения базовых стратегий на программах с условными операторами

На рисунке 9 представлены результаты сравнения базовых стратегий `topdown` и `native`. Интересно, что на коде, содержащем большое число условных операторов, в отличие от линейных программ, стратегия `native` показывает более эффективный результат. Это объясняется описанными в разделе 2.5 особенностями работы с условными операторами. Так, при использовании стратегии `topdown` для сохранения инварианта необходимо вызывать `destruct`, как только в порядке обработки встречается уравнение с условным оператором, что вызывает экспоненциальное дублирование подстановок в дальнейших уравнениях.

На рисунке 10a видно, что никакие эвристики не оказывают значимого влияния на эффективность стратегии `topdown`. Похоже, описанная выше неэффективность в экспоненциальной дубликации подстановок нивелирует потенциальный прирост производительности от эвристических оптимизаций. В то же время, для эвристик, применённых к стратегии `native` (рисунок 10b), наблюдаются те же закономерности, описанные в предыдущем разделе: эвристики, основанные на графовых свойствах, не демонстрируют улучшения или замедляют работу, а эвристики, основанные на типах



(a) Семейство стратегий `topdown`



(b) Семейство стратегий `native`

Рис. 10: Результаты сравнения стратегий с эвристическими оптимизациями на наборе данных IfAndRecursion

данных, улучшают производительность.

В результате можно установить, что, несмотря на то что на программах с условными операторами некоторые менее эффективные на линейном коде стратегии могут оказаться более эффективными из-за особенностей работы с условными операторами, тем не менее самой эффективной стратегией остаётся `native-contractions-strong-typebased`.

3.4. Выбор внутренней редукции

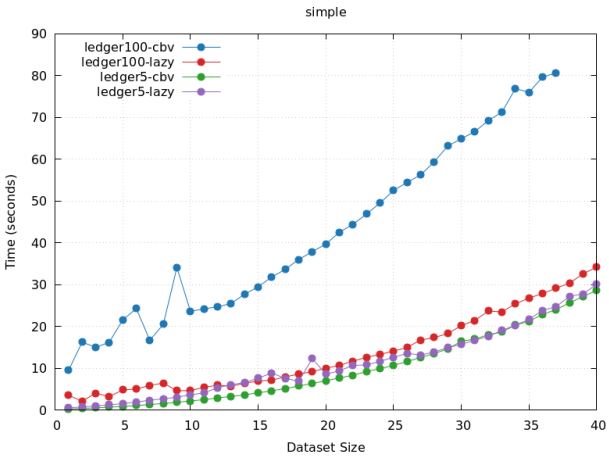


Рис. 11: Результаты сравнения внутренних редукций на разном количестве полей

В приведённых ранее графиках и рассуждениях варианты стратегий с внутренней редукцией `cbv` опускались, так как они не показывали на рассматриваемых программах значительного отличия от вариантов с `lazy`. Однако, оказалось, что для некоторых программ `lazy` показывает значительно более эффективный результат.

Напомним, что применение `lazy` особенно эффективно при редукциях термов, содержащих большое количество "мёртвого кода". Поэтому был проведён эксперимент по запуску лучшей стратегии `contractions-strong-typebased` на двух версиях набора данных Simple. В первом состояние контракта включало только 5 используемых полей, во втором после

добавления 95 неиспользуемых полей общее число полей равняется 100.

На рисунке 11 представлены результаты такого эксперимента. Видно, что при малом числе полей, как и наблюдалось в предыдущих экспериментах, выбор внутренней редукции не оказывает значимого влияния на эффективность вычисления. Однако, при увеличении числа полей от 5 до 100 стратегия, основанная на редукции `cbv`, замедляется в несколько раз (около трёх), тогда как при выборе редукции `lazy` замедление хоть и заметно, но не является настолько существенным.

Таким образом, при наличии в вычислении "мёртвого кода" существенного размера в виде неиспользуемых в рассматриваемой функции полей программы, выбор редукции `lazy` позволяет в несколько раз ускорить алгоритм. Поскольку в рамках исследования не было найдено случаев, в которых редукция `cbv` показывает заметное ускорение, можно сделать вывод о том, что редукция `lazy` лучше подходит для рассматриваемой задачи.

3.5. Эксперименты на реальных данных

В синтетических данных, эксперименты на которых описаны выше, изучалось влияние ключевых параметров на эффективность стратегий, будь то размер кода, глубина рекурсии или число условных операторов. В этом разделе будут рассмотрены результаты сравнения стратегий на коде реального смарт-контракта кошелька с мультиподписью.

Для этого был взята спецификация и доказательства трёх ключевых функций этого контракта, `sendTransaction`, `submitTransaction` и `confirmTransaction`. Ключевые характеристики этих функций приводятся в таблице 1.

Функция	Строк	Вызовов функций	Глубина рекурсии	Условных операторов
send	4	2	1	2
submit	45	15	2	7
confirm	22	7	2	4

Таблица 1: Ключевые характеристики рассматриваемых функций

Внутри доказательства спецификации каждой из функций несколько раз вызывается процедура упрощения системы уравнений с помощью одной из стратегий редукции, для каждого из доказываемых аспектов поведения функций или различных предположений на аргументы функции и исходное состояние системы. Затем, после того как все эти подцели доказаны, вызывается команда QED, производящая финальную проверку типов построенного доказательства.

В таблице 2 приведены результаты таких экспериментов. Все используемые стратегии включают в себя `lazy` как внутреннюю тактику, согласно соображениям, высказанным в разделе 3.4. Для экспериментов на этом этапе по результатам экспе-

риментов на синтетических данных были отобраны четыре стратегии: `native` (NL), `native-contractions-strong-typebased` (NCSTL), `topdown` (TL) и `topdown-contractions-strong-typebased` (TCSTL).

	NL	**NCSTL**	**TL**	**TCSTL**
1	3.42	2.87	**2.30**	5.69
QED	2.5	**2.39**	3.8	4.53
Σ	5.92	**5.26**	6.1	10.22

(a) Функция sendTransaction

	NL	**NCSTL**	**TL**	**TCSTL**
1	0.90	**0.90**	1.46	1.24
2	0.95	**0.84**	1.05	1.04
3	0.99	**0.94**	1.44	1.43
4	1.86	**1.51**	3.58	2.93
5	1.44	**1.22**	12.26	7.99
6	23.40	**7.46**	46.16	58.54
7	23.63	**7.75**	41.19	58.16
QED	**11.51**	12.63	78.09	87.26
Σ	64.68	**33.25**	185.23	218.59

(b) Функция submitTransaction

	NL	**NCSTL**	**TL**	**TCSTL**
1	0.69	0.65	**0.53**	0.49
2	0.78	**0.70**	1.03	0.70
3	0.10	**0.09**	0.11	0.10
4	1.73	**1.44**	1.62	2.08
5	1.79	**1.25**	3.03	2.20
6	**0.20**	0.22	**0.20**	**0.20**
7	2.08	**1.35**	3.47	2.53
8	3.64	**2.03**	5.68	4.65
9	16.80	12.33	12.86	**11.79**
QED	**10.25**	10.34	31.74	26.73
Σ	38.06	**30.4**	60.27	51.47

(c) Функция confirmTransaction

Таблица 2: Результаты экспериментов на реальных данных

Стратегии класса `topdown` показывают неэффективные результаты. Вероятно, это связано с теми же причинами, по которым они работали относительно медленно в экспериментах в разделе 3.3. Во всех рассматриваемых функциях присутствуют условные операторы в достаточном количестве, а с ними эти стратегии работают менее эффективно. Такая неэффективная работа этих стратегий на реальных данных является дополнительным аргументом против их использования на практике, несмотря на то, что, как показано в разделе 3.2, стратегии из класса `topdown` на некоторых примерах могут показывать лучшую асимптотику.

Что же касается двух оставшихся стратегий, базовой стратегии `native` и самой эффективной стратегии `native-contractions-strong-typebased`, вторая не демонстрирует такого серьёзного улучшения относительно первой, как наблюдалось в некоторых тестах из раздела 3.2. По-видимому, при таких характеристиках кода, как небольшая глубина рекурсии и большое число условных операторов, эвристики не дают такого серьёзного улучшения. Однако, это улучшение всё ещё значительно: так, на запусках 6 и 7 функции `submitTransaction` эвристика улучшает результат стратегии `native` в три раза, а на запуске 9 функции `confirmTransaction` — в полтора.

Более того, не нашлось ни одного запуска, в котором какая-то стратегия была бы заметно эффективнее `native-contractions-strong-typebased`. Эксперименты из предыдущих разделов также не находили таких случаев (за исключением, возможно, крайне больших и менее интересных на практике размеров кода). Ни про какие другие стратегии этого сказать нельзя. Таким образом, несмотря на то, что для каких-то программ выигрыш от использования этой стратегии может быть менее заметен, нет необходимости выбирать стратегию исходя из свойств рассматриваемой программы, ведь единственная стратегия на всех экспериментах показывает результаты не хуже остальных.

3.6. Выводы и результаты по главе

Создан тестовый набор из синтетической части (четыре программы на языке Ursus линейно растущего размера, покрывающие фундаментальные конструкции языков программирования) и реальной части (смарт-контракт из практики компании Pruvendo). По результатам экспериментов с запуском стратегий на тестах можно сделать следующие выводы:

- все стратегии класса `bottomup` экспоненциальны и неприменимы на практике;
- базовая стратегия `topdown` на линейном коде уже значительно эффективнее, чем `native`;
- графовые эвристики и слабая версия эвристики на типах данных оказывает отрицательное влияние или незначительное положительное влияние на эффективность стратегий;
- лучшие результаты на линейном коде показывает стратегия `native` с применённой эвристикой `contractions-strong-typebased`, при этом стратегия `topdown` в базовой версии или с применённой эвристикой имеют лучшую асимптотику, однако будут более эффективными лишь на крайне больших данных;
- лучший результат на коде с условными операторами показывает стратегия `native`

с применённой эвристикой `contractions-strong-typebased`, тогда как все стратегии из класса `topdown` на таком коде хуже даже базовой стратегии `native`;

- стратегия `native-contractions-strong-typebased-lazy` является особо эффективной в программах с большой глубиной рекурсии и малым числом условных операторов, однако на всех исследуемых программах эта стратегия не показывала результат заметно хуже любой другой стратегии;
- итого, лучшей стратегией по всем экспериментам является `native-contractions-strong-typebased-lazy`.

Заключение

В рамках данной работы были достигнуты следующие результаты:

- Разработаны стратегии вычисления, всего 40 различных вариаций. В том числе, представлена методология разработки эвристических оптимизаций, основанных на структурных свойствах данных.
- Для сравнения стратегий создан набор программ на языке Ursus, включающий фундаментальные конструкции языков программирования.
- Все приведенные в тексте работе стратегии реализованы на языке Ltac. Эта реализация и тестовые наборы доступны в репозитории `https://github.com/trilis/ursus-solver-strategies`
- По результатам сравнения стратегий, лучшая из них на некоторых программах (с большой глубиной рекурсии и малым числом условных операторов) демонстрируют асимптотическое улучшение в производительности относительно базовой, а именно полиномиальный рост времени работы относительно экспоненциального.
- Несмотря на то, что на некоторых программах улучшение производительности менее выражено, самая эффективная стратегия на всех рассмотренных программах не показывает результат значимо хуже любой другой стратегии. Это позволяет рекомендовать её использование во всех случаях, без предварительного анализа характеристик программы.
- Наиболее эффективная стратегия будет использоваться в дальнейших коммерческих проектах по формальной верификации, поскольку оптимизации имеют практическое значение для развития системы Ursus.

Список литературы

[1] The Coq Development Team. — The Coq Proof Assistant, 2022. — Version 8.16. URL: https://doi.org/10.5281/zenodo.7313584.

[2] Gonthier G. Formal proof - The four-color theorem // Notices of the AMS. — 2008. — 01. — Vol. 55. — P. 1382–1393.

[3] Leroy Xavier. Formal Verification of a Realistic Compiler // Communications of the ACM. — 2009. — 07. — Vol. 52.

[4] Hawblitzel Chris, Howell Jon, Kapritsos Manos et al. IronFleet: proving practical distributed systems correct // Proceedings of the 25th Symposium on Operating Systems Principles. — SOSP '15. — New York, NY, USA : Association for Computing Machinery, 2015. — P. 1–17. — URL: https://doi.org/10.1145/2815400.2815428.

[5] Gross Jason, Erbsen Andres. 10 Years of Superlinear Slowness in Coq // Coq Workshop. — 2022.

[6] Gross Jason. Performance Engineering of Proof-Based Software Systems at Scale : PhD thesis / Jason Gross ; Massachusetts Institute of Technology. — 2021.

[7] Almakhour Mouhamad, Sliman Layth, Samhat Abed Ellatif, Mellouk Abdelhamid. Verification of smart contracts: A survey // Pervasive and Mobile Computing. — 2020. — Vol. 67. — P. 101227. — URL: https://www.sciencedirect.com/science/article/pii/S1574119220300821.

[8] Pruvendo. Ursus language documentation. — 2025. — URL: https://ursus-lang.dev/ (online; accessed: 2025-04-04).

[9] Barendregt Hendrik Pieter. The lambda calculus - its syntax and semantics. — North-Holland, 1985. — Vol. 103 of Studies in logic and the foundations of mathematics. — ISBN: 978-0-444-86748-3.

[10] Peyton Jones Simon L. The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). — USA : Prentice-Hall, Inc., 1987. — ISBN: 013453333X.

[11] Barendregt Henk (Hendrik). Lambda Calculi with Types. — 1992. — 01. — Vol. 2. — P. 117–309. — ISBN: 0198537611.

[12] Coquand Thierry, Huet Gérard. Constructions: A higher order proof system for mechanizing mathematics // EUROCAL '85 / Ed. by Bruno Buchberger. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1985. — P. 151–184.

[13] Pottier François. Qed time is quadratic (sharing of sub-terms is not detected) - Issue #18520. — 2024. — URL: https://github.com/rocq-prover/rocq/issues/18520 (online; accessed: 2025-04-04).

[14] Léchenet Jean-Christophe. Nested let-in cause high memory consumption - Issue #10206. — 2019. — URL: https://github.com/rocq-prover/rocq/issues/10206 (online; accessed: 2025-04-04).

[15] Grégoire Benjamin, Leroy Xavier. A compiled implementation of strong reduction // Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. — ICFP '02. — New York, NY, USA : Association for Computing Machinery, 2002. — P. 235–246. — URL: https://doi.org/10.1145/581478.581501.

[16] Boespflug Mathieu, Dénès Maxime, Grégoire Benjamin. Full Reduction at Full Throttle // Certified Programs and Proofs / Ed. by Jean-Pierre Jouannaud, Zhong Shao. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. — P. 362–377.

[17] Leivent Jonathan. vm_compute and native_compute defy Opaque - Issue #4476. — 2016. — URL: https://github.com/rocq-prover/rocq/issues/4476 (online; accessed: 2025-04-04).

[18] Furia Carlo A., Meyer Bertrand, Velder Sergey. Loop invariants: Analysis, classification, and examples // ACM Comput. Surv. — 2014. — Vol. 46, no. 3. — 51 p. — URL: https://doi.org/10.1145/2506375.

[19] TON Labs. Safe Multisig Wallet. — 2025. — URL: https://github.com/everx-labs/ton-labs-contracts/blob/master/solidity/safemultisig/SafeMultisigWallet.sol (online; accessed: 2025-04-04).

[20] Houy Sabine, Schmid Philipp, Bartel Alexandre. Security Aspects of Cryptocurrency Wallets—A Systematic Literature Review // ACM Computing Surveys. — 2023. — Vol. 56. — P. 1 – 31. — URL: https://api.semanticscholar.org/CorpusID:258718193.