СОДЕРЖАНИЕ

СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

– SMT-решатель

– условия корректности

– неинтерпретируемые функции

– область видимости

– фантомные типы

– тотальная функция

– GADTs

– арность

– BNFC

ВВЕДЕНИЕ

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

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 неинтерпритируемыми функциями.

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 автомати-

зирует 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.

1 ОБЗОР ЛИТЕРАТУРЫ И СУЩЕСТВУЮЩИЕ РЕШЕНИЯ

1.1 Уточняющие типы

Уточняющие типы (refinement types) - это расширение классических систем типов, позволяющее описывать не только принадлежность значения к некоторому базовому типу (например, int или bool), но и накладывать на значения дополнительные логические ограничения в виде предикатов. Формально, уточняющий тип записывается как $v : T \mid P(v)$, где T - базовый тип, а $P(v)$ - булев предикат над значением v

Стандартные системы типов позволяют гарантировать, что, например, переменная имеет тип int, но не могут выразить дополнительные свойства, такие как ограниченность диапазона. Это приводит к тому, что даже хорошо типизированные программы могут содержать ошибки времени выполнения, связанные с некорректными значениями, например:

- Деление на ноль: тип int не гарантирует, что делитель отличен от нуля.
- Выход за границы массива: тип индекса массива - int, но не гарантируется, что индекс находится в допустимом диапазоне.
- Нарушение инвариантов структур данных: например, что список отсортирован

С помощью уточняющих типов можно явно формализовать такие ограничения. Например, тип натуральных чисел может быть определён как $\text{type nat} = \{v : \text{int} \mid 0 \leq v\}$

Семантически, уточняющий тип определяет подмножество значений базового типа, для которых предикат истинный $[\![\{v : T \mid P(v)\}]\!] = \{x \in [\![T]\!] \mid P(x) = \text{true}\}$

1.2 Обзор языков с уточняющими типами

1.2.1 LiquidHaskell

Это система верификации и проверки типов для языка Haskell, которая расширяет систему типов Haskell добавлением уточняющих типов в виде аннотаций в программный код. Система предикатов поддерживает логику первого порядка без кванторов с неинтерпретируемыми функциями и линейной арифметикой. Для проверки соблюдения свойств программы используется SMT-Решатель. Применяется как плагин к компилятору GHC, что позволяет легко интегрировать систему в существующие проекты. Пример кода показан на Листинг 2

```
1 {-@ incr :: x:{Int | x ⩾ 0} → {v:Int | v = x + 1} @-}
2 incr x = x + 1
```

Листинг 2 — Пример кода на LiquidHaskell

Особенности:

- Возможность использовать в ограниченных участках кода, без необходимости вносить изменения в весь проект
- Отсутствие накладных расходов во время выполнения
- Интеграция с экосистемой языка Haskell
- Поддержка нескольких SMT-решателей
- По сравнению с другими инструментами мощность языка предикатов специально ограниченна в угоду автоматизации

1.2.2 F*

F* - функциональный язык программирования с зависимыми типами, разработанный Microsoft Research для формальной верификации программ. Основной особенностью является генерация верифицированного кода в код на языках C, OCaml, F# и WebAssembly, сохраняя доказанные свойства. F* использует зависимые типы дополняя их уточнениями, которые могут содержать логику высшего порядка. Такой выбор системы типов дает мощные инструменты для верификации программного обеспечения, но требует от программиста больших усилий для доказательства корректности, а также направления проверки типов в нужное русло. Поэтому язык используется для верификации критически важных систем, с последующей генерацией кода в один из предоставленных языков программирования. Из-за этих свойств применять язык F*, как язык общего назначения проблематично

```
1 let abs (x:int) : Tot (y:int{y ⩾ 0}) =
2   if x < 0 then -x else x
```

Листинг 3 — Пример кода на F*

1.2.3 Dafny

Dafny - язык программирования, ориентированный на разработку верифицированных программ. Dafny сочетает в себе возможности императивного, функционального и объектно-ориентированного программирования с богатой системой статической верификации. Система уточняющих типов в Dafny реализована через аннотации, включающие предусловия, постусловия и инварианты:

– Предусловия: задаются ключевым словом requires и определяют условия, которые должны выполняться перед вызовом метода.
– Постусловия: задаются ключевым словом ensures и определяют гарантии, предоставляемые методом после его выполнения.
– Инварианты циклов: задаются ключевым словом invariant и определяют свойства, которые сохраняются на каждой итерации цикла.

Dafny верифицирует программы с использованием SMT-решателя Z3. Этот процесс проверяет, что реализация программы соответствует её спецификации. Когда Dafny не может автоматически доказать правильность программы, пользователь может помочь системе, предоставив дополнительные спецификации.

```
method Max(a: array<int>) returns (m: int)
  ensures forall k :: 0 ⩽ k < a.Length ⟹ a[k] ⩽ m
{
  m := a[0];
  var i := 1;
  while i < a.Length
    invariant 0 ⩽ i ⩽ a.Length
    invariant forall j :: 0 ⩽ j < i ⟹ a[j] ⩽ m
  {
    if a[i] > m { m := a[i]; }
    i := i + 1;
  }
}
```

Листинг 4 — Пример кода на Dafny

1.2.4 Why3

Why3 - система верификации программ, разработанная специально для проверки алгоритмического уровня, а не программного. Why3 предлагает свой язык WhyML, который сочетает функциональный стиль программирования с императивными конструкциями. В Why3 спецификации записываются в виде предусловий и постусловий функций, аналогично Dafny. Однако Why3 также

поддерживает механизм уточнения через клонирование модулей, что позволяет реализовывать абстрактные спецификации и постепенно уточнять их. Why3 генерирует условия верификации, которые затем проверяются с помощью различных SMT-решателей. Система поддерживает интеграцию с различными решателями, включая Z3, Alt-Ergo, CVC4.

```
function sum (n: int) : int
  requires n ⩾ 0
  ensures result = n * (n + 1) / 2
= if n = 0 then 0 else n + sum (n - 1)
```

Листинг 5 — Пример кода на Why3

1.2.5 Flux

Flux - это система уточняющих типов для языка Rust, показывающая, как логические уточнения могут работать совместно с механизмами владения (ownership) Rust для обеспечения верификации на основе уточняющих типов. Flux использует SMT-решатели для проверки логических ограничений в уточнениях типов. Система автоматически выводит инварианты для контейнеров, что значительно упрощает верификацию программ, работающих с коллекциями.

```
#[flux::sig(fn (a: &strg i32[@n]) → i32[n+1])]
fn incr(a: &mut i32) {
  *a += 1;
}
```

Листинг 6 — Пример кода на Flux

1.3 Резюме

Современные системы уточняющих типов, такие как LiquidHaskell, Why3 и Flux, демонстрируют впечатляющие возможности в верификации программ, но их архитектурные особенности создают барьеры для распростране-

ния технологии уточняющих типов. Анализ их реализации и опыта интеграции (например, Flux для Rust, LiquidHaskell для Haskell) выявляет проблемы, которые универсальный фреймворк мог бы решить. Архитектурная фрагментация становится основной преградой для повторного использования наработок. Каждая система, как показано в примере с Flux, глубоко интегрируется с особенностями базового языка. Это приводит к дублированию усилий: реализация таких компонентов, как управление SMT-решателями или генерация условий верификации, воссоздается заново для каждого языка. Универсальный фреймворк мог бы стандартизировать представление предикатов и условий корректности, предоставив API для работы с уточняющими типами. Это позволит сократить время реализации проверки уточняющих типов в целевом языке и сосредоточиться на других компонентах языка

2 ПОСТАНОВКА ЦЕЛИ И ЗАДАЧ

3 ПРОЕКТИРОВАНИЕ

3.1 Язык предикатов

Уточняющие типы дополняют стандартные системы типов логическими предикатами для определения и проверки семантических свойств программ. При разработке фреймворка с уточняющими типами, выбор базовой логики для 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.

3.1.1 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.

3.1.2 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 кванторами (FOL)
   - **Разрешимость**: Неразрешимость в целом, ограниченные фрагменты являются нишевыми и в меньшей степени поддерживаются инструментами
   - **Выразительность**: Обеспечивает поддержку предикатов с кванторами, например $\forall i.(0 \leq i \leq n) \implies \text{arr}[i] > 0$.
   - **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 существен-

но 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, автоматиче-

ский 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

3.1.3 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

p	::=	x, y, z	*Переменные*
	|	*true, false*	*Логические значения*
	|	p *OP* p	*Интерпретируемые операторы*
	|	$p \wedge p$	*Конъюнкция*
	|	$p \vee p$	*Дизъюнкция*
	|	$\neg p$	*Отрицание*
	|	$f(p_1, p_2, ..)$	*Неинтерпретируемые функции*
	|	$k(x, y, ..)$	*Переменные Хорна*

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

3.2 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

3.2.1 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 его применение будет показано для решения широкого ряда задач от вывода предикатов, до реализации функций усиливающих предикат

3.2.2 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: $\text{Int} \to \text{Bool}$

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 переменных, которые связывают предикаты. Например тип функции $x : \{v : \text{Int} \mid v > 0\} \to y : \{v : \text{Int} \mid v > 0\} \to \{v : \text{Int} \mid v = x + y\}$ является зависимым, так как тип результата (его уточнение) зависит от аргументов функции, а сигнатура этого типа будет $\text{Int} \to \text{Int} \to \text{Int}$, которая успешно кодируется в сорта SMT.

Основываясь на сортах, поддерживаемых SMT-LIB, сигнатура типов должна поддерживать

– примитивные типы данных

– типы данных, в том числе параметрические

– функции

3.3 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 7

```
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 7 — 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, полученная внутри фреймворка сигнатура будет переводиться в формат SMT-LIB. Таким образом, определение и использование неинтерпретируемых функций в модельном языка будет выглядит однородно и лаконично

3.4 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.

3.5 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.

3.5.1 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

4 РЕАЛИЗАЦИЯ

В предыдущей главе были описаны требования и контракты которые должен предоставлять итоговый фреймворк. В этой главе будет рассмотрена реализация самого фреймворка, а в следующей главе его применение для реализации проверки уточняющих типов в подмножестве языка ocaml

Языком для реализации был выбран Haskell из следующих соображений

- Богатая система типов позволяющая давать гарантии корректности при построении и обходе выражений, а также поддержка полиморфизма при помощи классов типов
- Поддержка алгебраических типов данных и сопоставления с образцом, позволяющие легко моделировать синтаксис модельного языка и упрощающие рекурсивных обход синтаксических деревьев
- Система типов позволяет гарантировать отсутствие побочных эффектов там где они не нужны. Это сильно повышает тестируемость и предсказуемость кода.
- Богатая экосистема. На Haskell уже реализованы уточняющие типы для самого языка в фреймворке LiquidHaskell. LiquidHaskell является ярким примером реализации уточняющих типов, на который можно ссылаться при реализации. Также в экосистеме есть необходимый набор библиотек для парсеров, работы с SMT-решателями, форматирования ошибок и работы со сложными рекурсивными структурами данных

4.1 Используемые инструменты

Данная работа сильно полагается на два инструмента написанных на Haskell.

4.1.1 free-foil

Проверка типов являются важными компонентами реализации языка программирования, ответственными за обеспечение безопасности типов и выявление ошибок в момент компиляции. Одним из основных аспектов проверки типов является управление областями видимости - процесс отслеживания объявлений переменных и их типов в различных участках кода. Несмотря на кажущуюся простоту, управление областями в средствах проверки типов сопряжено с многочисленными трудностями, которые усложняют реализацию и приводят к ошибкам.

Основные проблемы возникают в случаях

– Использования того же имени, что и у переменной выше по области видимости

– Переименовании переменных так, чтобы избежать конфликта имен с уже существующими переменными

1) Концепция Free foil использует идею предложенную в foil для отслеживания областей видимости. Каждое выражение параметризуется фантомным типом `n :: S` который определяет область в которой это выражение используется. Используя возможности системы типов Haskell, в момент компиляции проверяется, что работа с выражениями идет строго с учетом их области видимости.

На основе предоставленной пользователем сигнатуры своего языка, которая включает в себя все используемые конструкции, генерируется абстрактное синтаксическое дерево параметризованное фантомными типами как указано выше.

2) Представление абстрактного синтаксического дерева

```
-- Рекурсивный тип описывающий выражения в области n
data AST binder sig n where
  -- | Свободная переменная в области n.
  Var :: Foil.Name n → AST binder sig n
  -- | Другие синтаксические конструкции в области n
  --   основанные на сигнатуре языка
  Node
    :: sig (ScopedAST binder sig n) (AST binder sig n)
    → AST binder sig n
-- Рекурсивный тип описывающий подвыражение
-- с расширенной областью l
data ScopedAST binder sig n where
  ScopedAST
  -- | Тип описывающий как расширяется область из n в l
    :: binder n l
  -- | Выражение в области l
    → AST binder sig l
    → ScopedAST binder sig n
```

Листинг 8 — Описание абстрактного синтаксического дерева во free-foil

– sig - сигнатура языка, описывающая конструкции языка

– binder - тип описывающий то как расширяется область видимости из n в l. binder может быть как введением одной переменной например $\lambda x.t$, где все выражение имеет область типа n, а t уже имеет область типа l. введение переменной x расширяет область из n в l. Тип binder может быть и сложнее, описывая введение более чем одной переменной в область видимости, например в конструкция сопоставления с образцом $\mathrm{Cons}(\mathrm{x}, \mathrm{xs}) \to t$.

– n и l - фантомные типы описывающие текущую область видимости

3) Подстановка

Имея показанную на Листинг 8 структуру синтаксического дерева можно предоставить обобщенную подстановку не зависящую от модельного языка. Подстановка реализована на основе класса `Sinkable` гарантирующего безопасное переименование переменных с учетом областей видимости.

Подводя итог, free-foil дает следующие преимущества которые сильно помогут при реализации фреймворка

1) Безопасная работа с областями видимости
2) Универсальность

Поддерживает произвольные языки с помощью пользовательских сигнатур. Позволяя реализовывать универсальные алгоритмы (например, подстановку), которые работают на разных языках без повторной реализации

3) Интеграция с существующими инструментами

Совместимость с генераторами парсеров и AST, такими как BNF Converter. Такой подход позволяет автоматически преобразовывать синтаксические определения из грамматики BNF в безопасное представления описанное выше.

4.1.2 liquid-fixpoint

liquid-fixpoint это фреймворк для решения ограничений в формате CHC (Constrained Horn clause). liquid-fixpoint использует логический язык первого порядка для представления ограничений со следующими конструкциями

1) Объявление переменных с указанием типа (переводятся в smt сорта)
2) Предикаты, например: конъюнкции, дизъюнкции, отрицания, константы
3) Квалификаторы: Предоставленные пользователем подсказки для помощи решателю
4) Неинтерпретируемые функции, например len
5) CHC

liquid-fixpoint интегрирован с SMT-решателями, такими как Z3 и CVC4. Также важной особенностью является то, что перед тем как отправить ограничение решателю, liquid-fixpoint их оптимизирует

Поскольку liquid-fixpoint реализован на Haskell, имеет богатый API и встроенную поддержку вывода предикатов, а также используется при решении ограничений в проверенном временем LiquidHaskell, было принято решение выбрать этот фреймворк как основу для работы с ограничениями.

4.2 Компоненты фреймворка

В главе проектирования были описание основные компоненты и требования которые должны быть реализованы во фреймворке. В этой главе будут подробно рассмотрены принятые решения, предоставленные интерфейсы и описание реализация основных функций

4.2.1 Окружение

В обзоре free-foil, было показано, что этот фреймворк предоставляет ряд абстракций которые позволяют обобщенно работать с переменными (см. Var) и их объявлением через параметр binder, который описывает как новые переменные расширяют область видимости. С другой стороны уточняющие

типы являются зависимыми. Если посмотреть на тип функции sum, то он будет выглядеть следующим образом

$$x : \{v : \text{int} \mid \text{true}\} \to y : \{v : \text{int} \mid \text{true}\} \to \{v : \text{int} \mid v = x + y\}$$

На этом примере видно, что уточнение в результирующем типе зависит от данных которые которые передаются в аргументы функции. Из этого также видно что у нас типы сами по себе содержат область видимости

- $x : \{v : \text{int} \mid \text{true}\}$ находится в области n_1
- $y : \{v : \text{int} \mid \text{true}\}$ расширяется переменной x и находится в области n_2
- $\{v : \text{int} \mid v = x + y\}$ расширяется переменной y и находится в области n_3
- а уточнение $v = x + y$ также расширяется переменной v и уже находится в области n_4

Исходя из этой особенности фреймворку нужно предоставить некоторое окружение которое бы предоставляло возможность связывать переменные с типами, с учетом их областей видимости

```
data Env sig binder n where
  -- Конструктор описывающий пустое окружение
  EmptyEnv :: Env sig binder Foil.VoidS
  -- Конструктор описывающий расширение окружения одной переменной
  NonEmptyEnv ::
    (Foil.CoSinkable binder, Foil.DExt n l) ⇒
    -- Окружение в предыдущей области n
    Env sig binder n →
    -- Новая переменная расширяющая область с n до l
    Foil.NameBinder n l →
    -- Тип связанный с именем, описанный в виде AST free-foil
    Foil.AST binder sig n →
    -- Новое окружение в области l
    Env sig binder l
```

Листинг 9 — определение окружения

В текущем варианте фреймворка окружение реализовано как связанный список с использованием механизма обобщенных алгебраических типов данных (GADTs) и показано на Листинг 9. Список расширяется из окружения `Foil.VoidS`, где нет никаких переменных, до окружения с переменными. Каждое новое определение добавляется с учетом текущего окружение и расширяет его из области `n`, до области `l` с при помощи `Foil.NameBinder n l`. Класс типов `Foil.DExt n l` служит подсказкой компилятору Haskell о том, что область `n` расширяется до `l`

Функции для работы с окружением выглядят следующим образом

1) Расширение окружения является конструктором `NonEmptyEnv` и имеет такую же сигнатуру

```
extendedEnv ::
  (Foil.DExt i o, Foil.CoSinkable binder) ⇒
  Env sig binder i →
  Foil.NameBinder i o →
  Foil.AST binder sig i →
  Env sig binder o
extendedEnv env binder typ = NonEmptyEnv env binder typ
```

2) Поиск типа связанного с переменной выглядит интереснее. В первую очередь благодаря подходу free-foil мы можем гарантировать, что если у нас есть свободная переменная в области n, и есть окружение в области l, и область l расширяет область n (проверяется через связку классов `Foil.DExt`), то у нас гарантированно заданная переменная есть в окружении. Благодаря чему мы можем сделать функцию поиска в окружении тотальной. Код функции представлен на Листинг 10. В коде несколько раз встречается `error "impossible case"`, это сделано чтобы сделать сопоставление с образом исчерпывающем, и во время

исполнение не срабатывают. Из сигнатуры функции можно увидеть что возвращаемый тип должен реализовывать класс IsType. Про него будет рассказано в чуть дальше

```
lookupEnv ::
  (Bifunctor sig, IsType sig binder) ⇒
  -- Текущее окружение
  Env sig binder o →
  -- Имя переменной
  Foil.Name o →
  -- Тип связанный с переменный
  Foil.AST binder sig o
lookupEnv env varId = case env of
  EmptyEnv → error "impossible case"
  NonEmptyEnv env' binder term →
    if Foil.nameOf binder == varId
      then Foil.sink term
      else
        case Foil.unsinkName binder varId of
          Nothing → error "impossible case"
          Just varId' → Foil.sink $ lookupEnv env' varId'
```

Листинг 10 — Поиск в окружении типа связанного с переменной

Уже было описана важная гарантия того, что поиск связанного с переменной типа в окружении всегда тотальный, но есть и другая важная особенность: благодаря free-foil у нас есть единообразный формат работы с именами, которые хранятся как ключи в окружении. В дальнейшем это пригодится для генерации переменных Хорна.

4.2.2 Типы

Работа с типами описывается в виде следующего контракта, представленного на Листинг 11.

```
class IsType sig binder where
  withPred ::
    Foil.Distinct n ⇒
    Foil.AST binder sig n →
    (WithPred sig binder n → (a, WithPred sig binder n)) →
    (Maybe a, Foil.AST binder sig n)
  toTypeSignature :: Foil.AST binder sig n → TypeSignature
```

Листинг 11 — Контракт работы с типами

4.2.2.1 Модификация предикатов внутри типов

Функция `withPred` в первую очередь нужна для того, чтобы фреймворк и его пользователи могли получать из типов их предикаты, работать с ними, модифицировать и делать на их основе вычисления. Напомним, что типы уточнения выглядят следующим образом $\{v : \text{int} \mid \text{true}\}$. В то же время в языке могут быть типы которые не имеют уточнений, например функции.

$$\text{arg} : \{v : \text{int} \mid \text{true}\} \to \{v : \text{int} \mid \text{true}\}$$

На данном примере видно, что у типа аргумента и результирующего типа есть свои предикаты, но сам тип функции предиката не имеет.

Давайте подробнее разберем сигнатуру функции `withPred`

```
1  withPred ::
2    Foil.Distinct n ⇒
3    Foil.AST binder sig n →
4    (WithPred sig binder n → (a, WithPred sig binder n)) →
5    (Maybe a, Foil.AST binder sig n)
6
7  data WithPred sig binder i where
8    WithPred ::
9      ( IsPred sig binder
10     , Foil.DExt i o
11     ) ⇒
12     Foil.NameBinder i o
13     → Foil.AST binder sig o
14     → WithPred sig binder i
```

– На строке 3 передается тип в таком же формате, как и в окружение, через абстрактное синтаксическое дерево предоставляемое free-foil

– На строке 4, вторым аргументом, передается функция которая производит работу с предикатом внутри типа, если этот предикат есть, и возвращает посчитанное на его основе значение, а также модифицированный предикат (также может остаться без изменений). В случае

если модифицируется только предикат и дополнительные вычисления не производятся, то возвращаем значением можно быть элемент типа `()`, также известного как unit имеющий только одно значения.

– На строке 5 показаны возвращаемые значение. Возвращается кортеж из двух элементов

    1) Опциональное посчитанное значение. Значение присутствует, если у типа был предикат
    2) Заново сконструированный тип, с измененным предикатом

Также был введен специальный тип данных WithPred – экзистенциальный тип данных, где переменная o появляется только во внутренней сигнатуре, и не появляется в самом типе `WithPred sig binder i`. Это решает две проблемы

1) Кодирование фантомного типа o который появляется только при распаковке типа, и будет каждый раз новый
2) Сохранение гарантии, что предикат внутри WithPred реализует контракт предикатов

4.2.2.2 Сигнатура типа

Как было описано в главе проектирования, сигнатуры типов нужны для определения типов выражений в SMT-решателе. Это требование также является частью контракта работы с типами из Листинг 11. Финальный вариант сигнатуру типов выглядит представлен на Листинг 12

```
1 data TypeSignature
2   = BoolType
3   | IntType
4   | VarType Id
5   | DataType Id [TypeSignature]
6   | FunType TypeSignature TypeSignature
7   | ForallType Id TypeSignature
```

Листинг 12 — Определение сигнатуры типов в фреймворке

1) Id представляет строковый идентификатор переменных и имен типов
2) BoolType и IntType представляют bool и int соответственно, и являются интерпретируемыми типами
3) VarType представляет переменную типа для параметрического полиморфизма и представляется в виде неинтерпретируемого типа
4) DataType также представляет собой неинтерпретируемый тип, с возможность параметризации другими типами
5) FunType представляет собой тип функций, функции с арностью могут быть представлены как связка нескольких FunType, например

```
FunType IntType (FunType IntType BoolType)
```

6) ForallType вводит переменную типа в область видимости типа указанного вторим аргументом

Важно заметить что сигнатура типов не представлена в виде AST из free-foil, это сделано намеренно по причине того, что пользовательские типы уже представлены в виде этого AST и при конвертации будет гарантированна уникальность имен. Эта гарантия избавляет от проблем с конфликтом имен и затенением переменных с учетом того, что дополнительные операции с сигнатурой типов не производятся.

```
toTypeSignature :: Foil.AST binder sig n -> TypeSignature
```

Сама функция конвертации требуется в контракте работы с типами и по сути является обходом типа с отбросом всех встречающихся предикатов

4.2.3 Предикаты

Грамматика предикатов описана ровно так как предполагалось на моменте проектирования и показана на Рис. 1. Операторы указанные как OP в грамматике представлены следующим набором $=, \leq, <, \geq, >, +, -, *, \wedge, \vee$

Контракт который должны реализовывать предикаты модельного языка показан на Листинг 13

```
class IsPred sig binder where
  isUnknown :: Foil.AST binder sig n  → Bool
  mkAnd :: Foil.AST binder sig n →
    Foil.AST binder sig n → Foil.AST binder sig n
  mkEq :: Foil.AST binder sig n →
    Foil.AST binder sig n → Foil.AST binder sig n
  mkHornVar :: String → [Foil.Name n] → Foil.AST binder sig n
  toPredicate :: Foil.AST binder sig n → P.Pred
```

Листинг 13 — Контракт предикатов

– `isUnknown, mkHornVar` используются для вывода предикатов и описаны в Раздел 4.2.3.2

– `mkAnd, mkEq` нужны для реализации функций усиления предикатов независящих от модельного языка, по сути они просто создают узлы в синтаксическом дереве предикатов с равенством и конъюнкцией

– `toPredicate` необходим для конвертации предикатов модельного языка в предикаты фреймворка

4.2.3.1 Неинтерпретируемые функции

Неинтерпретируемые функции представлены следующим образом

```
data Measure = Measure
  { measureName :: String
  , sort :: TypeSignature
  }
```

Неинтерпретируемые функции могут быть описаны на модельном языке используя типы реализующие контракт указанный выше. Сами функции регистрируются следующим образом

```
mkMeasure ::
  C.IsType sig binder ⇒
  -- Идентификатор функции
  String →
```

```
5   -- Тип функции
6   Foil.AST binder sig n →
7   Measure
```

4.2.3.2 Вывод предикатов

Для вывода предикатов нам нужно объявить переменные Хорна с их сигнатурой, которые будут описывать предикат. а также вставить применение зарегистрированной переменной к аргументам из окружения. Для каждого нового вывода нужно создать новые переменную с уникальным именем. Имена создаются по следующему принципу $k_1, k_2, .., k_n$. Это приводит нас к необходимости поддерживать состояние которое бы хранило все существующие переменные, а также поддерживало индекс следующей.

```
1 data HornVarsState = HornVarsState
2   { nextHornVarIndex :: Int -- Индекс следующей переменной
3   , hornVars :: [HornVar] -- Текущий переменные
4   }
```

По скольку состояние изменяется с ходом генерации условий корректности, а Haskell является языком с неизменяемыми переменными, то поддержка корректного состояния ложится на плечи пользователя. К счастью в Haskell есть множество инструментов для работы с изменяемыми состояниями, например монада State, а также изменяемые участки памяти, работающие в IO, такие как IORef и MVar.

Места для вывода предикатов должны быть помечены специальной заглушкой, на место которой встанет применение переменной Хорна к окружению. Заглушки могу встречаться в аннотациях типов выставленных программистом, а также в местах уточнения типа полиморфных функций и типов

данных после их вывода. Для понимания является ли предикат заглушкой в контракт предикатов добавлена функция

```
1 isUnknown :: Foil.AST binder sig n  → Bool
```

В пункте про окружение (см Раздел 4.2.3.2), показывались возможности которое дает предоставленное окружение. В первую очередь все имена переменных описываются через `Foil.Name`, это позволяет нам генерировать новые имена переменных без боязни затенить переменные выше по области видимости, а также и расширять AST модельного языка вставкой свободных переменных через `Var`

Давайте подробно рассмотрим алгоритм замены заглушек на применение переменных Хорна с использованием контрактов которые мы уже ввели.

В первую очередь определим сигнатуру функции которая будет производить замену

```
1 freshTypeWithPredicate ::
2   (Bifunctor sig, Foil.Distinct n, IsType sig binder) ⇒
3   HornVarsState →
4   Env sig binder n →
5   Foil.AST binder sig n →
6   (Foil.AST binder sig n, HornVarsState)
```

В аргументы функции мы передаем состояние `HornVarsState` которое хранит текущие переменные хорна, окружение с переменными и их типами, а так же сам тип в формате AST free-foil. На выходе мы получаем кортеж из типа с модифицированным предикатом и новое состояние. Если же у типа не было предиката, или предикат не был заглушкой, то тип вернется как был, а состояние переменных не изменится.

Первым шагом, на основе окружения нужно собрать все имена и их типы, у которых есть предикат (типы без предикатов не участвуют в условиях корректности). Делается это следующим образом

```
envSorts = flip mapMaybe (envToList env) $ \(name, envTyp) →
  fst $ withPred envTyp $ \withPredInst →
    let sort = toTypeSignature envTyp
    in ((name, sort), withPredInst)
```

1) Преобразуем окружение в список пар имен и их типов
2) Делаем проход по полученным парам, и смотрим на их типы. Если у типа был предикат, то вычисляем его сигнатуру, иначе отбрасываем такую пару.
3) На выходе получаем список всех имен переменных и их сигнатуры типов

Следующим шагом нужно обработать сам тип который был передан аргументом

```
mTypeSort = join . fst $
 withPred argType $ \wp@(WithPred _ p) →
   let
     sort = if isUnknown p
       then Just $ toTypeSignature argType
       else Nothing
   in (sort, wp)
```

Здесь схема аналогична предыдущему шагу, за исключением того, что помимо наличия предиката, также проверяется то, что он является заглушкой

Последним шагом объявляется переменная Хорна и предикат типа заменяется на ее применение

```
 1 case mTypeSort of
 2   Just typSort →
 3     let
 4       (hornVarName, refinementState')
 5         = mkFreshHornVar refinementState
 6             (typSort : map snd envSorts)
 7       freshedType = snd $ withPred typToFresh
 8         $ \(WithPred pNameBinder _) →
 9           let
10             newPred = mkHornVar hornVarName
11               ( Foil.nameOf pNameBinder : map fst envSorts )
12           in ((), WithPred pNameBinder newPred)
13     in (freshedType, refinementState')
14   _ → (typToFresh, refinementState)
```

На основе предыдущего шага делается решение нужно ли модифицировать предикат и если да, то делаются следующие операции

– Создается новая переменная Хорна (5 строка) с учетом сигнатуры модифицируемого типа и сигнатур типов из окружения. На выходе получается имя новой переменной и обновленное состояние

– В строке 7 объявляется модифицированный тип через `withPred`. Внутри типа предикат заменяется на применение новой переменной Хорна при помощи еще одного элемента контракта предикатов `mkHornVar`

```
1 mkHornVar ::
2   -- Имя переменной
3   String →
4   -- Имена аргументов
5   [Foil.Name n] →
6   -- Новый предикат в синтаксисе модельного языка
7   Foil.AST binder sig n
```

Тут важно подметить что имена переменных используют String, а не формат имени из free-foil, это потому что имена переменных уникально глобально, а не в рамках скоупов, из-за чего отдельная работа с механизмами free-foil не требуется

На выходе получается готовый вывода предикатов в модельном языке, где для его работы нужно только организовать работу с состоянием переменных и вызывать `freshTypeWithPredicate` для типов где могут быть заглушки

4.2.4 Ограничения

Ограничения представлены в виде следующей структуры, описывающей предложенный в момент проектирования вариант Рис. 2

```
1 data Constraint
2   = CPred Pred ErrorMessage
3   | CAnd [Constraint]
4   | CImplication VarId TypeSignature
5       Pred Constraint ErrorMessage
```

– `ErrorMessage` представляет из себя текстовое поле которое пользователь указывает во время генерации ограничений. Оно служит для описания контекста где произошла ошибка и может содержать данные о месте, переменных и узлах синтаксического дерева спровоцировавших ошибку

– `CAnd` объединяет несколько ограничений в одно

– `CPred` это лист дерева ограничений, задающий предикат. Этот узел используется как базовый элемент проверки подтипирования уточняющих типов. Из правила ниже в CPred будет записан $q[w := v]$

$$\frac{\Gamma \vdash \forall (v : t).p \Rightarrow q[w := v]}{\Gamma \vdash b\{v : p\} <: b\{w : q\}}\ \text{Sub-Base} \quad (1)$$

Рисунок 4 — Правило подтипирования базовых типов с уточнением

– `CImplication` описывает узел вида $\forall (v : t).p \Rightarrow c$, где v это VarId, t сигнатура типа, p предикат, c ограничение

Функции для создания ограничений выглядят следующим образом

1) `cAnd` дублирует конструктором и служит для объединения ограничений из разных узлов программы

```
cAnd :: [Constraint] → Constraint
cAnd = CAnd
```

2) `cPed` принимает предикат реализующий контракт `IsPred` и переводит его в формат фреймворка

```
cPred :: IsPred sig binder ⇒ Foil.AST binder sig n →
  Text → Constraint
cPred p = CPred (toPredicate p)
```

3) `cImplication` хитрее, потому что импликация возникает в случаях введения новой переменной с ее типом в область или при подтипировании

```
cImplication ::
  ( IsType sig binder, Foil.Distinct n
  , Bifunctor sig, Foil.CoSinkable binder) ⇒
  -- специальный тип из free-foil хранящий все имена переменных
  Foil.Scope n →
  -- имя переменной связанное с типом
  Foil.NameBinder n l →
  -- тип
  Foil.AST binder sig n →
  -- ограничение с правой стороны импликации
  Constraint →
  -- сообщение ошибки при нарушении ограничения
  ErrorMessage →
  -- ограничение с импликацией
  Constraint
```

Листинг 14 — Создание импликации из типа

Алгоритм создания импликации выглядит следующим образом. Представим, что у нас есть переменная x с типом $\{v : \mathrm{int} \mid v \geq 0\}$. В функцию мы принимаем отдельно `Foil.NameBinder` для x и сам тип,

это сделано для того, чтобы в предикате $v \geq 0$ заменить все вхождения v на x. Такой шаг нужен для того чтобы связать уточнение типа с переменной, которая будет использования в окружении и в ограничениях ниже по дереву (правая сторона импликации). А в случае правила подтипирования (см. Рис. 4) на место имени переменной x передается v из самого уточнения. Замена вхождения v на x происходит благодаря free-foil, предоставляющего независимую от языка подстановку, что позволяет не требовать от пользователей фреймворка реализовывать ее в виде еще одного контракта

4.3 Резюме

Полученный фреймворк предоставляет следующие основные компоненты

1) Окружение. Реализовано с использование GADTs и free-foil для безопасного управления областями видимости. Поддерживает связывание переменных с их типами, включая уточнения, с гарантией тотальности поиска
2) Типы. Предоставлен контракт IsType для работы с типами, включая модификацию предикатов и преобразование в сигнатуры типов для SMT-решателей
3) Предикаты. Грамматика соответствует логике QF_UFLIA (без кванторов, линейная арифметика, неинтерпретируемые функции). Предоставлены функции для работы с предикатам в модельном языке, проверки «заглушек» и генерации переменных Хорна для вывода предикатов.

4) Ограничения. Автоматическая генерация ограничений на основе контрактов работы с типами и предикатами.

Ключевыми возможностями является

1) Безопасное управление областями видимости: Гарантируется корректная работа с переменными, избегание конфликтов имён.
2) Автоматический вывод предикатов: Замена «заглушек» на переменные Хорна с последующим выводом уточнений через SMT-решатели.
3) Интеграция с SMT: Автоматическая проверка условий корректности, включая поддержку неинтерпретируемых функций и линейной арифметики.
4) Расширяемость: Возможность добавления пользовательских типов и предикатов через контракты (`IsType`, `IsPred`).

5 РЕАЛИЗАЦИЯ ЯЗЫКА НА ФРЕЙМВОРКЕ

В предыдущей главе была показана реализация фреймворка для проверки и вывода уточняющих типов, но помимо реализации самого фреймворка также надо оценить его применимость. Показателем качества и применимости фреймворка предлагается считать по следующим пунктам

1) Пользователю не требуется работать с предикатами, если он самостоятельно их не модифицирует
2) Контракты достаточно мощные, для реализации необходимых возможностей модельного языка
3) Предоставленные ограничения покрывают потребности проверки уточняющих типов в модальном языке

Для этого на фреймворке была реализована проверка уточняющих типов для подмножества языка ocaml со следующими возможностями

1) Ветвление и рекурсия
2) Функции высших порядков
3) Вывод предикатов
4) Параметрический полиморфизм и вывод типов
5) Полиморфные алгебраические типы данных
6) Сопоставление с образцом

Данный список возможностей реализован во многих функциональных языках и способен показать применимость фреймворка. Сам язык в дальнейшей работе будет называться модельный язык. Важно понимать, что работа велась только над подмножеством ocaml с указанными возможностями

5.1 Синтаксис языка

Основные конструкции языка представлены в следующем виде на Листинг 15 в синтаксисе LBNF (Labeled BNF), где каждому выражению присвоено свое имя.

```
-- 0
ConstInt.     Term ::= Integer;
-- true
Bool.         Term ::= ConstBool ;
-- x
Var.          Term ::= VarIdent;
-- Cons(x, xs)
ConApp.       Term ::= ConIdent ConAppArgs;
-- foo(a, b, c)
FunApp.       Term ::= VarIdent "(" [FuncAppArg] ")";
-- if (true) {x} else {y}
If.           Term ::=
  "if" "(" FuncAppArg ")" "{" Term "}" "else" "{" Term "}";
-- let rec x = 0 in x
Let.          Term ::= Decl Term ;
-- (x) ⇒ {x}
Fun.          Term ::= "(" [FunArgName] ")" "⇒" "{" Term "}" ;
-- x + y
Op.           Term ::= FuncAppArg IntOp FuncAppArg;
-- switch (xs) { | Nil ⇒ 0}
Switch.       Term ::=
  "switch" "(" VarIdent ")" "{" [SwitchCase] "}";
```

Листинг 15 — Синтаксис основных конструкций

Объявление функций и переменных происходит через связку `let in`, где у переменных и функций могут присутствовать аннотации типов как показано на Листинг 16

```
/*@ val inc: x:int ⇒ int[v|v == x + 1] */
let inc = (x) ⇒ {
    x + 1
};
```

Листинг 16 — Пример аннотации типов в функции abs

Синтаксис типов представлен на Листинг 17. Предикаты описаны эквивалентно тому, что представлено во фреймворке

```
KnownRefinement.   Refinement ::= "[" VarIdent "|" Pred "]";
UnknownRefinement. Refinement ::= "[" "?" "]";
SimpleRefinement.  Refinement ::= ;

-- types
TypeFun.             RType ::= FuncArg "⇒" RType ;
TypeRefined.         RType ::= BaseType Refinement;
TypeData.            RType ::= VarIdent TypeDataArgs Refinement;
_.                   RType ::= "(" RType ")";
```

Листинг 17 — Синтаксис типов и уточнений

Важным замечанием будет уточнить то, что в самой реализации есть два синтаксиса и AST сгенерированных BNFC

1) Представлен на листингах выше, разделены типы и выражения
2) Объединяет типы и выражения под один тип, в первую очередь это сделано для того что бы типы и выражения имели одни области видимости и позволяли делать переименование переменных между друг другом

По сути оба синтаксиса эквиваленты друг другу, но отличаются представлением AST и упрощением некоторых узлов. Конвертация из AST полученного после разбора текста в первый синтаксис реализована наивной конвертацией из одного дерева в другое. На основе сгенерированного BNFC AST, генерируется следующая (см. Листинг 18) сигнатура языка для free-foil. На основе представленного типа данных можно штатными методами вывести стандартные классы типов необходимые для работы free-foil `deriving (Generic, Functor, Foldable, Traversable)`, а также `Bifunctor, Bifoldable, Bitraversable` за счет инструментов из библиотеки bifunctors

```
data TermSig scope term where
  ConstIntSig :: Integer → TermSig scope term
  BooleanSig :: ConstBool → TermSig scope term
  ConstructorSig :: ConIdent → TermSig scope term
  IfSig :: term → term → term → TermSig scope term
  LetSig :: term → scope → TermSig scope term
  LetRecSig :: term → scope → scope → TermSig scope term
  FunSig :: scope → TermSig scope term
  AppSig :: term → term → TermSig scope term
  AnnSig :: term → term → TermSig scope term
  OpExprSig :: term → Op → term → TermSig scope term
  SwitchSig :: term → [term] → TermSig scope term
```

Листинг 18 — Сигнатура основных конструкций языка

Типы представленные в том же типе и сигнатуре следующим образом

```
TypeRefinedSig :: term → scope → TermSig scope term
TypeFunSig :: term → scope → TermSig scope term
TypeForallSig :: scope → TermSig scope term
TypeDataSig :: VarIdent → [term] → scope → TermSig scope term
```

Листинг 19 — Сигнатура типов языка

Также надо обозначить тип, которые описывает как различные наборы переменных расширяют область видимости. В нашем случае есть только три конструкции расширяющие область видимости (см. Листинг 20), это

1) `PatternNoBinders` и `PatternSomeBinders`, которые описывают расширение области от 0 до n переменных в конструкции `switch case`, например

   ```
   switch (xs) {
     | Nil        ⇒ ys
     | Cons(h, t) ⇒ append(t, ys)
   }
   ```

2) `PatternVar` описывающее расширение области ровно одной переменной. Применяется во всех остальных случаях

```
data Pattern o i where
  PatternVar :: (Foil.NameBinder o i) → Pattern o i
  PatternNoBinders :: Pattern o o
  PatternSomeBinders ::
    (Foil.NameBinder o i1) →
    (Pattern i1 i2) →
    Pattern o i2
```

Листинг 20 — Тип для расширения области видимости

Само же AST будет выглядеть как синоним для `Foil.AST` и `Foil.ScopedAST` примененные к сигнатуре языка.

```
type Term = Foil.AST Pattern TermSig
type ScopedTerm = Foil.ScopedAST Pattern TermSig
```

Поскольку все конструкции языка описываются через рекурсивную связку `Node` и `ScopedAst` из free-foil, то удобным будет добавить синонимы сопоставления с образцом, по следующему формату

```
pattern Let :: Pattern o i → Term o → Term i → Term o
pattern Let binder val body = Foil.Node
  (LetSig val
    (Foil.ScopedAST binder body))
```

Листинг 21 — Синонимы сопоставления с образцом для AST

Такие синонимы сделаны для каждой конструкции в языке.

5.2 Реализация контрактов фреймворка

Для реализации контракта `IsType` нужно представить две метода

1) `withPred` для извлечения предиката из типа, проведения над предикатом вычислений и обратное конструирование типа. В случае подмножества ocaml, для которого реализуется проверка типов, предикаты есть только у типов данных. В AST они разделены на два вида
    a) `TypeRefined` - отвечающий за базовые типы, например `int[v|v >= 0]`, `bool[b|b]`

б) `TypeData` - отвечающий за пользовательские алгебраические типы данных, например `list(int)[v|len(v) >= 0]`

Реализация для TypeRefined показана на Листинг 22.

```
1 withPred (TypeRefined base (PatternVar v) p) f =
2   case (Foil.assertDistinct v, Foil.assertExt v) of
3     (Foil.Distinct, Foil.Ext) →
4       case f $ Refinements.WithPred v p of
5         (res, Refinements.WithPred v' p') →
6           ( Just res
7           , TypeRefined base (PatternVar v') p')
```

Листинг 22 — Реализация `withPred` для базовых типов

Если подробнее рассмотреть код по строчно, то алгоритм выглядит следующим образом

– На 1 строчке разбираем тип на базовый - base, переменную `v` связывающую данные с предикатам, и предикат p. На вход принимаем функцию `f` для модификации предиката
– 2 и 3 строчки нужны для компилятора haskell, что бы сказать ему что классы free-foil будут выполнять и в области расширенной переменной `v`
– На 4 строке Пакуем `v` и `p` в обертку `WithPred`, и применяем к ней `f`
– На 5 строке распаковываем результат вычисления функции и новый предикат
– На 6 и 7 строке возвращаем вычисление, полученное из применения функции `f` и конструируем тип с новым предикатом, но тем же базовым типов

Для `TypeData` реализация будет аналогичным. Для других же типов, например функций, где предикатов нет, реализация будет вы-

глядеть тривиально. Функцией для предикатов мы ничего не могли посчитать, потому что нет предикатов, а тип остается таким же без модификации

```
withPred t _ = (Nothing, t)
```

Листинг 23 — Реализация `withPred` для типов без предикатов

2) `toTypeSignature` для преобразования типов в формат сопоставимый с сортами SMT-LIB. Для реализации нужно рекурсивно обойти тип как показано на Листинг 24, отбросить предикат и сконвертировать в формат сигнатуры типов из фреймворка.

```
...
toTypeSignature (TypeRefined BaseTypeInt _ _)
  = Refinements.IntType
toTypeSignature (TypeRefined BaseTypeBool _ _)
  = Refinements.BoolType
toTypeSignature (TypeData name args _ _) = Refinements.DataType
  name' args'
  where
    name' = Refinements.Id $ getRawVarId name
    args' = map Refinements.toTypeSignature args
toTypeSignature (TypeFun _ argT retT) = Refinements.FunType
  (Refinements.toTypeSignature argT)
  (Refinements.toTypeSignature retT)
...
```

Листинг 24 — Реализация `toTypeSignature`

Контракт для предикатов `IsPred` требует больше методов, но концептуально они довольно простые

a) `isUnknown` - показывает является ли предикат заглушкой или нет

```
isUnknown Unknown = True
isUnknown _ = False
```

б) `mkHornVar` - Создает предикат в виде применения переменной Хорна

```
mkHornVar name vars = HVar
  (VarIdent name)
  (Foil.Var <$> vars)
```

в) `mkAnd` и `mkEq` - создают узлы применения операторов с одноименным названием к двум других предикатам

г) `toPredicate` - тривиальная обход AST предикатов и преобразование его в формат фреймворка

5.3 Проверка типов

Проверка типов представляет из себя двунаправленную проверку типов, где выражения делятся на два вида:

1) выражения, тип которых можно вывести из контекста $\Gamma \vdash e \Longrightarrow t$
2) выражения, тип которых известен и его надо проверить в контексте $\Gamma \vdash e \Longleftarrow t$

Результатом проверки типов является успех или ошибка проверки базовых типов, а также, в случае успехов, ограничения, корректность которых проверятся при помощи SMT-решателя.

Базой для проверки уточняющих типов является проверка подтипирования, описываемая следующими правилами

$$\frac{\Gamma \vdash \forall v_1 : b.p_1 \Rightarrow p_2[v_2 := v_1]}{\Gamma \vdash b\{v_1 : p_1\} \prec: b\{v_2 : p_2\}}\ \text{Sub-Base}$$

Рисунок 5 — Правило подтипирования базовых типов

$$\frac{\Gamma \vdash s_2 \prec: s_1 \quad \Gamma; x_2 : s_2 \vdash t_1[x_1 := x_2] \prec: t_2}{\Gamma \vdash x_1 : s_1 \to t_1 \prec: x_2 : s_2 \to t_2}\ \text{Sub-Base}$$

Рисунок 6 — Правило подтипирования функций

$$\frac{\Gamma \vdash s_2 \prec: s_1 \quad \Gamma; x_2 : s_2 \vdash t_1[x_1 := x_2] \prec: t_2}{\Gamma \vdash x_1 : s_1 \to t_1 \prec: x_2 : s_2 \to t_2}\ \text{Sub-Base}$$

Рисунок 7 — Правило подтипирования функций

Аналогичное правилу подтипирования базовых типов на Рис. 6 используется и для пользовательских типов данных, только с учетом проверки подтипирования аргументов типа

Сами функции проверки и вывода представлены на Листинг 25 и Листинг 26. `Env` это синоним для окружения из фреймворка и представлен в виде `type Env = LR.Env TermSig Pattern`

```
check :: (F.DExt F.VoidS i) ⇒
  -- Описывает текущий набор переменных в области видимости
  F.Scope i →
  -- Текущее окружение из переменных и их типов
  Env i →
  -- Выражение для проверки
  Term i →
  -- Тип выражения (типы и выражения представлены в одном типе)
  Term i →
  -- На выходе получаем ограничения
  CheckerM LR.Constraint
```

Листинг 25 — Сигнатура функции проверки типов

```
synths ::
  (F.DExt F.VoidS i) ⇒
  F.Scope i →
  Env i →
  -- Выражение тип которого надо вывести
  Term i →
  -- На выходе получаем ограничения и тип выражения
  CheckerM (LR.Constraint, Term i)
```

Листинг 26 — Сигнатура функции вывода типов

На основе рекурсивного вызова `check` и `synths` производится двунаправленная проверка базовых типов и генерация ограничений

5.3.1 Ветвление

Для понимания проблемы которая возникает при ветвлении надо рассмотреть простой код на Листинг 27. В нем на основе if нужно в разные ветки

предоставить информацию что $x \geq 0$ или $x < 0$. Но в конце рекурсивного обхода, при проверка выражения x (ветка истинности), достав из окружения тип переменной x получим $x : \{v : \text{int} \mid \text{true}\}$, что не является подтипом $\{v : \text{int} \mid v \geq 0\}$. Для этого нужно усилить предикат в типе $\{v : \text{int} \mid \text{true}\}$, добавив в него равенство между переменными x и v. В итоге тип у x будет выглядеть как $x : \{v : \text{int} \mid \text{true} \wedge x = v\}$

```
/*@ val abs : x:int ⇒ int[v|0⩽v] */
let abs = (x) ⇒ {
    let pos = x ⩾ 0;
    if (pos) {
        x
    } else {
        0 - x
    }
};
```

Листинг 27 — Функция получения модуля числа

После усиления предиката ограничения будут выглядеть следующим образом

$$\begin{gathered} \forall x, c, y, v.(c = 0 \leq x) \Rightarrow c \Rightarrow v = x \Rightarrow (0 \leq v) \\ \forall x, c, y, v.(c = 0 \leq x) \Rightarrow \neg c \Rightarrow v = 0 - x \Rightarrow (0 \leq v) \end{gathered} \tag{2}$$

Для данной операции фреймворк предоставляет готовую функцию для работы с окружением (см. Листинг 28), которое делает показанное усиление предиката за пользователя. Сама же функция усиления предиката показана на Листинг 29

```
lookupEnvWithStrengthening ::
  (Foil.Distinct o, Bifunctor sig, IsType sig binder) ⇒
  Env sig binder o →
  Foil.Name o →
  Foil.AST binder sig o
lookupEnvWithStrengthening env name = singletonT name resTyp
  where
    resTyp = lookupEnv env name
```

Листинг 28 — Поиск в окружении с автоматическим усилением предиката

```haskell
singletonT ::
  (Foil.Distinct n, IsType sig binder) ⇒
  -- Имя переменной
  Foil.Name n →
  -- Тип с предикатом
  Foil.AST binder sig n →
  -- Тип с усиленным предикатом
  Foil.AST binder sig n
singletonT varName typ = snd $ withPred typ $ \(WithPred pNameBinder p) →
  case (Foil.assertDistinct pNameBinder, Foil.assertExt pNameBinder) of
    (Foil.Distinct, Foil.Ext) →
      let
        newPred = mkAnd
            p -- Старый предикат
            ( mkEq -- x = v
              (Foil.Var $ Foil.sink varName)
              (Foil.Var $ Foil.nameOf pNameBinder)
            )
      in ((), WithPred pNameBinder newPred)
```

Листинг 29 — Поиск в окружении с автоматическим усилением предиката

5.3.2 Вывод предикатов

Весь вывод предикатов реализован в фреймворке и как его пользователю нужно сделать только три вещи

1) Сделать передачу изменяемого состояния с переменными Хорна
2) Реализовать обход типов и вызвать замену заглушек на применение переменных Хорна из фреймворка.
3) Во время проверки уточняющих типов вызывать реализованную в пункте два функции

5.3.3 Вывод типов

Реализованная проверка типов поддерживает вывод типов и параметрический полиморфизм используя алгоритм Хиндли — Милнера, но данная часть работы напрямую не относится к представленной теме диплома. Поэтому хочется подсветить только основной момент, что Вч местах создания экземпляра

полиморфной функции или типа данных с явным указанием типов нужно проставить у типов в предикаты заглушки

5.3.4 Типы данных и сопоставление с образцом

Для пользовательских алгебраических типов данных в первую очередь реализована поддержка работы с предикатами и их представление в сигнатуре типов, которые уже были рассмотрены. Вторым важным критерием была поддержка неинтерпретируемых функций, которые критичны для кодирования логики работы с пользовательскими типами. Ярким примером их использования может быть функция кодирующая длину списка.

```
 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   ;
 7
 8 /*@ val append :
 9   xs:list('a) ⇒ ys:list('a) ⇒ list('a)[v|len(v) == len(xs) + len(ys)] */
10 let rec append = (xs, ys) ⇒ {
11   switch (xs) {
12     | Nil        ⇒ ys
13     | Cons(h, t) ⇒ append(t, ys)
14   }
15 };
```

Листинг 30 — Использование неинтерпретируемых функций для кодирования длины списка

На Листинг 30 показано использование функции `len` в конструкторе `Cons` гарантирующем, что длина списка увеличивается на 1, а также в функции `append`, гарантирующая что длина результирующего списка равно сумме длин списков поданных на вход `xs` и `ys`

Сами неинтерпретируемые функции задаются как пара из имени и типа. В формат фреймворка они конвертируются при помощи `mkMeasure`, которая принимаем имя и тип реализующий контракт `IsType`

5.4 Резюме

В данной главе описана реализация проверки уточняющих типов на примере подмножества языка ocaml. Язык поддерживает

1) Ветвление и рекурсия
2) Функции высших порядков
3) Вывод предикатов
4) Параметрический полиморфизм и вывод типов
5) Полиморфные алгебраические типы данных
6) Сопоставление с образцом

В данной реализации показано применение представленного в работе фреймворка и его возможностей. Реализованная проверка типов показывает следующие важные аспекты

1) Применимость выбранной мощности предикатов для верификации свойств программного обеспечения
2) Предоставленные фреймворком ограничения полностью покрывают ограничения генерируемые при проверке типов
3) При реализации проверки типов не пришлось работать с предикатами самостоятельно, всю работу на себя берет фреймворк
4) Не было необходимости самостоятельно работать с SMT-решателями

ЗАКЛЮЧЕНИЕ

В рамках данной дипломной работы был спроектирован и разработан фреймворк для обобщенного вывода и проверки уточняющих типов, предназначенный для создания систем проверки типов в различных языках программирования. Основной целью было обеспечение универсальных механизмов для работы с уточняющими типами без необходимости каждый раз реализовывать сложные алгоритмы при добавлении такой системы в новый язык программирования.

В ходе работы над дипломным проектом были выполнены следующие задачи:

- Проведен обзор существующих языков программирования с уточняющими типами и основные аспекты их реализации
- Спроектирован фреймворк. Были разработаны требования и архитектура фреймворка, включающие контракты для работы с типами и предикатами, механизмы работы с окружением и системы генерации условий корректности. В рамках проектирования были определены необходимые компоненты и их взаимодействие, что обеспечило создание гибкого и расширяемого решения.
- На основе спроектированных решений реализован фреймворк. Основными компонентами которого являются
    1) Окружение с безопасным управлением областями видимости с учетом требований уточняющих типов
    2) Система контрактов для модификации предикатов в типах и преобразования в сигнатуры типов для SMT-решателей

3) Предикаты, поддерживающие логику QF_UFLIA (без кванторов, линейная арифметика, неинтерпретируемые функции). Автоматизация вывода предикатов
4) Механизм ограничений для автоматической генерации условий корректности на основе контрактов для типов и предикатов

– Продемонстрировано применение фреймворка. На основе разработанного фреймворка была реализована проверка уточняющих типов для подмножества языка OCaml, что подтвердило его применимость и эффективность. Реализованная система поддерживает:
    1) Ветвление и рекурсию
    2) Функции высших порядков
    3) Вывод предикатов
    4) Параметрический полиморфизм и вывод типов
    5) Полиморфные алгебраические типы данных
    6) Сопоставление с образцом

Основные преимущества разработанного фреймворка

– Безопасное управление областями видимости: Гарантируется корректная работа с переменными, избегание конфликтов имён и корректное расширение окружения.
– Автоматический вывод предикатов: Обеспечивается замена «заглушек» на переменные Хорна с последующим выводом уточнений через SMT-решатели.

– Упрощенная интеграция с SMT: Предоставляется автоматическая проверка условий корректности без необходимости прямого взаимодействия с решателями.
– Расширяемость: Фреймворк поддерживает добавление пользовательских типов и предикатов через четко определенные контракты (IsType, IsPred).
– Высокий уровень абстракции: Пользователь фреймворка не обязан работать с предикатами напрямую, если это не требуется для специфических модификаций.

Разработанный фреймворк для обобщенного вывода уточняющих типов является вкладом в область верификации программного обеспечения, предоставляя универсальный инструмент для реализации систем проверки типов в различных языках программирования. Его применимость была успешно продемонстрирована на примере подмножества языка OCaml, что подтверждает эффективность и жизнеспособность предложенного подхода.

Результаты данной работы могут быть использованы как исследователями в области типизированных языков программирования, так и разработчиками, стремящимися внедрить продвинутые системы типов в новые или существующие языки. Фреймворк обеспечивает высокий уровень абстракции и повторного использования кода, что значительно упрощает разработку систем проверки уточняющих типов.

Репозиторий с исходным кодом: https://github.com/AbsoluteNikola/free-foil-refinement-types