ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Василенко Елизавета Геннадьевна

АВТОМАТИЗАЦИЯ ДОКАЗАТЕЛЬСТВ В LIQUIDHASKELL С ПОМОЩЬЮ РЕЛЯЦИОННЫХ ТИПОВ

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
Д.Г. Штукенберг

Руководитель
канд.физ.-мат.наук
И.Е. Куралёнок

Санкт-Петербург 2021

Оглавление

Аннотация

Типы являются полезной абстракцией в языках программирования. Ограничения, накладываемые с помощью типов, позволяют программистам запрещать нежелательные поведения программ, тем самым способствуя их корректности. Для более тщательной верификации программ используют внешние инструменты, в том числе автоматизированное доказательство теорем о программах с помощью языков с зависимыми типами [7, 18, 6]. LiquidHaskell является реализацией типов уточнения для языка Haskell [21]. Приближаясь к выразительности зависимых типов, но обладая бóльшим автоматизмом, эта система позволяет верифицировать некоторые свойства программ на Haskell не прибегая к трудоёмким доказательствам на языках с зависимыми типами. Целью этой работы является автоматизация нового класса доказательств в LiquidHaskell за счёт расширения системы типов реляционными правилами типизации. Прошлые исследования реляционных типов показывают, что в число новых автоматически доказываемых утверждений войдёт ряд интересных свойств из области безопасности и защиты данных [20, 22, 13]. В данной работе формализуется алгоритмическая реляционная система типов, описывается её реализация и тестирование для Haskell и приводятся результаты сопоставления системы с предшественниками.

Ключевые слова: теория типов, реляционные типы, типы уточнения, двунаправленная проверка типов, статическая верификация, Haskell.

Types are a useful abstraction in programming languages. Restrictions imposed by types allow programmers to prohibit undesirable program behavior, thereby promoting program correctness. External tools are used to verify programs more thoroughly, including automated theorem prooving using languages with dependent types [7, 18, 6]. LiquidHaskell is an implementation of refinement types for Haskell [21]. Approaching the expressiveness of dependent types, but with more automatism, this system allows to verify some properties of programs in Haskell without resorting to time-consuming proofs in dependently typed languages. The goal of this work is to automate a new class of proofs in LiquidHaskell by extending the type system with relational typing rules. Past research on relational types shows that the new automatically provable assertions will include a number of interesting properties from the field of security and privacy [20, 22, 13]. This paper formalizes an algorithmic relational type system, describes its implementation and testing for Haskell, and compares the system to its predecessors.

Keywords: type theory, relational types, refinement types, bidirectional typechecking, static verification, Haskell.

Введение

Системы типов в языках программирования способствуют проверке корректности программ путём классификации выражений языка на этапе компиляции. Чем выразительнее система типов, тем точнее программист может задать допустимые аргументы функций, таким образом избавляясь от необходимости бросания исключений или другой динамической обработки некорректных входных данных. Аналогично, более точный тип возвращаемого значения может облегчить, а иногда и заменить тестирование поведения функции.

При всех достоинствах выразительных систем типов их использование часто создаёт и дополнительные трудности. В полиморфных системах типов компилятору могут требоваться такие подсказки от программиста, как аннотации типов [9, 8]. В языках с зависимыми типами может требоваться изменение структуры программы в угоду удобства верификации, что мешает использовать зависимые типы для языков общего назначения за редкими исключениями [6].

Поиском компромисса между надёжностью программ и трудоёмкостью их написания обусловлен научный интерес к дизайну выразительных систем типов, обладающих высокой степенью автоматизма, т.е. не требующих большого количества подсказок в виде аннотаций или изменённой структуры.

Одним путём автоматизации является делегация генерации подсказок SMT-решателям [4], проверяющим булевы формулы определённого вида на выполнимость. Применение такого подхода привело к появлению систем типов уточнения [24], в которых приведение типов сводится к решению булевых формул.

Наиболее развитой реализацией типов уточнения на данный момент является LiquidHaskell [21]. Опыт этой работы по расширению системы типов языка Haskell был перенесён и на другие языки. Серия аналогичных работ по реализации более выразительных систем типов, не требующих изменения синтаксиса выражений, включает типы уточнения для Python [16], TypeScript [26] и C [2].

Другим способом уменьшения числа необходимых подсказок являются

реляционные системы типов. Реляционные типы добавляют в систему типов возможность формулировать и доказывать утверждения о разнице между двумя термами в случаях когда это проще, чем рассуждать про термы независимо [5, 23].

Реляционные типы до сих пор не были реализованы ни в одном языке программирования общего назначения. Предыдущие исследования реляционных типов показывают их потенциал в качестве простого метода верификации алгоритмов из области безопасности и защиты данных.

Реализация реляционных типов в LiquidHaskell может поспособствовать упрощению поиска уязвимостей в чувствительных к безопасноти алгоритмах на языке Haskell. Опыт разработки такой системы так же может быть полезен при реализации реляционных систем типов для других языков программирования общего назначения.

1. Постановка задачи

Целью данной работы является уменьшение количества подсказок, требуемых для статической проверки свойств программ в LiquidHaskell, за счёт расширения системы типов LiquidHaskell реляционными правилами.

Решение поставленной задачи включает следующие подпункты:

- формализация алгоритмической реляционной системы типов на основе SMT-солвера для лямбда-исчисления с пользовательскими типами данных и параметрическим полиморфизмом;
- реализация полученной системы типов для языка Haskell;
- оценка полученной системы с точки зрения выразительности и количества требуемых подсказок.

Предикаты	p,q	$::=$	$x, y, z, \ldots$	*переменные*
		$\mid$	$\text{true}, \text{false}$	*булевы значения*
		$\mid$	$0, -1, 1, \ldots$	*числа*
		$\mid$	$p_1 \bowtie p_2$	*операторы лин. арифмет.*
		$\mid$	$p_1 \wedge p_2$	*конъюнкция*
		$\mid$	$p_1 \vee p_2$	*дизъюнкция*
		$\mid$	$\neg p$	*отрицание*
		$\mid$	$f(\overline{p})$	*неинт. функция*

Рис. 1: Синтаксис предикатов

2. Обзор литературы

2.1. Задача выполнимости формул в теориях

Задача выполнимости формул в теориях (англ. *SMT*) является обобщением задачи выполнимости булевых формул (англ. *SAT*). Обе задачи касаются существования значений переменных, обращающих булеву формулу в истину.

Многие задачи из области верификации сводятся к решению булевых формул, синтаксис которых расширен *неинтерпретируемыми функциональными символами*, что позволяет включать в логику пользовательские операции помимо стандартных логических.

Для проверки таких формул на выполнимость были разработаны SMT-решатели, в т.ч. Z3 [27], CVC [3], MathSat [12] и прочие. Все решатели имеют несколько отличающиеся характеристики, но существует единый стандарт [4].

2.2. Синтаксис предикатов

Синтаксис предикатов, который будет использоваться в данной работе, показан на Рис. 1. Выбор такого синтаксиса обусловлен в первую очередь гарантиями разрешимости и оценками времени работы, которые имеют SMT-решатели.

Система, представленная здесь, является фрагментом линейной арифметики с неинтерпретируемыми фунциями, ограниченной предикатами без

кванторов (QF-UFLIA) [4].

Она включает булевы литералы (true, false), целочисленные литералы $(0, 1, 2, \ldots)$, переменные, принимающие булевы и целочисленные значения $(x, y, z, \ldots)$, операторы линейной арифметики $(p_1 + p_2,\ p_1 \bmod p_2, \ldots)$ и логические операторы $(p_1 \wedge p_2,\ p_1 \vee p_2 \ldots)$.

Вместо операций линейной арифметики $(\bowtie)$ могут быть взяты операции любой другой разрешимой SMT-решателями логики.

Также можно использовать неинтерпретируемые функции с заданным для них набором аксиом, чтобы смоделировать любую другую теорию. При этом нужно соблюдать некоторую аккуратность, т.к. при этом может теряться гарантия разрешимости формулы. Например, даже при условии отсутствия кванторов в логике, с добавлением операции умножения помимо операций линейной арифметики задача выполнимости формул становится неразрешимой.

Тем не менее, в большинстве случаев использование неинтерпретируемых функций не составляет проблем. Их удобно использовать для проверки условий, касающихся таких функций в пользовательском домене как длина списка, глубина дерева, и прочих.

Примеры предикатов, принимаемых решателями, включают например $0 \leq x$, не содержащий неинтерпретируемых функций, или $0 \leq x \wedge x < length(x)$, где присутствует неинтерпретируемый символ $length$.

SMT-решатели часто поддерживают квантификацию формул, что тоже может быть использовано для верификации программ [17, 25]. Однако это не является стандартом, легко приводит к неразрешимости формул и сильно усложняет чтение ошибок, что является большим недостатком при разработке систем типов.

2.3. SMT-решатели и системы типов

При использовании SMT-решателей для проверки типизируемости программы, обычной практикой является сведение задачи типизации к проверке набора формул на *общезначимость* [10, 14].

Формула называется общезначимой, когда она верна при любых значени-

Базовый тип	b	$::=$	`int`	*числа*
Уточнения	r	$::=$	$\{\nu : p\}$	*условие*
Тип	t,s	$::=$	$b\{r\}$	*уточ. базовый тип*
		$\mid$	$x : t \to t$	*зав. функция*
Кайнд	k	$::=$	B	*базовый*
		$\mid$	$\star$	*универсальный*
Контекст	Γ	$::=$	$\emptyset$	*пустой*
		$\mid$	$\Gamma; x : t$	*связывание переменной*
Термы	e	$::=$	`c`	*константы*
		$\mid$	x	*переменные*
		$\mid$	`let` $x = e$ `in` e	*let-выражение*
		$\mid$	$\lambda x.\ e$	*функция*
		$\mid$	$e\ x$	*применение*
		$\mid$	$e : t$	*аннотация типа*

Рис. 2: Синтаксис системы с типами уточнения

ях входящих в неё переменных и неинтерпретируемых функций. Проверка на общезначимость просто сводится к проверке выполнимости отрицанием формулы.

Переменным в SMT-формулах могут быть присвоены неинтерпретируемые *сорта*, позволяющие проверять корректность применения тех или иных операторов к переменным. Эти сорта могут получены из типов переменных в том или ином языке программирования с помощью стандартных техник мономорфизации и дефункционализации.

Корректность предикатов с точки зрения сортов далее будет обозначаться как $\Gamma \vdash p : \Pi$ в некотором контексте Γ, содержащем используемые в p переменные и их типы.

2.4. Типы уточнения

В наиболее общем виде можно задать типы уточнения как расширение системы типов, добавляющее в синтаксис типы вида $t\{p\}$, где t – тип исходной системы, а p – предикат некоторой, обычно разрешимой, логики, истинность которого проверяется системой статически. Часто удобно считать, что p –

Общезначимость $\boxed{\Gamma \vdash c}$

$$\frac{\mathsf{SmtValid}(c)}{\emptyset \vdash c}\textsc{Ent-Emp} \qquad \frac{\Gamma \vdash \forall x\!:\!b.\ p \Rightarrow c}{\Gamma; x\!:\!b\{x\!:\!p\} \vdash c}\textsc{Ent-Ext}$$

Подтипизация $\boxed{\Gamma \vdash t_1 \prec\!: t_2}$

$$\frac{\Gamma \vdash \forall \nu_1\!:\!b.\ p_1 \Rightarrow p_2[\nu_2 := \nu_1]}{\Gamma \vdash b\{\nu_1\!:\!p_1\} \prec\!: b\{\nu_2\!:\!p_2\}}\textsc{Sub-Base}$$

$$\frac{\Gamma \vdash s_2 \prec\!: s_1 \qquad \Gamma; x_2\!:\!s_2 \vdash t_1[x_1 := x_2] \prec\!: t_2}{\Gamma \vdash x_1\!:\!s_1 \to t_1 \prec\!: x_2\!:\!s_2 \to t_2}\textsc{Sub-Fun}$$

Рис. 3: Подтипизация, сводящаяся к общезначимости

произвольное булево выражение языка программирования. Полный синтаксис подобной системы для лямбда-исчисления представлен на Рис. 2.

Помимо таких систем часто встречаются типы уточнения в упрощённых формах, не позволяющих уточнить тип произвольным p, но предоставляющих набор конкретных предикатов, истинность которых система умеет проверять. В числе таких например типы целочисленных интервалов в языке Ада [19]. Их можно было бы выразить в более общем синтаксисе как $\texttt{int}\{x\!:\!-1 \leq x \wedge x \leq 1\}$.

Системы типов с уточнениями часто формулируют в двунаправленной манере, позволяющей автоматизировать проверку типов с точностью до изредка необходимых аннотаций типов. Для двунаправленных систем оказываются полезны три вида суждений – подтипизация, вывод типа и проверка типа. Они представлены на Рис. 3, 4 и 5 для иллюстрации. Суждение $\Gamma \vdash t : k$ означает корректное построение типа t в контексте Γ.

Обращение к SMT-решателю инкорпорировано в систему с помощью отношения общезначимости предиката в контексте (Рис. 3).

LiquidHaskell является системой типов уточнения для языка Haskell. Система реализована в двунаправленной манере. Поддержка синтаксиса Haskell

Вывод типа $\boxed{\Gamma \vdash e \Rightarrow t}$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x \Rightarrow t}\textsc{Syn-Var} \qquad \frac{\mathsf{prim}(\mathsf{c}) = t}{\Gamma \vdash \mathsf{c} \Rightarrow t}\textsc{Syn-Con}$$

$$\frac{\Gamma \vdash t : k \quad \Gamma \vdash e \Leftarrow t}{\Gamma \vdash e{:}t \Rightarrow t}\textsc{Syn-Ann} \qquad \frac{\Gamma \vdash e \Rightarrow x{:}s \to t \quad \Gamma \vdash y \Leftarrow s}{\Gamma \vdash e\ y \Rightarrow t[x := y]}\textsc{Syn-App}$$

Рис. 4: Двунаправленная типизация, вывод типа

Type Checking $\boxed{\Gamma \vdash e \Leftarrow t}$

$$\frac{\Gamma \vdash e \Rightarrow s \qquad \Gamma \vdash s \prec: t}{\Gamma \vdash e \Leftarrow t}\textsc{Chk-Syn} \qquad \frac{\Gamma; x{:}t_1 \vdash e \Leftarrow t_2}{\Gamma \vdash \lambda x.\ e \Leftarrow x{:}t_1 \to t_2}\textsc{Chk-Lam}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow t_1 \qquad \Gamma; x{:}t_1 \vdash e_2 \Leftarrow t_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Leftarrow t_2}\textsc{Chk-Let}$$

Рис. 5: Двунаправленная типизация, проверка типа

включает такие тонкости, как рекурсия, паттерн-матчинг, классы типов, уточнение конструкторов алгебраических типов данных и обобщённых алгебраических типов данных, а так же абстрагирование по уточнению.

2.5. Реляционные типы

Реляционные системы типов в наиболее общем виде были описаны в работах [23, 1]. Описанная в них система получила название RHOL. Утверждение типизации в этой системе имеет вид $\Gamma \vdash e_1 \sim e_2 : t_1 \sim t_2 \mid \phi$, где e_1 и e_2 – термы языка, t_1 и t_2 – типы, а ϕ – предикат, определённый на переменных контекста Γ и e_1 и e_2.

Так же, как и типы уточнения, реляционные типы позволяют менее общие формулировки [22, 5]. В отличие от систем типов уточнения, пока что нет работ по алгоритмизации реляционных систем типов в наиболее общем случае.

Основная сложность в создании детерминированного алгоритма проверки типа состоит в том, что реляционные системы подразумевают несколько правил вывода в зависимости от вида термов e_1 и e_2, в том числе:

- *синхронные правила*, применимые для двух термов одинаковой структуры;
- асинхронные правила, не имеющие ограничений на вид одного из термов;
- структурные правила, не имеющие никаких ограничений на вид термов.

Предикат	p, q	$::=$	$\mathtt{r}_1, \mathtt{r}_2$	*рел. синонимы*
		$\mid$	$\cdots$	*см. Рис. 1*
Квантифицированный предикат	ϕ,ψ	$::=$	p	*без квантификации*
		$\mid$	$\forall x\ y.\ p \Rightarrow \phi$	*неинст. предикат*
Реляционный контекст	Ψ	$::=$	$\varnothing$	*пустой контекст*
		$\mid$	$\Psi; p$	*предикат*
		$\mid$	$\Psi; \langle e_1,\ e_2,\ \phi \rangle$	*неинст. пр-т*

Рис. 6: Синтаксис реляционных предикатов и контекстов

3. Предлагаемая алгоритмическая реляционная система

3.1. Синтаксис

Тип в реляционной системе имеет вид $\tau_1 \sim \tau_2 \mid \phi$, где τ_1 и τ_2 – типы уточнения. Синтаксис предиката ϕ приведён на Рис. 6. Выделенные константы $\mathtt{r}_1$ и $\mathtt{r}_2$ используются в предикате в качестве синонимов для компонент типизируемой пары.

В дополнение к контексту типизации Γ вводится реляционный контекст Ψ, содержащий предикаты двух видов. Предикаты без кванторов уже готовы к обработке SMT-солвером, а предикаты с кванторами отложены до момента, когда их можно будет инстанциировать, заменив в них связанные переменные на конкретные выражения.

Введение квантора всеобщности, всегда сопровождаемого импликацией, помогает задать единообразную структуру предикатов, аналогичную структуре стрелочных типов. Полные условия правильного построения (англ. *well-formedness*) предикатов в составе реляционных типов представлены на Рис. 7[1].

Правило WF-Base связывает два базовых типа и типизируемый предикат без кванторов в контексте Γ. Двум стрелочным типам соответству-

[1]Потерю имевшейся в RHOL возможности сопоставлять типы с разным количеством стрелок в составе корректного реляционного типа при необходимости можно компенсировать с помощью замены меньшего типа τ на функциональный $x{:}\mathtt{unit} \to \tau$ с фиктивным аргументом необходимое количество раз.

Правильно построенный реляционный тип $\boxed{\Gamma \vdash t \sim s \mid \phi}$

$$\frac{\Gamma \vdash t_1 : B \quad \Gamma \vdash t_2 : B \quad \Gamma; x_1{:}t_1; x_2{:}t_2 \vdash p[\mathtt{r}_1 := x_1][\mathtt{r}_2 := x_2] : \Pi}{\Gamma \vdash t_1 \sim t_2 \mid p}\text{WF-Base}$$

$$\frac{\Gamma; x_1{:}s_1; x_2{:}s_2 \vdash p : \Pi \quad \Gamma; x_1{:}s_1; x_2{:}s_2 \vdash t_1 \sim t_2 \mid \phi}{\Gamma \vdash x_1{:}s_1 \to t_1 \sim y{:}s_2 \to t_2 \mid \forall x_1\, x_2.\ p \Rightarrow \phi[\mathtt{r}_1 := \mathtt{r}_1\ x_1][\mathtt{r}_2 := \mathtt{r}_2\ x_2]}\text{WF-Fun}$$

Рис. 7: Правильно построенный реляционный тип

ет квантифицированный предикат, связывающий две новые переменные. В терминах этих переменных выражается реляционное p предположение об аргументах функции x_1 и x_2.

Ниже приводятся примеры типов, удовлетворяющих и не удовлетворяющих правильному построению, и объясняется их семантика.

Типы, удовлетворяющие правильному построению

$$\varnothing \vdash \mathtt{int} \sim \mathtt{int} \mid \mathtt{r}_1 < \mathtt{r}_2 \tag{1}$$

$$x{:}\mathtt{int} \vdash \mathtt{int}\{\nu{:}\nu = x\} \sim \mathtt{int}\{\nu{:}\nu \neq x\} \mid \text{true} \tag{2}$$

$$\varnothing \vdash x_1{:}\mathtt{int} \to \mathtt{int} \sim x_2{:}\mathtt{int} \to \mathtt{int} \mid \forall x_1\, x_2.\ x_1 < x_2 \Rightarrow \mathtt{r}_1\ x_1 < \mathtt{r}_2\ x_2 \tag{3}$$

Тип (1) содержит пары целых чисел, первое из которых меньше второго. Типы с тривиальным предикатом, такие как (2), эквиваленты типам произведения. Тип (3) представляет пару функций, чьи значения связаны неравенством, когда им связана пара подставленных аргументов.

Типы, не удовлетворяющие правильному построению

$$\varnothing \vdash x_1{:}\mathtt{int} \to \mathtt{int} \sim x_2{:}\mathtt{int} \to \mathtt{int} \mid \text{true} \tag{4}$$

$$\varnothing \vdash x_1{:}\mathtt{int} \to \mathtt{int} \sim x_2{:}\mathtt{int} \to \mathtt{int} \mid \forall x_1\, x_2.\ \text{true} \Rightarrow \mathtt{r}_1 = \mathtt{r}_2 \tag{5}$$

$$\varnothing \vdash x_1{:}\mathtt{int} \to \mathtt{int} \sim x_2{:}\mathtt{int} \to \mathtt{int} \mid \forall x_1\, x_2.\ x_1 < x_2 \Rightarrow \mathtt{r}_1\ x_2 < \mathtt{r}_2\ x_1 \tag{6}$$

$$\varnothing \vdash \mathtt{int} \sim x{:}\mathtt{int} \to \mathtt{int}\{\nu{:}0 \leq \nu\} \mid \forall x_1\, x_2.\ \text{true} \Rightarrow 0 \leq \mathtt{r}_2\ \mathtt{r}_1 \tag{7}$$

Тип (4) построен неверно, т.к. правило WF-Fun требует появления квантора и импликации в предикате, связывающем две функции. Вместо true в качестве тривиального предиката для функций подойдёт $\forall x_1\, x_2.\ \text{true} \Rightarrow \text{true}$.

Не смотря на то, что в предикатах допустимо проверять равенство функций (как функциональных символов), тип (5) также не корректен. По правилу WF-Fun связанные квантором переменные должны фигурировать в предикате в качестве аргументов при всех вхождениях реляционных констант. Таким образом, константы $\mathtt{r}_1$ и $\mathtt{r}_2$ могут участвовать в предикатах только в полностью применённом виде. Более того, аргументом $\mathtt{r}_1$ обязательно должна быть первая из связанных переменных, а аргументом $\mathtt{r}_2$ – вторая. Тип (6) нарушает это условие.

Из этих ограничений следует невозможность анализа применения одного из сопоставляемых выражений к другому. Например, в (7) невозможно выразить то, что такой вызов дал бы неотрицательный результат.

3.2. Декларативная типизация

В этом разделе обсуждаются декларативные правила вывода суждений о принадлежности реляционному типу, позволяющих типизировать термы в двунаправленном стиле. Отношение *вывода* типа $\Gamma \mid \Psi \vdash e_1 \sim e_2 \Rightarrow t_1 \sim t_2 \mid \phi$ утверждает, что в контекстах Γ и Ψ типы t_1 и t_2 были получены как наиболее общие (англ. *principal*) для термов e_1 и e_2 соответственно, и может быть порождён истинный предикат ϕ. Для каждой пары t_1, t_2 существует единственная тройка e_1, e_2, ϕ.

Аналогично, отношение *проверки* типа $\Gamma \mid \Psi \vdash e \sim d \Leftarrow t \sim s \mid \phi$ утверждает, что в контекстах Γ и Ψ типы t и s могут быть присвоены термам e и d соответственно, и предикат ϕ выполняется. Для пары t_1, t_2 могут подходить несколько троек e_1, e_2, ϕ, удовлетворяющих отношению проверки типа.

Для перехода между режимами вывода и проверки вводится отношение подтипизации на реляционных типах, показанное на Рис. 8. Суждение $\Gamma \mid \Psi \vdash t_1 \sim t_2 \mid \psi \prec: \phi$ означает, что в контекстах Γ и Ψ из квантифицированного предиката ψ следует квантифицированный предикат ϕ в предположении, что предикаты являются отношениями на одной и той же паре типов t_1

Подтипизация реляционных типов $\boxed{\Gamma \mid \Psi \vdash s \sim t \mid \psi \prec\!: \phi}$

$$\frac{\Gamma_\Psi; \mathtt{r}_1 : s; \mathtt{r}_2 : t \vdash q \Rightarrow p}{\Gamma \mid \Psi \vdash s \sim t \mid q \prec\!: p}\textsc{Sub-Base}$$

$$\frac{\begin{array}{c}\phi_s \doteq \ \phi[\mathtt{r}_1 := \mathtt{r}_1\ x_1][\mathtt{r}_2 := \mathtt{r}_2\ x_2] \\ \psi_s \doteq \ \psi[\mathtt{r}_1 := \mathtt{r}_1\ x_1][\mathtt{r}_2 := \mathtt{r}_2\ x_2] \\ \Gamma_\Psi; x_1 : s_1; x_2 : s_2 \vdash p \Rightarrow q \quad \Gamma; x_1 : s_1; x_2 : s_2 \mid \Psi; p \vdash t_1 \sim t_2 \mid \psi \prec\!: \phi\end{array}}{\Gamma \mid \Psi \vdash x_1 : s_1 \to t_1 \sim x_2 : s_2 \to t_2 \mid \forall x_1\, x_2.\ q \Rightarrow \psi_s \prec\!: \forall x_1\, x_2.\ p \Rightarrow \phi_s}\textsc{Sub-Fun}$$

$$\Gamma_\Psi \doteq \Gamma; x : \mathtt{unit}\{\nu : \bigwedge_{p \in \Psi} p\}$$

Рис. 8: Подтипизация реляционных типов

и t_2. Правила подтипизации сводят проверку тождественной истинности импликации двух предикатов высшего порядка к проверке набора импликаций первого порядка. Это делает задачу разрешимой с помощью SMT-решателя.

В системе типов с уточнениями вводилось понятие общезначимости предиката p в контексте Γ. Для установления общезначимости с учётом второго контекста Ψ вводится новое обозначение Γ_Ψ. Переменная x должна быть свободной в Γ и p. Контекст Γ_Ψ, содержащий конъюнкцию всех предикатов без кванторов из Ψ может быть использован для проверки общезначимости p с помощью SMT-решателя.

На Рис. (9) представлены правила вывода реляционного типа. Суждение имеет форму $\Gamma \mid \Psi \vdash e_1 \sim e_2 \Rightarrow t_1 \sim t_2 \mid \phi$. Суждение правильно построено, когда свободные переменные предикатов из Ψ и выражений e_1 и e_2 включены в Γ. Выражения e_1 и e_2 удовлетворяют синтаксису на Рис. **??** и находятся в административной нормальной форме (англ. *administrative normal form, A-normal form*). Реляционный тип $t_1 \sim t_2 \mid p$ должен быть правильно построен в Γ.

Аналогичное суждение проверки реляционного типа представлено на Рис. (10).

Вывод реляционного типа $\boxed{\Gamma \mid \Psi \vdash e_1 \sim e_2 \Rightarrow t_1 \sim t_2 \mid \phi}$

$$\frac{\Gamma \vdash e_1 \Rightarrow t_1 \quad \Gamma \vdash e_2 \Rightarrow t_2 \quad e_1, e_2 \text{ – переменные или константы}}{\Gamma \mid \Psi \vdash e_1 \sim e_2 \Rightarrow t_1 \sim t_2 \mid \mathsf{inst}(\Psi,\ e_1,\ e_2)}\textsc{RelSyn-Unary}$$

$$\frac{\mathsf{inst}(\Psi,\ e_1\ x_1,\ e_2\ x_2) \doteq \forall v_1\, v_2.\ p \Rightarrow \phi \quad \Gamma \vdash e_1 \Leftarrow x_1 : s_1 \to t_1 \quad \Gamma \vdash e_2 \Leftarrow x_2 : s_2 \to t_2 \quad \Gamma \mid \Psi \vdash x_1 \sim x_2 \Rightarrow s_1 \sim s_2 \mid q}{\Gamma \mid \Psi \vdash e_1\ x_1 \sim e_2\ x_2 \Rightarrow t_1 \sim t_2 \mid \forall v_1\, v_2.\ p \wedge q[\mathtt{r}_1 := v_1][\mathtt{r}_2 := v_2] \Rightarrow \phi}\textsc{RelSyn-App}$$

inst	:	$(\Psi \times E \times E) \to \Pi$
$\mathsf{inst}(\Psi,\ f_1\ x_1,\ f_2\ x_2)$ $\quad \mid\ \langle f_1,\ f_2,\ \forall v_1\, v_2.\ p \Rightarrow \phi \rangle \in \Psi$	$\doteq$	$(p \Rightarrow \phi)[v_1 := x_1][v_2 := x_2]$
$\mathsf{inst}(\Psi,\ x_1,\ x_2)$ $\quad \mid\ \langle x_1,\ x_2,\ p \rangle \in \Psi$	$\doteq$	p
$\mathsf{inst}(\Psi,\ e_1,\ e_2)$	$\doteq$	true

Рис. 9: Двунаправленная типизация, вывод реляционного типа

На Рис. 10 показаны правила проверки реляционного типа.

Relational Type Checking $\boxed{\Gamma \mid \Psi \vdash e \sim d \Leftarrow t \sim s \mid \phi}$

$$\frac{\Gamma \mid \Psi \vdash e \sim d \Rightarrow s_1 \sim s_2 \mid \psi \quad \Gamma; x : t_\Psi \vdash s_1 \prec: t_1 \quad \Gamma; x : t_\Psi \vdash s_2 \prec: t_2 \quad \Gamma \mid \Psi \vdash s_1 \sim s_2 \mid \psi \prec: \phi}{\Gamma \mid \Psi \vdash e \sim d \Leftarrow t_1 \sim t_2 \mid \phi}\textsc{RelChk-Syn}$$

$$\frac{\phi_s \doteq \phi[\mathtt{r}_1 := \mathtt{r}_1\ v_1][\mathtt{r}_2 := \mathtt{r}_2\ v_2] \quad \Gamma; x_1 : s_1; x_2 : s_2 \mid \Psi; p \vdash e_1 \sim e_1 \Leftarrow t_1[v_1 := x_1] \sim t_2[v_2 := x_2] \mid \phi}{\Gamma \mid \Psi \vdash \lambda x_1.\ e_1 \sim \lambda x_2.\ e_2 \Leftarrow v_1 : s_1 \to t_1 \sim v_1 : s_2 \to t_2 \mid \forall v_1\, v_2.\ p \Rightarrow \phi_s}\textsc{RelChk-Lam}$$

$$\frac{\phi_s \doteq \phi[\mathtt{r}_1 := \mathtt{r}_1\ x_1][\mathtt{r}_2 := \mathtt{r}_2\ x_2] \quad \Gamma_r \doteq \Gamma; f_1 : (x_1 : s_1 \to t_1); f_2 : (x_2 : s_2 \to t_2); x_1 : s_1; x_2 : s_2 \quad \Gamma_r \mid \Psi; p; \langle f_1,\ f_2,\ \forall x_1\, x_2.\ p \Rightarrow \phi_s \rangle \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi}{\Gamma \mid \Psi \vdash \mathtt{rec}\ f_1 = \lambda x_1.\ e_1 \sim \mathtt{rec}\ f_2 = \lambda x_2.\ e_2 \Leftarrow t_1 \sim t_2 \mid \forall x_1\, x_2.\ p \Rightarrow \phi_s}\textsc{RelChk-Rec}$$

$$\frac{\Gamma \mid \Psi \vdash d_1 \sim d_2 \Rightarrow s_1 \sim s_2 \mid \psi \quad \Gamma; x_1 : s_1; x_2 : s_2 \mid \Psi; \psi[\mathtt{r}_1 := x_1][\mathtt{r}_2 := x_2] \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi}{\Gamma \mid \Psi \vdash \mathtt{let}\ x_1 = d_1\ \mathtt{in}\ e_1 \sim \mathtt{let}\ x_2 = d_2\ \mathtt{in}\ e_2 \Leftarrow t_1 \sim t_2 \mid \phi}\textsc{RelChk-Let}$$

$$\frac{\begin{array}{c}\psi_s \doteq \psi[\mathtt{r}_1 := x_1][\mathtt{r}_2 := x_2] \\ \Gamma \vdash s_1 \prec: \mathtt{bool} \quad \Gamma \vdash s_2 \prec: \mathtt{bool} \quad \Gamma \mid \Psi \vdash x_1 \sim x_2 \Rightarrow s_1 \sim s_2 \mid \psi \\ \Gamma \mid \Psi; \psi_s; x_1; x_2 \vdash d_1 \sim d_2 \Leftarrow t_1 \sim t_2 \mid \phi \\ \Gamma \mid \Psi; \psi_s; x_1; \neg x_2 \vdash d_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi \\ \Gamma \mid \Psi; \psi_s; \neg x_1; \neg x_2 \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi \\ \Gamma \mid \Psi; \psi_s; \neg x_1; x_2 \vdash d_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi\end{array}}{\Gamma \mid \Psi \vdash \mathtt{if}\ x_1\ \mathtt{then}\ d_1\ \mathtt{else}\ e_1 \sim \mathtt{if}\ x_2\ \mathtt{then}\ d_2\ \mathtt{else}\ e_2 \Leftarrow t_1 \sim t_2 \mid \phi}\textsc{RelChk-If}$$

$$\frac{\Gamma \vdash x_1 \Leftarrow \mathtt{bool} \quad \Gamma \mid \Psi; x_1 \vdash d_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi \quad \Gamma \mid \Psi; \neg x_1 \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi}{\Gamma \mid \Psi \vdash \mathtt{if}\ x_1\ \mathtt{then}\ d_1\ \mathtt{else}\ e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi}\textsc{RelChk-IF-L}$$

$$\frac{\Gamma \vdash x_2 \Leftarrow \mathtt{bool} \quad \Gamma \mid \Psi; x_2 \vdash e_1 \sim d_2 \Leftarrow t_1 \sim t_2 \mid \phi \quad \Gamma \mid \Psi; \neg x_2 \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi}{\Gamma \mid \Psi \vdash e_1 \sim \mathtt{if}\ x_2\ \mathtt{then}\ d_2\ \mathtt{else}\ e_2 \Leftarrow t_1 \sim t_2 \mid \phi}\textsc{RelChk-IF-R}$$

Рис. 10: Двунаправленная типизация, проверка реляционного типа

```
data Expr b
  = Var   Id
  | Lit   Literal
  | App   (Expr b) (Arg b)
  | Lam   b (Expr b)
  | Let   (Bind b) (Expr b)
  | Case  (Expr b) b Type [Alt b]
  | Cast  (Expr b) Coercion
  | Tick  (Tickish Id) (Expr b)
  | Type  Type
  | Coercion Coercion
```

Рис. 11: Синтаксис выражений в GHC Core Language, Glasgow Haskell Compiler, Version 8.10.2

4. Реализация реляционных типов для Haskell

4.1. Поддерживаемый синтаксис

LiquidHaskell является плагином к компилятору GHC [11], что позволяет ему оперировать выражениями промежуточного представления Haskell – GHC Core Language. Синтаксис выражений в GHC 8.10.2 представлен на Рис. 11.

Реляционные правила типизации были реализованы в соответствии с декларативной системой из Главы 3 с некоторыми расширениями.

Были реализованы правила проверки и вывода типа для типовых абстракций и аппликаций. Эти правила не влияют на реляционный предикат.

Была добавлена поддержка для объявлений пользовательских типов данных и паттерн-матчинга на них. Эти правила являются обобщением правил RelChk-If, RelChk-If-L и RelChk-If-R с Рис. 10, являющихся частным случаем паттерн-матчинга по булевым значениям.

Таким образом, были поддержаны все виды выражений языка кроме `Cast` и `Coercion`.

4.2. Тестирование

Для тестирования корректности реализации были написаны порядка 100 примеров программ, которые должны приниматься или отвергаться реля-

Время компиляции тестовых примеров			
Название примера	Число строк кода	Время компиляции (сек)	Количество булевых формул на выходе
Монотонность фибоначчи	18	2.8	83
Быстрое возведение в степень	24	4.6	417
Map fusion	158	12.1	276
Сортировка вставками	165	6.6	154
Градиентный спуск	390	16.2	478

Рис. 12: Время компиляции тестовых примеров на процессоре 2,8 ГГц Dual-Core Intel Core i5 с оперативной памятью 8 Гб 1600 МГц DDR3

ционной системой. Примеры были частично взяты из прошлых статей и частично придуманы самостоятельно.

Бóльшая доля примеров может быть найдена в репозитории проекта[2].

Был направлен пул-реквест в ecсrypto, криптографическую библиотеку на языке Haskell[3]. Пул-реквест не был принят в основную ветку проекта, т.к. время компиляции проекта сильно увеличилось с добавлением зависимости от LiquidHaskell.

На Рис. 12 приведена таблица времени работы на разных примерах.

[2] `https://github.com/oquechy/liquidhaskell/tree/develop/tests/relational`
[3] `https://github.com/mfourne/eccrypto/tree/liquidhaskell`

5. Оценка результатов

В этой главе даётся сравнение полученных результатов с работами предшественников. Интерес представляют три родственные системы типов:

- LiquidHaskell [15, 21], алгоритмическая система типов уточнения для Haskell;
- RelCost [22], алгоритмическая реляционная система типов уточнения для ML-подобного языка,
- RHOL [23], декларативная реляционная система типов для лямбда-исчисления с условными выражениями и рекурсией.

5.1. LiquidHaskell

Реляционная система позволяет сводить проверку типов к решению SMT-формул несколько другим образом, чем не реляционные правила LiquidHaskell. Одним из свойств системы благодаря правилу RelSyn-Unary является то, что типизация в реляционной системе работает не хуже, чем типизация пар в LiquidHaskell, в том смысле, что реляционной системе не понадобятся подсказки для типизации двух выражений, когда без них обходится исходная система.

На нескольких примерах, которые могут быть доказаны с использованием индукции по реляционному свойству, было показано, что реляционная система может доказать без подсказок или с меньшем их количеством ряд новых свойств. Одним из таких является свойство о минимальности количества операций сравнения, производимых сортировкой вставками на сортированном списке.

Помимо индукции, преимуществом системы является использование реляционных сигнатур функций в качестве подсказок для типизации выражений, где эти функции были применены.

Дальнейшее устранение подсказок может быть сделано с помощью переменных Хорна. Переменные Хорна в SMT-решателях позволяют проверять

```
data Tick a = Tick { tcost :: Int, tval :: a }

le :: Int → [Int] → Bool
le _ []       = ⊤
le y (x : xs) = y <= x && le y xs

sorted :: [Int] → Bool
sorted []       = ⊤
sorted (x : xs) = sorted xs && le x xs

isort :: [Int] → Tick [Int]
isort = ...

isort ~ isort :: xs1:[Int] → Tick [Int] ~ xs2:[Int] → Tick [Int]
               | sorted xs1 && len xs1 = len xs2 =>
                    tcost (r1 xs1) <= tcost (r2 xs2)
```

Рис. 13: Реляционная сигнатура сортировки вставками в LiquidHaskell использует определения пользовательских предикатов `sorted` и `le` и типа данных `Tick`

на общезначимость формулы с экзистенциальными переменными. В качестве экзистенциальных переменных как раз могут быть некоторые подсказки, если знать, в каких местах они будут находиться.

5.2. RelCost

Система RelCost предоставляет собственный ML-подобный язык программирования, расширенный примитивами для рассуждений о сравнительном анализе стоимости исполнения двух программ. Типы уточнения в этой системе представлены представлены в ограниченном виде:

- $\mathrm{list}[\mathrm{n}]^{\alpha}\ \tau$ – тип пары списков длины n из элементов типа τ, отличающихся в α позициях;
- $\Box\tau$ – тип пары из равных друг другу элементов типа τ;
- $\sigma \xrightarrow{\mathrm{diff(n)}} \tau$ – тип пары функций из σ в τ, время исполнения которых отличается на n;

- $\sigma \xrightarrow{\text{exec(l,r)}} \tau$ – тип функции из σ в τ, время исполнения которой лежит в отрезке от l до r.

Эта система статически анализирует разницу во времени исполнения двух функций опираясь на предположения о стоимости вызова примитивных функций. Например, предположив, что операция умножения имеет тип $\texttt{int} \xrightarrow{\text{exec(1,1)}} \texttt{int}$, можно установить, что алгоритм двоичного возведения в степень будет работать одинаковое время при возведении любых оснований в одну и ту же степень.

Реляционная система LiquidHaskell позволяет проводить аналогичные рассуждения про анализ стоимости. Типы $\text{list[n]}^{\alpha}\ \tau$ и $\Box\tau$ симулируются с помощью предикатов. Специальные правила проверки типа, относящиеся к стрелочным типам, содержали вычисления разницы стоимости. Вместо добавления новых правил типизации для каждого эффекта, его обработку в LiquidHaskell можно задать с помощью пользовательской монады.

На Рис. 13 показано, как с помощью дополнительных конструкций в LiquidHaskell можно сформулировать свойство о том, что сортировка вставками делает минимальное количество сравнений на сортированном списке. Для реализации сортировки в стиле RelCost следует переопределить все используемые в алгоритме функции как монадические стрелки. При этом свойство о сравнительной стоимости доказывается без подсказок.

Таким образом, в отличие от алгоритмических систем-предшественников, реляционная LiquidHaskell более универсальна, т.к. её можно переиспользовать для рассуждений о разных видах реляционных эффектов и предикатов.

5.3. RHOL

Добавление реляционных типов в LiquidHaskell было основано на синхронных правилах RHOL, обобщённых таким образом, чтобы избегать потери информации. Асинхронные и структурные правила при этом практически исключены из системы для сохранения детерминизма. На Рис. 14 видны отличия синхронных правил этой работы от правил RHOL на примере RelChk-If. По сравнению с If-RHOL, это правило не препятствует дальнейшей ти-

Relational Typing Judgement $\boxed{\Gamma \mid \Psi \vdash e \sim d \Leftarrow t \sim s \mid \phi}$

$$\frac{\begin{array}{c}\psi_s \doteq \psi[\mathtt{r}_1 := x_1][\mathtt{r}_2 := x_2] \\ \Gamma \vdash s_1 \prec: \mathtt{bool} \quad \Gamma \vdash s_2 \prec: \mathtt{bool} \quad \Gamma \mid \Psi \vdash x_1 \sim x_2 \Rightarrow s_1 \sim s_2 \mid \psi \\ \Gamma \mid \Psi; \psi_s; x_1; x_2 \vdash d_1 \sim d_2 \Leftarrow t_1 \sim t_2 \mid \phi \\ \Gamma \mid \Psi; \psi_s; x_1; \neg x_2 \vdash d_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi \\ \Gamma \mid \Psi; \psi_s; \neg x_1; \neg x_2 \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi \\ \Gamma \mid \Psi; \psi_s; \neg x_1; x_2 \vdash d_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi\end{array}}{\Gamma \mid \Psi \vdash \mathtt{if}\ x_1\ \mathtt{then}\ d_1\ \mathtt{else}\ e_1 \sim \mathtt{if}\ x_2\ \mathtt{then}\ d_2\ \mathtt{else}\ e_2 \Leftarrow t_1 \sim t_2 \mid \phi}\ \textsc{RelChk-If}$$

$$\frac{\begin{array}{c}\Gamma_\Psi \vdash x_1 \Leftrightarrow x_2 \qquad \Gamma \mid \Psi; x_1; x_2 \vdash d_1 \sim d_2 : t_1 \sim t_2 \mid \phi \\ \Gamma \mid \Psi; \neg x_1; \neg x_2 \vdash e_1 \sim e_2 : t_1 \sim t_2 \mid \phi\end{array}}{\Gamma \mid \Psi \vdash \mathtt{if}\ x_1\ \mathtt{then}\ d_1\ \mathtt{else}\ e_1 \sim \mathtt{if}\ x_2\ \mathtt{then}\ d_2\ \mathtt{else}\ e_2 \Leftarrow t_1 \sim t_2 \mid \phi}\ \textsc{If-RHOL}$$

Рис. 14: Сравнение синхронных правил для условных выражений

пизации при невозможности статически вывести эквивалентность условий в условных выражениях. Аналогичным обобщениям подверглись правила RelChk-Let и RelSyn-App.

Устранение недетерминизма в асинхронных правилах не было проделано в рамках этой работы. Без них остаётся неясным, насколько предлагаемая система полна по отношению к исходной RHOL, в частности, можно ли доказать любое выводимое в RHOL утверждение с помощью подсказок.

5.4. Выводы и результаты по главе

Суммируя, данная работа занимает нишу алгоритмических реляционных систем для чистых функциональных языков высшего порядка. С точки зрения теории, было бы интересно оценить систему на полноту по отношению к RHOL и описать точное местоположение необходимых системе подсказок.

Заключение

В рамках данной работы были достигнуты следующие результаты:

- сформулирована алгоритмическая реляционная система типов с типами уточнения;
- система реализована на на языке Haskell и протестирована;
- дана оценка выразительности и автоматизма системы по сравнению с родственными работами.

Полученная формализация реляционной системы может быть переиспользована для реализации реляционных типов поверх других функциональных языков.

В случае с LiquidHaskell, все утверждения, которые можно доказать при помощи реляционных правил, могут быть выведены и без их использования. Преимущество реляционных правил в меньшем количестве требуемых подсказок. Возможность улучшения системы для дальнейшей минимизации количества необходимых подсказок является вопросом последующих исследований. Одним возможным путём может быть использование хорновских переменных для синтеза предикатов.

Другим направлением дальнейшей работы может быть улучшение взаимодействия реляционной и не реляционной систем. Например, поддержка использования утверждений, доказанных реляционно, в не реляционных рассуждениях.

Реализация системы доступна на GitHub вместе с примерами принимаемых и отвергаемых программ: `https://github.com/oquechy/liquidhaskell`.

Список литературы

[1] Aguirre Galindo Alejandro. Relational Logics for Higher-Order Effectful Programs : Ph. D. thesis / Alejandro Aguirre Galindo ; Universidad Politécnica de Madrid. — 2021.

[2] Sammler Michael, Lepigre Rodolphe, Krebbers Robbert et al. Artifact and Appendix of "RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types". — 2021. — Mar. — Access mode: https://doi.org/10.5281/zenodo.4649822.

[3] Barrett Clark, Berezin Sergey. CVC Lite: A New Implementation of the Cooperating Validity Checker // Computer Aided Verification / Ed. by Rajeev Alur, Doron A. Peled. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2004. — P. 515–518.

[4] Barrett C., Stump Aaron, Tinelli C. The SMT-LIB Standard Version 2.0. — 2010.

[5] Bidirectional Type Checking for Relational Properties / Ezgi Çiçek, Weihao Qu, Gilles Barthe et al. // Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI 2019. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 533–547. — Access mode: https://doi.org/10.1145/3314221.3314603.

[6] Brady Edwin C. Idris, a general-purpose dependently typed programming language: Design and implementation // Journal of Functional Programming. — 2013. — Vol. 23. — P. 552 – 593.

[7] Casinghino Chris, Sjöberg Vilhelm, Weirich Stephanie. Combining Proofs and Programs in a Dependently Typed Language // Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 33–45. — Access mode: https://doi.org/10.1145/2535838.2535883.

[8] Dunfield Joshua, Krishnaswami Neelakantan R. Complete and Easy Bidirectional Typechecking for Higher-Rank Polymorphism // Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. — ICFP '13. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 429–442. — Access mode: https://doi.org/10.1145/2500365.2500582.

[9] Dunfield Joshua, Krishnaswami Neelakantan R. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types // Proc. ACM Program. Lang. — 2019. — Jan. — Vol. 3, no. POPL. — Access mode: https://doi.org/10.1145/3290322.

[10] Floyd R.W. Assigning meanings to programs // Mathematical Aspects of Computer Science. — 1967.

[11] GHC Team. Glasgow Haskell Compiler User's Guide. — 2015. — Access mode: https://downloads.haskell.org/~ghc/8.10.2/docs/html/users_guide/extending_ghc.html (online; accessed: 2021-05-31).

[12] Ganesalingam M., Gowers W. A Fully Automatic Theorem Prover with Human-Style Output. — 2019. — 10. — P. 13–57. — ISBN: 978-3-030-28482-4.

[13] Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy / Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias et al. // Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '15. — New York, NY, USA : Association for Computing Machinery, 2015. — P. 55–68. — Access mode: https://doi.org/10.1145/2676726.2677000.

[14] Hoare C. A. R. Procedures and parameters: An axiomatic approach // Symposium on Semantics of Algorithmic Languages. — 1971.

[15] Jhala Ranjit, Vazou Niki. Refinement Types: A Tutorial. — 2020. — 2010.07763.

[16] Knowles Kenneth. Executable Refinement Types // ArXiv. — 2014. — Vol. abs/1403.3336.

[17] Leino K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness // LPAR. — 2010.

[18] McBride Conor. Epigram: Practical Programming with Dependent Types // Proceedings of the 5th International Conference on Advanced Functional Programming. — AFP'04. — Berlin, Heidelberg : Springer-Verlag, 2004. — P. 130–170. — Access mode: https://doi.org/10.1007/11546382_3.

[19] The NYU Ada Translator and Interpreter / Robert B. K. Dewar, Gerald A. Fisher, Edmond Schonberg et al. // SIGPLAN Not. — 1980. — Nov. — Vol. 15, no. 11. — P. 194–201. — Access mode: https://doi.org/10.1145/947783.948659.

[20] Probabilistic Relational Verification for Cryptographic Implementations / Gilles Barthe, Cédric Fournet, Benjamin Grégoire et al. // SIGPLAN Not. — 2014. — Jan. — Vol. 49, no. 1. — P. 193–205. — Access mode: https://doi.org/10.1145/2578855.2535847.

[21] Refinement Types for Haskell / Niki Vazou, Eric L. Seidel, Ranjit Jhala et al. // Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. — ICFP '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 269–282. — Access mode: https://doi.org/10.1145/2628136.2628161.

[22] Relational Cost Analysis / Ezgi Çiçek, Gilles Barthe, Marco Gaboardi et al. // Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. — POPL 2017. — New York, NY, USA : Association for Computing Machinery, 2017. — P. 316–329. — Access mode: https://doi.org/10.1145/3009837.3009858.

[23] A Relational Logic for Higher-Order Programs / Alejandro Aguirre, Gilles Barthe, Marco Gaboardi et al. // Proc. ACM Program. Lang. — 2017. — Aug. — Vol. 1, no. ICFP. — Access mode: https://doi.org/10.1145/3110265.

[24] Sage: Hybrid Checking for Flexible Specifications / Jessica Gronski, Kenneth Knowles, Aaron Tomb et al. — 2006. — 01.

[25] Secure distributed programming with value-dependent types / Nikhil Swamy, Juan Chen, Cédric Fournet et al. // Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011 / Ed. by Manuel M. T. Chakravarty, Zhenjiang Hu, Olivier Danvy. — ACM, 2011. — P. 266–278. — Access mode: https://doi.org/10.1145/2034773.2034811.

[26] Vekris Panagiotis, Cosman Benjamin, Jhala Ranjit. Refinement Types for TypeScript // SIGPLAN Not. — 2016. — Jun. — Vol. 51, no. 6. — P. 310–325. — Access mode: https://doi.org/10.1145/2980983.2908110.

[27] de Moura Leonardo, Bjørner Nikolaj. Z3: An Efficient SMT Solver // Tools and Algorithms for the Construction and Analysis of Systems / Ed. by C. R. Ramakrishnan, Jakob Rehof. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. — P. 337–340.