СОДЕРЖАНИЕ

1 ТЕРМИНОЛОГИЯ

– SMT-решатель

– условия корректности

– неинтерпретируемые функции

2 ВВЕДЕНИЕ

Современные информационные системы играют ключевую роль в различных сферах человеческой деятельности - от финансов и здравоохранения до транспорта и коммуникаций. С ростом сложности и масштабов таких систем возрастает и риск возникновения ошибок, сбоев и уязвимостей, которые могут привести к серьезным последствиям: финансовым потерям, утечкам данных, нарушению безопасности и даже угрозам жизни и здоровью.

Одной из основных проблем является обеспечение надежности программного обеспечения, которое лежит в основе информационных систем. Надежность включает в себя корректность работы, устойчивость к ошибкам, безопасность и предсказуемость поведения. Однако традиционные методы тестирования и отладки не всегда способны гарантировать отсутствие ошибок, особенно в сложных распределённых и параллельных системах. Даже при использовании современных средств автоматизированного тестирования остаётся вероятность пропуска критических дефектов, которые могут проявиться только в редких или трудно воспроизводимых сценариях. Это особенно актуально для систем, работающих в реальном времени, а также для приложений, связанных с обработкой чувствительных данных или управлением физическими объектами.

В связи с этим возникает необходимость в более строгих методах верификации и проверки программ, которые позволяют выявлять ошибки на ранних этапах разработки, снижая тем самым риски и затраты на исправление дефектов. Одним из таких методов является использование формальных систем типов, которые обеспечивают статическую проверку свойств программ ещё

на этапе компиляции. Формальные методы позволяют не только обнаруживать ошибки, но и доказывать корректность программ относительно заданных спецификаций, что особенно важно для критически важных приложений.

Функциональные языки программирования и их системы типов играют важную роль в повышении надёжности программного обеспечения. Благодаря чистоте функций, отсутствию побочных эффектов и строгой типизации, функциональные языки позволяют создавать более предсказуемый и легко анализируемый код. Чистота функций означает, что результат их выполнения зависит только от входных данных и не изменяет состояние программы, что существенно упрощает анализ и тестирование.

Системы типов в функциональных языках обеспечивают раннее обнаружение многих классов ошибок, таких как несоответствие типов, неправильное использование данных и нарушение контрактов функций. Это значительно снижает вероятность возникновения ошибок во время выполнения программы, поскольку некорректные операции блокируются ещё на этапе компиляции. Кроме того, современные функциональные языки часто поддерживают расширенные системы типов, включая алгебраические типы данных, полиморфизм и типы высших порядков, что позволяет выразить сложные свойства программ непосредственно в типах. Это способствует более строгой спецификации и проверке программных компонентов, делая код более надёжным и понятным для дальнейшей поддержки и развития.

Примером может служить язык Haskell, где система типов позволяет кодировать инварианты и ограничения, обеспечивая безопасность и корректность программ. Например, с помощью алгебраических типов данных можно явно различать успешные и ошибочные результаты вычислений, что предотвращает

многие распространённые ошибки, связанные с обработкой исключительных ситуаций. Такие возможности делают функциональные языки привлекательными для разработки критически важных систем, где надёжность является приоритетом.

Однако даже самые продвинутые системы типов имеют свои ограничения. Традиционные типы позволяют гарантировать лишь базовые свойства данных, такие как принадлежность к определённому множеству (например, целые числа или строки), но не способны выразить более сложные ограничения, например, что число должно быть положительным, строка - непустой, а список - отсортированным. В результате часть инвариантов приходится проверять вручную, что увеличивает вероятность ошибок. Одним из вариантов решения этой проблемы могут служить уточняющие типы.

Уточняющие типы (refinement types) представляют собой расширение традиционных систем типов, позволяющее дополнить базовые типы логическими предикатами, которые ограничивают множество значений, описываемых типом. Это обеспечивает более точную статическую проверку программ и позволяет выявлять ошибки, которые не могут быть обнаружены обычными системами типов. Использование уточняющих типов позволяет формализовать и проверять сложные свойства данных и функций, такие как диапазоны значений, инварианты структур данных и пред- и постусловия функций. Это значительно повышает надёжность программного обеспечения, снижая вероятность ошибок времени выполнения.

Например в Листинг 1 показано как, в языке Haskell с использование фреймворка liquid-haskell можно в аннотации определить возвращаемый тип с предикатом того, что возвращаемое число натуральное и больше или равно

исходному. Также можно заметить, что на чистом Haskell сигнатура `Int -> Int`, а это не позволяет гарантировать нужные свойства в момент проверки типов.

```
1 {-@ abs :: x:Int → {v:Int | 0 ⩽ v && v ⩾ x} @-}
2 abs :: Int → Int
3 abs x = if x < 0 then 0 - x else x
```

Листинг 1 — Пример использования уточняющих типов

Такая модель гарантирует на этапе компиляции, что возвращаемое значение abs натуральное. Это предотвращает множество ошибок, связанных с некорректными данными, и позволяет сосредоточиться на бизнес-логике, не тратя ресурсы на рутинные проверки.

Тем не менее интеграция уточняющих типов в язык программирования особенно усложняется необходимостью тесного взаимодействия с SMT-решателем, который выступает ядром автоматической проверки корректности предикатов, задающих уточняющие типы. Такой подход позволяет существенно снизить нагрузку на программиста: вместо ручного конструирования доказательств корректности достаточно формализовать требуемые свойства в виде логических предикатов, а SMT-решатель берет на себя задачу проверки их истинности. Однако именно эта интеграция порождает целый ряд новых проблем, требующих особого внимания при проектировании и реализации фреймворка.

Во-первых, необходимо четко определить грамматику предикатов, которые будут поддерживаться системой уточняющих типов. С одной стороны, грамматика должна быть достаточно выразительной, чтобы позволять задавать интересные и практически значимые свойства (например, линейные неравенства, свойства списков, инварианты структур данных). С другой стороны, она обязана оставаться в области, разрешимой для SMT-решателя, иначе

задача проверки станет неразрешимой или слишком затратной по времени. На практике это приводит к необходимости ограничивать язык предикатов, например, только к кванторно-свободной линейной арифметике или арифметике с uninterpreted functions.

Во-вторых, требуется реализовать корректный и эффективный механизм трансляции предикатов, написанных на языке программирования, в формат, понятный SMT-решателю (например, SMT-LIB). Это предполагает не только синтаксическую трансляцию, но и сохранение семантики, в том числе обработку переменных, функций, областей видимости и других особенностей исходного языка. Любая ошибка на этом этапе может привести к ложным срабатываниям или, напротив, к пропуску ошибок в программе.

Третья проблема - автоматическая проверка условий корректности , возникающих при использовании уточняющих типов. Для каждого использования уточняющего типа система должна сформулировать логическую формулу, выражающую требуемое свойство, и передать ее SMT-решателю для проверки. Если формула невалидна, решатель может предоставить контрпример, что значительно облегчает диагностику ошибок, но требует от системы поддержки обратной связи и интерпретации результатов SMT-решателя для пользователя.

Таким образом, успешное внедрение уточняющих типов требует решения целого комплекса задач: от формализации поддерживаемой логики и построения транслятора предикатов, до эффективной интеграции с SMT-решателем и организации обратной связи для пользователя. Разработка специализированного фреймворка, который стандартизирует эти процессы, предоставит средства настройки и расширения грамматики предикатов, а также автоматизирует вывод и проверку уточняющих типов, способна значительно снизить

порог вхождения для исследователей и разработчиков. Это, в свою очередь, откроет путь к более широкому распространению уточняющих типов в академических и промышленных проектах, повысив надежность и безопасность программного обеспечения.

3 ОБЗОР ЛИТЕРАТУРЫ И СУЩЕСТВУЮЩИЕ РЕШЕНИЯ

Здесь нужно описать что мы ищем в других реализациях и привести список других реализаций

4 ПРОЕКТИРОВАНИЕ

4.1 Язык предикатов

Типы уточнений дополняют стандартные системы типов логическими предикатами для определения и проверки семантических свойств программ. При разработке фреймворка с типами уточнений выбор базовой логики для этих предикатов имеет решающее значение. Поэтому выбор логики для выражения этих предикатов имеет решающее значение, поскольку он определяет как выразительность системы типов (т.е. диапазон свойств, которые могут быть заданы), так и разрешимость проверки типов (т.е. возможность автоматизации проверки). Слишком выразительная логика чревата неразрешимостью, что делает проверку типов непрактичной, в то время как чрезмерно строгая логика ограничивает полезность уточняющих типов для реальных задач проверки.

4.1.1 Основные свойства логики для уточняющих типов

1) **Разрешимость**: Выполнимость/валидность формул должна поддаваться алгоритмической проверке.
2) **Выразительность**: Логика должна поддерживать предикаты, относящиеся к практическим свойствам программы (например, арифметические неравенства и инварианты структуры данных).
3) **Эффективность**: Скорость автоматизированной проверки с помощью SMT-решателей должно быть приемлемым для пользователей фреймворка.
4) **Интеграция с SMT-решателями**: Поддержка в современных системами для решения SMT-задач.

4.1.2 Сравнение логик

1) Логика первого порядка без кванторов (QF_FOL)
   - **Разрешимость**: Гарантируется для ряда фрагментов, например QF_UFLIA (неинтерпретируемые функции + линейная арифметика).
   - **Выразительность**: Ограничена формулами без кванторов, но достаточна для многих уточнений (например, $x > 0$, $f(x) = y$).
   - **Эффективность**: SMT-решатели разрешают ограничения QF_FOL за миллисекунды с помощью DPLL(T) и процедур, специфичных для различных теорий
   - **Интеграция с SMT-решателями** - нативно поддерживается в большинстве решателей
2) Логика первого порядка с кванторам (FOL)
   - **Разрешимость**: Неразрешимость в целом, ограниченные фрагменты являются нишевыми и в меньшей степени поддерживаются инструментами
   - **Выразительность**: Обеспечивает поддержку предикатов с кванторами, например $\forall i.(0 \leq i \leq n) \implies \mathrm{arr}[i] > 0$.
   - **Эффективность**: Обработка квантора приводит к значительным накладным расходам, часто приводящим к тайм-аутам, что
   - **Интеграция с SMT-решателями**: частичная
3) Другие логики
   a) Логики высших порядков (Higher-Order Logic, HOL)

Логики высших порядков позволяют использовать кванторы над функциями и предикатами, что существенно расширяет выразительность. Например, в HOL можно формализовать индуктивные инварианты или свойства рекурсивных структур данных. Однако разрешимость HOL фрагментов крайне ограничена: проверка условий корректности требует интерактивных доказательств, что противоречит требованию автоматизации через SMT-решатели. Кроме того, HOL не поддерживается большинством промышленных SMT-решателей, за исключением специализированных инструментов, что делает её непрактичной для интеграции в фреймворк уточняющих типов.

б) Модальные и временные логики (LTL, CTL)

Модальные логики, такие как линейная временная логика (LTL) или логика ветвящегося времени (CTL), предназначены для спецификации динамических свойств систем (например, «всегда» или «в будущем»). Хотя они эффективны для верификации протоколов параллелизма или реактивных систем, их семантика несовместима с задачами статической проверки инвариантов данных, которые требуют анализа статических условий корректности, а не временных траекторий.

в) Теории массивов и структур данных

Многие SMT-решатели поддерживают теории массивов (QF_AUF), позволяющие кодировать операции чтения/

записи и инварианты для структур данных. Однако их использование сопряжено с риском. Кроме того, автоматический вывод инвариантов для массивов часто требует ручного задания аксиом, что усложняет интеграцию в систему уточняющих типов, ориентированную на автоматизацию.

В итоге разрешимые без кванторов фрагменты логики первого порядка (QF_FOL) обеспечивают прочную основу для предикатов в уточняющих типах, поскольку они гарантируют баланс между выразительностью и автоматизированной, предсказуемой проверкой. Начинать с фрагмента QF_UFLIA (неинтерпретируемые функции без кванторов и линейная целочисленная арифметика) особенно выгодно, поскольку он охватывает основные теории, необходимые для большинства приложений с уточняющими типами, включая линейную арифметику для числовых инвариантов и неинтерпретированные функции для абстрактных рассуждений об операциях в программном коде. Этот фрагмент хорошо поддерживается SMT-решателями, обеспечивая масштабируемую автоматизированную проверку условий корректности без непредсказуемости или проблем с производительностью. Расширение поддерживаемых теорий возможно лишь при условии строгого контроля за их комбинацией и доказательством сохранения разрешимости, и не будет рассмотрено в этой работе

4.1.3 Синтаксис

Синтаксис языка предикатов определяется на Рис. 1 и исходит напрямую из логики первого порядка без кванторов с неинтерпретируемыми функциями и линейной арифметикой целых чисел. Единственным исключением являются

переменные Хорна, которые требуются для вывода предикатов в уточняющих типов. Они будет рассмотрены позже

$$
\begin{array}{rcll}
p & ::= & x, y, z & \textit{Переменные} \\
 & | & \textit{true, false} & \textit{Логические значения} \\
 & | & p\ OP\ p & \textit{Интерпретируемые операторы} \\
 & | & p \wedge p & \textit{Конъюнкция} \\
 & | & p \vee p & \textit{Дизъюнкция} \\
 & | & \neg p & \textit{Отрицание} \\
 & | & f(p_1, p_2, ..) & \textit{Неинтерпретируемые функции} \\
 & | & k(x, y, ..) & \textit{Переменные Хорна}
\end{array}
$$

Рисунок 1 — Синтаксис языка предикатов

Для проверки уточняющих типов в модельном языке нужно расширить грамматику самого языка таким образом чтобы язык предикатов был не мощнее представленного на Рис. 1. Предполагается, что в язык будут добавлены предикаты в аналогичном формате, что позволит сделать конвертацию предикатов модельного языка в предикаты фреймворка тривиальным преобразование одного синтаксического дерева в другое. Пример реализации такого подхода будет будет в главе с реализацией подможножества языка Ocaml на предоставленном фреймворке

4.2 Уточняющие типы

Обычно уточняющие типы представляются в формате $e :: \{v : T \mid p\}$ где

- e выражение для которого задается уточняющий тип
- v переменная связывающая значение во время исполнение
- T базовый тип выражения
- p называется уточнением типа и представляет собой ограничение множества значений выражения e. Например, уточнение в типе $e :: \{v : \text{int} \mid v > 0\}$ ограничивает множество значений e до множества положительных чисел

4.2.1 Доступ к предикатам

Одним из основных контрактов для работы с фреймворком должна быть возможность работы с типами и извлечения из них предикатов, их обработка и модификация. Это дает важную возможность вынесения в фреймворк необходимых и часто используемых функций. Базово такой контракт можно описать как функцию $\{v : T \mid p_1\} \to (p_1 \to p_2) \to \{v : T \mid p_2\}$ которая принимает тип-уточнение, функцию которая модифицирует предикат и в итоге конструируется новый тип, с новым предикатом. В дальнейшем этот контракт будет расширен и показано его применение для решения широкого ряда задач от вывода предикатов, до реализации функций усиливающих предикат

4.2.2 Сигнатура типа

Сорта (sorts) в контексте SMT — это аналоги типов данных в языках программирования. Они определяют множество значений, которые могут принимать переменные, аргументы функций или возвращаемые значения. Каждый сорт задаёт область допустимых значений и правила взаимодействия с операциями теории.

Так как фреймворк отвечает за перевод условий корректности в формат SMT-решателей, нужно предоставить некоторый формат который бы позволил из уточняющих типов получать сорта.

Для начала рассмотрим какие виды сортов бывают в SMT-LIB

- Базовые: Описываются одним идентификатором, без аргументов. Пример: Int, Bool
- Параметризованные: Описываются как идентификатор и аргументы ($< \text{sort\_symbol} >< \text{sort}_1 > ... < \text{sort}_n >$). Пример (Pair Int Bool)

– Функциональные: Представляют функции и описываются как аргументы и возвращаемое значение. Пример: $\mathrm{Int} \to \mathrm{Bool}$

Зная сорта из smt-lib, можно предоставить так называемую сигнатуру типа которая бы покрывал базовые системы типов в функциональных языках. Под сигнатурой типа подразумевается его базовый тип, с удалением всех предикатов и связываемых предикаты переменные. Например тип функции $x : \{v : \mathrm{Int} \mid v > 0\} \to y : \{v : \mathrm{Int} \mid v > 0\} \to \{v : \mathrm{Int} \mid v = x + y\}$ является зависимым, так как тип результата (его уточнение) зависит от аргументов функции, а сигнатура этого типа будет $\mathrm{Int} \to \mathrm{Int} \to \mathrm{Int}$, которая успешно кодируется в сорта SMT.

Основываясь на сортах поддерживаемых smt-lib, сигнатура типов должна поддерживать

– функции
– типы данных, в том числе параметрические
– примитивные типы данных

4.3 Неинтерпретируемые функции

Неинтерпретируемые функции - это функциональные символы в логике и SMT (теории выполнимости по модулю), которые не имеют фиксированной интерпретации или определения, кроме их названия, арности (количества аргументов) и свойства быть тотальными и детерминированными. В отличие от встроенных функций, таких как сложение или умножение, поведение которых определяется определенной теорией (например, арифметикой), неинтерпретируемые функции не имеют какого-либо внутреннего значения или связанных с ними вычислений. Единственным предполагаемым свойством является конгруэнтность: если два входа равны, их выходные данные в соответствии с

функцией также должны быть равны (т.е. если $x = y$, то $f(x) = f(y)$). Неинтерпретируемые функции полезны тем, что они позволяют рассуждать о структуре и взаимосвязях вычислений, не привязываясь к конкретной реализации или семантике. Эта абстракция особенно эффективна при верификации программ, где неинтерпретируемые функции могут моделировать неизвестное или сложное поведение, и в то же время позволяют автоматически рассуждать о равенствах и ограничениях.

Ярким примером использования неинтерпретируемых функции является кодировка длины списка в его уточнении показанная на Листинг 2

```
1 /*M len : list('a) ⇒ int */
2
3 type list('a) =
4   | Nil                       ⇒ [v| len(v) == 0]
5   | Cons (x:'a, xs:list('a)) ⇒ [v| len(v) == 1 + len(xs)]
6   ;
```

Листинг 2 — Кодировка длины списка в его типе. Пример на языке ocaml

Имея возможность кодировать типы из модельного языка в известную нам сигнатуру позволяет определять неинтерпретируемые функции в самом модельном языке, никак не требуя отдельно расширять его синтаксис. Уже затем внутри фреймворка, полученная сигнатура будет переводиться в формат SMT-LIB. Таким образом определение и использование неинтерпретируемых функций в модельном языка будет выглядит однородно и лаконично

4.4 Условия корректности

В ходе проверки уточняющих типов генерируются ограничения условий корректности, синтаксис которых представлен на Рис. 2. Ограничения состоят из предикатов без кванторов, показанных на Рис. 1, конъюнкции двух ограничений, подразумевающим, что должно выполняться оба условия, а также

импликациеи. Импликация говорит о том, что для каждого x типа b, если условие p выполняется, значит, должно выполняться и ограничение c. Тип b преобразуется в сорт SMT за счет сигнатуры типов описанной выше

SMT-решатели алгоритмически определяют, является ли ограничение c валидным, сводя его к набору формул c_i вида $\forall \vec{x}.p_i \Rightarrow q_i$, таким образом, что c валидно, если валидны c_i. Валидность каждого c_i определяется путем проверки выполнимости предиката $p_i \wedge \neg q_i$ (без кванторов): формула валидна, если не существует удовлетворяющего присваивания

$$
\begin{array}{rcll}
c & ::= & p & \textit{Предикат, Рис. 1} \\
 & | & c_1 \wedge c_2 & \textit{Конъюнкция} \\
 & | & \forall x : b.p \Rightarrow c & \textit{Импликация}
\end{array}
$$

Рисунок 2 — Синтаксис условий корректности

Имея готовые контракты для работы с типами и предикатами получается готовый API для создания ограничений во время проверки типов.

4.5 Вывод предикатов

Двунаправленная проверка типов разделят вывод и проверку типов позволяя аннотировать типы только на верхнем уровне, а типы подвыражений можно вывести на основе имеющихся аннотаций. После всех выражений уже можно генерировать условия корректности, поскольку для их генерации критично знать тип каждого выражения. Но в случае добавления в язык полиморфизма важной доработкой будет добавление вывода уточнений, что позволит избежать явных аннотаций во всех местах использования полиморфных функций и типов данных. Теперь проверка уточняющих типов будет разделена на три шага

1) Нужно расширить типы в модельном языке так, чтоб предикат мог представлять собой заглушку. И поскольку вывод базовых типов

выходит за рамки фреймворка и предоставляется для реализации разработчику языка, все типы указываемые при создании экземпляра полиморфных функций или типов данных должны в предикат этих типов указывать заглушку. Поэтому первым шагом для вывода проверки уточняющих типов должна производиться замена всех заглушек на применение переменных Хорна. В последствии на месте применения этих переменных будет выведен предикат

2) Следующим этапом идет генерация условий корректности, так же как она бы выглядела без вывода предикатов. Но здесь важной особенностью является то, что теперь полученные условия корректности содержат применение переменных Хорна, которые напрямую не поддерживаются SMT-решателями
3) В последнем этапе вместо того чтобы просить SMT-решатель определить достоверность VC, мы вызовем другой алгоритм вывода предикатов на основе ограничений и переменных Хорна. Этот решатель определяет, есть ли какие-либо уточнения, которыми могут быть заменены применения переменных Хорна, делая результирующие условия корректности верными.

4.5.1 Алгоритм

В конечном итоге для проверки уточняющих типов нам нужно в сгенерированных условиях корректности сопоставить для каждой переменной Хорна свой предикат без переменных и так чтобы получившиеся ограничения были верны, если же такого вывода нет, то результат будет Unsat.

Псевдокод функции solve представлен на Рис. 3. На вход подаются собранные во время проверки типов ограничения c и множество предикатов из которых могут быть представлены выведенные предикаты $\mathbb{Q}$. Процедура возвращает SAT если существует такое решение, которое для каждой переменной Хорна сопоставляет конъюнкцию предикатов из $\mathbb{Q}$ и удовлетворяет c

$$
\begin{aligned}
&\mathrm{solve}(\mathbb{Q}, c) = \text{if } \mathrm{SmtValid}\big(\sigma\big(\mathrm{cs}_p\big)\big) \text{ then Sat else Unsat} \\
&\quad \text{where} \\
&\quad \mathrm{cs} = \mathrm{flat}(c) \\
&\quad \mathrm{cs}_k = \big\{c \mid c \in \mathrm{cs}, c \equiv \forall \vec{x} : \vec{t}.p \Rightarrow k(y)\big\} \\
&\quad \mathrm{cs}_p = \{c \mid c \in \mathrm{cs}, c \not\equiv \forall \vec{x} : \vec{t}.p \Rightarrow k(y) \\
&\quad \sigma_0 = \lambda k. \wedge \{q \mid q \in \mathbb{Q}\} \\
&\quad \sigma = \mathrm{fixpoint}(\mathrm{cs}_k, \omega_0)
\end{aligned}
$$

Рисунок 3 — Алгоритм вывода предикатов

Алгоритм делится на два этапа

1) Подготовка

   Ограничения c разбиваются на множество ограничений *cs*, каждое их которых вида $\forall \overrightarrow{\mathrm{x} : \mathrm{t}}.p \Rightarrow p'$, где p' это либо применение переменной Хорна, либо конкретный предикат без переменных хорна. Это множество бьется на два cs_k (с переменными) и cs_p (без) как показано на Рис. 3. После этого запускается процедура нахождения неподвижной точки для нахождения решения σ для собранных ограничений cs_k.

2) Поиск неподвижной точки

   Начальным решением является σ_0, которое каждую переменную Хорна k сопоставляет с конъюнкцией всех предикатов из $\mathbb{Q}$. Любое решение которое сопоставляет переменным Хорна конъюнкции над $\mathbb{Q}$ будет слабее σ_0. Процедура fixpoint итеративно ослабляет решение σ путем выбора ограничения c которое не удовлетворяется

текущим решением σ и удаляет предикаты из k из начала c. Итерации продолжаются пока решение не будет удовлетворять cs_k. В итоге если полученное решение удовлетворяет также и cs_p то solve возвращает успех, иначе неудачу.

У предоставленного алгоритма есть важна особенность в том, что нужно предоставить ему множество предикатов которые могут быть использованы в виде конъюнкции. Используя все те же контракты предикатов и типов, можно получить встроенный в модельный язык синтаксис для ограничения предикатов который примерно может выглядеть как

```
1 Name(var1: Type, var2: Type) (Predicate)
```

На основе полученной информации станут известны имена переменных, а также сам предикат. После чего полученные предикаты передадутся решателю

5 РЕАЛИЗАЦИЯ

6 ИТОГИ

7 ЗАКЛЮЧЕНИЕ