

Санкт-Петербургский государственный политехнический университет  
Институт информационных технологий и управления  
Кафедра распределенных вычислений и компьютерных сетей



Работа допущена к защите

Зав. кафедрой

\_\_\_\_\_ Ю. Г. Карпов

" \_\_\_\_ " \_\_\_\_\_ 2014г.

# ВЫПУСКНАЯ РАБОТА БАКАЛАВРА

Тема: *Разработка и реализация модульной системы проверки и  
вычисления типов*

Направление: *123456 — номер и название направления*

Выполнил студент гр. 43507

Руководитель, ст. преп.

М. А. Буряков

Д. А. Тимофеев

Санкт-Петербург  
2014



Утверждаю  
Зав. кафедрой  
\_\_\_\_\_ Ю. Г. Карпов  
" \_\_\_\_ " \_\_\_\_\_ 2014г.

**ЗАДАНИЕ**  
на дипломную работу  
студенту М. А. Бурякову

1. Тема: *Разработка и реализация модульной системы проверки и вычисления типов*  
(утверждена распоряжением по институту от \_\_\_\_\_ № \_\_\_\_\_)
2. Срок сдачи работы.
3. Исходные данные к проекту (работе).
4. Содержание расчетно-пояснительной записки (перечень подлежащих разработке вопросов).
5. Перечень графического материала с точным указанием обязательных чертежей.
6. Консультанты по проекту (с указанием относящегося к ним разделов проекта, работы) .

Дата выдачи задания: \_\_\_\_\_ г.

Руководитель: \_\_\_\_\_ ст. преп. Д. А. Тимофеев

Задание принял к исполнению: \_\_\_\_\_ М. А. Буряков

# Реферат

С. 51 , рис. 5

Краткая характеристика всего документа, основной результат работы,  
список ключевых слов

# Abstract

51 pages , 5 figures

Brief description of the work, main result, keywords

# Оглавление

<b>Список обозначений</b>	<b>7</b>
<b>Список терминов</b>	<b>8</b>
<b>Введение</b>	<b>10</b>
<b>1 Системы типов</b>	<b>15</b>
1.1 Простейшая система типов . . . . .	16
1.2 Объявление пользовательских типов . . . . .	16
1.3 Подтипы . . . . .	17
1.3.1 Проблема когерентности . . . . .	17
1.4 Множественные отношения над типами . . . . .	18
1.5 Типы, конструируемые из других типов . . . . .	18
<b>2 Используемые технологии</b>	<b>19</b>
2.1 Среда MPS . . . . .	19
<b>3 Системы типов в платформах разработки языков</b>	<b>21</b>
<b>4 Язык описания систем типов</b>	<b>26</b>
4.1 Парадигма языка . . . . .	26
4.2 Репозиторий типов . . . . .	27
4.2.1 Аспекты репозитория . . . . .	27
4.2.2 Многократная запись в репозиторий . . . . .	29
4.3 Правила типизации . . . . .	29
4.3.1 Аннотации правил . . . . .	30
4.3.2 Подбор правил . . . . .	31
4.3.3 Альтернативные правила . . . . .	32

4.3.4	Автоматическая генерация аннотаций . . . . .	32
4.4	Операции над типами . . . . .	33
4.5	Типовые переменные . . . . .	34
4.6	Иерархический контекст . . . . .	35
<b>5</b>	<b>Правила вывода типов для языка Haskell</b>	<b>36</b>
5.1	Объявление алгебраических типов . . . . .	38
5.2	Лямбда-выражение . . . . .	40
5.3	Применение функции к аргументу . . . . .	41
5.4	Ссылка на переменную лямбда-выражения . . . . .	41
5.5	Ссылка на объявление выражения или конструктора . . . .	42
5.6	Объявление выражения . . . . .	42
5.7	Выражение <code>let</code> . . . . .	43
5.8	Выражение <code>case</code> и сопоставление с образцом . . . . .	44
<b>6</b>	<b>Описание системы типов языка Java</b>	<b>45</b>
6.1	Правила для констант . . . . .	45
6.2	Выражения <code>.class</code> . . . . .	46
6.3	Правила для <code>this</code> . . . . .	46
6.3.1	unqualified <code>this</code> . . . . .	46
6.3.2	qualified <code>this</code> . . . . .	46
6.4	Операции . . . . .	46
	<b>Заключение</b>	<b>49</b>

# Список обозначений

AST	Abstract Syntax Tree	абстрактное синтаксическое дерево
MPS	Metaprogramming System	система метапрограммирования
GADT	Generalized algebraic datatypes	обобщённые алгебраические типы дан



# Список терминов

Абстрактное синтаксическое дерево (abstract syntax tree, AST) -

Приписывание типов - сопоставление некоторым вершинам AST корректно составленных типов.

Проверка типов - алгоритмическая процедура, анализирующая синтаксически корректное AST программы с приписанными типами и определяющая, является ли AST с приписанными типами корректным по типам.

Корректность по типам AST с приписанными типами - результат работы проверки типов.

Типизируемость AST (типизируемость программы) - существование такого приписывания типов, что AST с данным приписыванием типов корректно по типам.

Единственность типизации AST - единственность такого приписывания типов, что AST с данным приписыванием типов корректно по типам.

Единственность типов - свойство системы типов, заключающееся в том, что любое синтаксически корректное AST единственно типизируемо.

Частичная проверка типов - проверка типов для части AST с приписанными типами. Так как часть AST не является синтаксически корректным AST, требование синтаксической корректности снимается. Если часть AST с приписанными типами не прошла частичную проверку типов, то целое AST с такими же приписанными типами не должно пройти полную проверку типов.

Вычисление типов - алгоритмическая процедура, производящая приписывание типов. Может завершаться неудачей.

Система типов - множество корректно составленных типов, процедура вычисления типов, процедура проверки типов.

Язык программирования - определяет множество синтаксически кор-

ректных AST и систему типов.

Гарантированный вывод типов – алгоритм вычисления типов, обладающий следующим свойством. Если он завершает работу, то AST с написанными типами корректно по типам, а если не завершает, то AST не типизируемо.

Восстановление типов – то же, что и гарантированный вывод типов, в случае единственности типов.

Аннотация типа – упоминание типа в AST программы. Если типы записываются в виде деревьев, то аннотацией типа можно считать включение типа в AST программы в качестве поддеревя. Аннотации типов, как правило, непосредственно указывают типы вершин AST.

Переменная типа (типовая переменная) –

# Введение

Большая часть современных языков программирования имеют статическую систему типов для классификации конструкций, использующихся при описании программы (переменные, выражения, функции), по типам данных, которые будут соответствовать этим конструкциям в момент выполнения программы. Статическая система типов может рассматриваться как ограничение на абстрактное синтаксическое дерево программы [1]. Проверка и вычисление типов осуществляется путем обхода этого дерева, в ходе которого каждому узлу дерева приписывается некоторый тип. При этом типы некоторых конструкций задаются пользователем явно, а другие должны быть вычислены автоматически. Алгоритмы, по которым производится вычисление и проверка типов, различны для разных языков и задаются, как правило, в описании языка. Но среди различных алгоритмов вычисления и проверки типов многие алгоритмы отличаются лишь в деталях или имеют похожие элементы. Поэтому при создании инструментов, предназначенных для работы с различными языками, например, интегрированных среды разработки, возникает потребность в модульной системе, позволяющей описывать различные алгоритмы вывода типов, выделяя общие элементы для использования в различных алгоритмах.

Традиционно алгоритмы вычисления и проверки типов реализуются отдельно в каждом компиляторе или среде разработки для того или иного языка. Потребность в объединении различных алгоритмов появилась с появлением платформ для разработки языков (Jetbrains MPS, Xtext, Spoofax и др.). Некоторые платформы для разработки языков включают инструменты для описания систем типов. Подход, предлагаемый Jetbrains MPS для вычисления типов, основан на создании переменной типа для каждой типизируемой конструкции программы и построении системы уравнений и неравенств относительно этих переменных. После то-

го, как система уравнений и неравенств будет построена, встроенный решатель пытается найти решение этой системы, которое и будет представлять вычисленные типы для соответствующих конструкций программы. Недостатком данного подхода является неопределённость выбора решения, удовлетворяющего требуемым уравнениям и неравенствам. Если уравнения и неравенства составлены так, что их решение не единственно, то в результате работы алгоритма будет выбрано одно произвольное решение, причём при выборе используются некоторые недетерминированные эвристические соображения, в результате чего ответ может изменяться от запуска к запуску. Для получения корректного результата разработчику языка требуется дополнить уравнения и неравенства дополнительными условиями, чтобы устранить неопределённости. Однако практика показывает, что для языков, система типов которых не была изначально приспособлена для метода решения уравнений и неравенств (например, Java), записать уравнения и неравенства так, чтобы решение было однозначным, является достаточно сложной задачей и удаётся далеко не всегда.

Задача данной работы заключается в разработке системы библиотечных или языковых конструкций (при необходимости, язык или набор языков), позволяющий описывать различные алгоритмы вычисления и проверки типов в виде, подобном их неформальному описанию в виде правил. Правила, описываемые в этой системе, должны быть по возможности локальными, то есть каждое правило должно применяться к небольшому фрагменту AST, содержащему определённую конструкцию языка. Правила должны быть комбинируемыми, то есть при добавлении к языку новой конструкции изменение системы типов должно сводиться к добавлению одного или нескольких правил. Правила должны быть понятными по отдельности, то есть записанными на наглядном языке, и каждое правило должно представлять собой некоторую смысловую единицу, описанную в спецификации языка.

Для иллюстрации разрабатываемой системы планируется описать в ней алгоритмы вывода типов языка Haskell 98 (алгоритм Хиндли-Милнера с *let*-полиморфизмом) и языка Java 6. Данный выбор языков продиктован следующими причинами: алгоритм Хиндли-Милнера является наиболее известным алгоритмом вывода типов с помощью уравнений, а язык Java, в частности его статическая система типов, снабжён качественной спецификацией, структурное соответствие с которой может служить критерием наглядности записи правил. Также важным является тот

факт, что конструкции языка Java уже реализованы в среде MPS, и также в среде MPS разработано несколько расширений языка Java.

Правила типизации обычно описываются на некотором в той или иной степени формальном языке. Семантика такого языка достаточно декларативна, что не позволяет её использовать непосредственно для построения алгоритма, находящего корректную типизацию. Заметим, что, согласно определению [1, definition 8.2.1], типизация является наименьшим отношением, удовлетворяющим правилам. Такое определение неудобно для построения алгоритма, поэтому алгоритм вывода типов простого типизированного лямбда-исчисления основывается на лемме инверсии [1, lemma 8.2.2, lemma 9.3.1], которая позволяет для конкретного терма вычислить его тип. Поэтому правила, записанные декларативно, будем называть условиями или ограничениями типизации, а правила, позволяющие построить алгоритм приписывания типов (в данном случае основанные на лемме инверсии) — правилами вывода типов.

Замечание 1.1. Под правилами вывода типов будем понимать небольшие, локально действующие правила, которые определены для языка заранее, и их конечное число (так же как конечно число аксиом – аксиомопорождающая схема является одной аксиомой), приблизительно по числу значимых конструкций языка. Целый алгоритм вывода типов правилом не является, алгоритм реализуется в последовательном (лучше даже параллельном) применении определённым образом трактуемых правил.

Получение правил вывода типов из условий типизации является весьма сложным процессом, и в общем случае, выполняется вручную. Поэтому не будем ставить задачу автоматизировать данный процесс, хотя его автоматизация для конкретных классов систем типов вполне возможна и представляет практический интерес. Предположим, что разработчик системы типов будет её записывать уже в виде правил вывода типов. Тогда встаёт вопрос о том, как будут выглядеть эти правила вывода типов. В [1] приводятся примеры правил вывода для систем типов, не содержащих *type inference*. Однако *type inference* привносит некоторые сложности в язык описания правил вывода типов, особенно в сочетании с другими возможностями системы типов.

В частности, для систем типов, осуществляющих *type inference*, не подходят правила, берущие тип одних термов и вычисляющие на его основе тип других термов. Вместо этого используются правила, создающие уравнения (в общем случае это могут быть произвольные ограничения, например, для *bounded quantification* появятся неравенства). Уравнения на-

капливаются в специальном пуле, а затем решаются с помощью определённого алгоритма.

Таким образом, можно выделить два подхода к описанию правил вывода типов: через вычисление, когда типы одних термов объявляются функционально зависящими от типов других термов, и через ограничения, когда правила вывода лишь создают уравнения, которые впоследствии уравнения решаются специальным алгоритмом.

При реализации и описании системы типов конкретного языка выбирается тот или иной подход в зависимости от особенностей языка. Но, когда разрабатывается система типов для встраивания в *language framework*, заранее неизвестно, какого вида язык будет описан, поэтому требуется остановиться на каком-то одном подходе, уточнить его и все возможности системы типов реализовать через него.

Замечание 1.2. Можно, конечно, предложить две системы типов, каждая из которых будет использовать свой подход (язык правил вывода), но в таком случае возникнет вопрос их взаимодействия. Более того, даже в этом случае появится желание на каждом языке правил вывода выразить как можно больше возможностей системы типов, что и предлагается сделать в данной работе на примере одного языка правил вывода.

Подход через вычисление весьма прост в реализации, но есть мнение, что он не подходит для систем, осуществляющих *type inference*. У подхода к выводу типов через решение уравнений имеется ряд сложностей. Во-первых, правила, удобно записываемые через вычисления, весьма неудобно выглядят, будучи переписанными в виде уравнений, а зачастую они и вовсе не могут быть переписаны. При записи уравнениями нельзя взять тип той или иной вершины и произвести с ним нетривиальное вычисление, потому что в момент создания уравнений тип неизвестен, а в какой этап алгоритма решения уравнений встроить это вычисление, не очень понятно. Был предложен приём, позволяющий откладывать это вычисление до окончательного вычисления типа некоторой вершины, но использование этого приёма откладывает типизацию некоторых других вершин, что приводит к необходимости создавать уравнения в процессе решения уравнений, а это чревато непредсказуемыми ошибками. Во-вторых, есть проблемы с самим алгоритмом решения уравнений. Когда ограничения представляют собой только уравнения, успешно используется алгоритм унификации, но когда возникают неравенства, предложенный на данный момент алгоритм становится недетерминированным. Добавление произвольных отношений между типами только усугубит недетерминированность,

так как система отношений может иметь много решений, и нет способа однозначно выбрать то или иное решение, удовлетворяющее условию.

Задачей данной работы является уточнение языка описания правил вывода типов, использующего в качестве основы парадигму вычислений, и адаптация языка для того, чтобы на нём можно было выразить системы типов, выполняющие *type inference*. Одновременно с уточнением языка правил также необходимо определить способ применения правил (некоторую виртуальную машину), иначе семантика языка описания правил будет не вполне ясна.

В качестве примера систем типов, осуществляющих *type inference*, планируется использовать упрощённые или модифицированные варианты языков Haskell и Java. В случае успешного применения предложенных алгоритмов к данным языкам возможно рассмотрение систем высших порядков или сложных типсистемных расширений объектных языков.

# Глава 1

## Системы типов

Понятие типа и системы типов можно определить по-разному. Мы будем рассматривать систему типов как синтаксический метод нахождения в исходной программе некоторых видов ошибок. Так как метод синтаксический, то оперирует он представлением программы в виде абстрактного синтаксического дерева (AST). Метод этот состоит из двух этапов: сначала каждой вершине AST программы приписывается некоторый тип, затем производится проверка корректности программы вместе с приписанными типами.

*Замечание 1.1.* Как правило, тип требуется не каждой вершине, а только некоторым элементам языка, например, выражениям. В дальнейшем будем это опускать, и, говоря, что вершине приписывается тип, будем подразумевать вершины, соответствующие именно таким конструкциям языка.

*Замечание 1.2.* Этап вычисления типов теоретически является необязательным – программиста может все типы аннотировать явно. Но это оказывается крайне не удобно, поэтому желательно свести к минимуму необходимое количество аннотаций. В идеальном случае программист может совсем не указывать аннотаций типов, а некоторый алгоритм либо подберёт такие типы, чтобы типизированная программа прошла проверку типов, либо докажет, что такие типы подобрать невозможно. На практике такой идеальной ситуации гарантированного вывода типов без аннотаций удаётся достичь лишь для отдельных классов систем типов, а в остальных случаях либо от программиста требуются некоторые обязательные аннотации, позволяющие гарантировать алгоритмическую разрешимость



дальнейшего гарантированного вывода типов, либо вывод типов проводится без гарантии того, что если он не удался, то невозможно подобрать корректные типы.

Сложность и эффективность алгоритмов, работающих как на первом этапе вычисления типов, так и на втором этапе проверки типов, может варьироваться в зависимости от возможностей, которые предоставляет система типов.

## 1.1 Простейшая система типов

Простейший вариант системы типов содержит конечное число встроенных в неё типов. Листья AST являются либо константами, либо ссылками на переменные – и те, и другие имеют заранее известный тип, так как тип констант виден из вида литерала, а тип переменных указывается при объявлении переменной. В такой системе типов вычисление типов производится снизу вверх по дереву, то есть тип любой вершины может быть вычислен, если известны типы нижележащих вершин. Проверка типов также происходит просто, так как все требования локальны и могут быть описаны таблично.

## 1.2 Объявление пользовательских типов

Если язык программирования разрешает пользователю вводить новые типы, то вместе с новыми типами должны быть объявлены конструкторы этого типа и операции с выражениями этого типа. Возможны ситуации, когда конструкторы и операции задаются неявно при каждом объявлении типа. Может быть и так, что конструктор не требуется – достаточно лишь объявить переменную этого типа.

Как правило, добавление в язык возможности объявления пользовательских типов не сильно влияет на алгоритмы вычисления и проверки типов, так как можно перед началом вычисления типов обойти все объявления новых типов, совершить некоторые предварительные вычисления, а затем рассматривать эти типы как встроенные в язык.

## 1.3 Подтипы

Подтипы в их наиболее простом виде добавляют одно отношение частичного порядка над множеством типов – *отношение подтипа*. При этом правила проверки типов модифицируются таким образом, что в любом месте программы, где корректно выражение некоторого типа, корректным становится и выражения всех его подтипов (*принцип подстановки* Барбары Лисков, *##*). Такой подход к подтипам весьма интересен в теоретическом плане, однако никак не способствует созданию алгоритма вычисления типов. Поэтому в большинстве языков программирования, содержащих подтипы, алгоритм вычисления типов задаётся отдельно.

### 1.3.1 Проблема когерентности

Так как отношение подтипа является отношением нестрогого частичного порядка, то оно задаёт направленный граф, вершинами которого являются типы, а ребро соответствует отношению прямого подтипа. Проблема может возникнуть, если в таком графе несколько различных путей, связывающих одни и те же два типа.

Допустим, в некотором Java-подобном языке определены подтипы:

```
byte    <:  int
byte    <:  Byte
int     <:  Integer
Integer <:  Object
Byte    <:  Object
```

Тогда при подстановке выражения, имеющего тип `byte`, туда, где ожидается тип `Object`, возникает неоднозначность. Можно преобразовать тип `byte` в `Byte`, а затем тип `Byte` преобразовать в `Object`. А можно тип `byte` преобразовать в `int`, затем в `int` в `Integer`, а `Integer` в `Object`. При этом результаты последовательных конвертаций могут получиться разными.

Если независимо от пути в графе от некоторого типа *A* до его надтипа *B* в результате приведения некоторого значения типа *A* к типу *B* будет одинаковым, то говорят о *когерентной* функции приведения.

Заметим, что если функция приведения является тождественной функцией, то она когерентна. Но в некоторых языках программирования, например в Java, функция приведения типов не является когерентной. Поэтому в этих языках отказываться от принципа подстановки для всех пар

типов, которые могут быть приведены один у другому, оставляя принцип подстановки (и термин «подтип») только для тех пар типов, для которых функция приведения когерентна.

## 1.4 Множественные отношения над типами

Чтобы избежать проблем с некогерентной функций приведения типов, некоторые языки вводят несколько отношений, напоминающих отношение подтипа. Эти отношения определяют допустимость неявного приведения типов в тех или иных конструкциях языка. Каждое из этих отношений является либо простым (заданным таблично или алгоритмически), либо составным, то есть произведением определённой комбинации других отношений.

#пример: Java conversions#

## 1.5 Типы, конструируемые из других типов

Когда простых перечислимых типов недостаточно, в языках появляются составные типы, содержащие в себе в качестве составной части другие типы. Составные типы удобно представлять в виде деревьев, листьями которых будут простые типы.

В качестве примеров составных типов можно назвать:

- функциональные типы (например, `Int -> String` в языке Haskell),
- конкретизации полиморфных типов (например, `List<Integer>` в языке Java).
- типы записей в языках с полиморфизмом строчных переменных (например, `##` в языке SML).

#Неразрешимость системы типов Java. Пример некорректной работы `javac`. F-bounded quantification и кому это нужно ([github.com/ohnosequences/ohnosequences.github.com/blob/blog/post/typed/graphs/posts/2014-05-18-typed-property-graphs-in-java.markdown](https://github.com/ohnosequences/ohnosequences.github.com/blob/blog/post/typed/graphs/posts/2014-05-18-typed-property-graphs-in-java.markdown)) Enum #

## Глава 2

# Используемые технологии

### 2.1 Среда MPS

Для практического построения моделей систем типов планируется использовать такой language framework как JetBrains MPS. Удобство его использования для этой цели заключается в том, что он позволяет легко описывать языки, а также предоставляет некоторый высокоуровневый API для работы с кодом, написанном на этих языках. MPS уже содержит язык описания правил вывода типов, однако язык этот основан на создании уравнений, а следовательно, достаточно ограничен.

Использование MPS подсказывает некоторые особенности машины системы типов. Во-первых, типы присваиваются не терму, оторванному от контекста, а конкретной вершине AST. Это вполне отвечает практической необходимости, потому как возможны системы типов, где одно и то же поддерево AST, повторенное в разных местах дерева, будет иметь в них различный тип. Во-вторых, сами типы также записываются в виде AST. В случае простых типов это не имеет особого значения, но уже появление параметризованных типов (которые ещё и могут встречаться в тексте программы) выявляет общую природу кода на языке и типов этого языка.

Среда MPS позволяет использовать в одной программе разные языки, который, в общем случае, могут быть разработаны независимо. Для совместимости языков, в числе прочего, требуется совместимость систем типов. Для этого требуется, как минимум, существование движка системы

типов, позволяющего разработчикам языка описывать правила вычисления типов. Таким образом, нужно прояснить набор базовых понятий, которые нужны для описания правил системы типов, причём этот набор должен быть достаточно богат, чтобы описать как Haskell-like, так и ML-like и Java-like системы типов. При этом следует оставить как можно меньше универсальных, основополагающих понятий, а остальные выразить через них. При этом для описания систем типов разных классов можно создать несколько удобных языков, генерируемых в минималистичный язык/API движка типсистемы.

## Глава 3

# Системы типов в платформах разработки языков

В связи с развитием сред разработки языков, которые сильно упрощают создание новых языков в отношении синтаксиса, редактора и трансляции, возникает необходимость в разработке технологии создания систем типов для создаваемых языков.

Основные действия, которые может производить система типов ([#пересказ книги dslbook.org](#) (Marcus Voelter)<#>).

- Присваивает фиксированный тип некоторым простым элементам программы. Например, такими простыми элементами, тип которых не нужно вычислять, потому что он известен заранее, являются числовые или строковые литералы.
- Переносит (копирует) тип одного элемента программы на другие связанные с ним элементы программы. Например, тип выражения, состоящего из переменной — такой же, как тип, указанный при объявлении этой переменной.
- Вычисляет типы более сложных элементов программы на основе типов составных частей. Например, тип бинарной операции может за-

висеть от типов её аргументов, причём эта зависимость может быть достаточно нетривиальной.

- Производит проверку типов. В конечном итоге, система типов должна проверить программу на наличие ошибок, связанных с типами, и сообщить о них пользователю. Для этого язык определяет набор ограничений на типы в программе или набор процедур проверки типов, которые запускаются для вычисленных типов и сообщают пользователю об ошибках.

Типы удобно представлять таким же способом, как и AST программы. Во-первых, это объясняется тем, что типы также часто имеют древовидную структуру. Во-вторых, типы обычно используются в программе, чтобы явным образом указать тип того или иного элемента программы (например, во многих языках тип переменной всегда должен быть указан явно).

Теоретически, систему типов можно рассматривать (если временно забыть о проверках типов) как функцию `typeof`, которая вычисляет тип указанного элемента программы (тип вершины AST). Эта функция может быть реализована любым удобным образом, однако на практике чаще используются три основных подхода: рекурсия, унификация и сравнения с образцом.

В рекурсивном подходе к вычислению типа имеется полиморфная (перегруженная) функция `typeof`, которая принимает элемент программы и возвращает его тип, при этом вызывая себя, чтобы вычислить типы тех элементов программы, от которых зависит вычисляемый тип.

Унификация предполагает, что разработчики языка определили набор уравнений, содержащих типовые переменные и типы-значения. После этого некоторая машина пытается решить уравнения, присваивая переменным значения. Интересным свойством этого подхода является отсутствия различия между выводом и проверкой типов. Разработчик языка всего лишь задаёт набор требований, которые должны удовлетворяться в корректной программе. Если решения, удовлетворяющего всем требованиям, не существует, то это означает, что в программе обнаружена ошибка.

При сравнении с образцом вычисление типа одного элемента программы на основе типов других элементов производится с помощью таблицы. В таблице отмечаются все возможные варианты типов связанных элементов, и на основе этой таблицы производится как вывод, так и проверка

типов. Те варианты, которые не отражены в таблице, считаются ошибочными.

На данный момент известны три платформы для разработки языков, для которых существуют инструменты для создания систем типов.

**Xtext** — платформа для создания текстовых предметно-ориентированных языков программирования, входящая в Eclipse Modeling Project. Известны два различных инструмента для описания систем типов в Xtext: XTypes (#разработчик - Lorenzo Bettini#) и Xtext Typesystem Framework (#разработчик - Marcus Voelter#).

Xtext Typesystem Framework основан на рекурсивном подходе к вычислению типов. Он удобен для описания систем типов, которые сводятся к операциям присваивания фиксированных типов, переноса типов одного элемента на другой, вычисление типов элемента из типов его частей. Данные операции записываются декларативно с использованием специально созданного языка. Более сложные операции могут быть реализованы на императивном языке общего назначения, но только если они согласуются с рекурсивным подходом. В частности, система типов Хиндли-Милнера не соответствует рекурсивному подходу, а значит может быть реализована только в виде отдельного блока, то есть фактически, реализация системы типов Хиндли-Милнера будет лишь совместимой с Xtext Typesystem Framework, но не использующей его.

**MPS** (Metaprogramming System, система метапрограммирования) — платформа для создания предметно-ориентированных языков, использующая проекционный редактор. Имеет встроенную поддержку систем типов.

MPS содержит предметно-ориентированный язык для описания правил системы типов. Он основан на унификации, однако содержит ограниченную поддержку принципа сравнения с образцом. Уравнения, составляемые правилами и затем решаемые по специальному алгоритму, бывают нескольких видов.

- Собственно уравнения, требующие, чтобы тип в левой части структурно совпадал с типом в правой части.
- Неравенства, требующие, чтобы тип левой части структурно совпадал с некоторым надтипом типа в правой части. При этом отношений



подтипа возможны два: сильное и слабое. Слабое следует из сильного, обратное неверно. Сами оба отношения подтипа считаются рефлексивными и транзитивными и задаются либо в виде задания у некоторого типа списка его непосредственных надтипов, либо в виде набора уравнений, логически равносильных утверждению о том, что некоторый тип является подтипом другого типа.

- Требования сравнимости, требующие, чтобы два типа находились в отношении сравнимости. Отношение сравнимости симметрично и рефлексивно. Какие типы можно считать сравнимыми, а какие нет — задают отдельные правила.

Существенным недостатком такой модели описания системы типов является невозможность обеспечить транзитивность отношения подтипа. Хотя, конечно, могут существовать в принципе алгоритмически неразрешимые системы типов, но и во многих простых ситуациях транзитивность отношения подтипа, номинально постулируемая, на практике не обеспечивается.

Ещё один недостаток этой модели заключается в том, что решений, удовлетворяющих системе неравенств, может быть много, и решение, возвращаемое реализацией решающего алгоритма, выбирается из них некоторым не вполне предсказуемым образом. Однако разработчику языка, как правило, нужно, чтобы решение было выбрано по некоторым конкретным правилам. Проблема в том, что не предусмотрено механизма, позволяющего разработчику описать алгоритм, по которому из нескольких допустимых решений системы будет выбран ответ.

**Spoofax** — IDE, онованная на Eclipse, для создания текстовых языков. В отличие от MPS, основным понятием которого является вершина AST, Spoofax оперирует термами, то есть, например, два одинаковых выражения, встретившиеся в разных местах программы, неразличимы вне контекста программы. Имеет встроенную поддержку систем типов.

Системы типов в Spoofax основываются на рекурсивном подходе с поддержкой сравнения по образцу. Следует отметить удобные декларативные языковые конструкции для описания правил системы типов и отношений между типами. Эти языковые конструкции частично транслируются, частично интерпретируются машиной системы типов, которая эффективно вычисляет типы в программе. Однако, так же как и в Xtext Typesystem Framework, метод уравнений унификации не поддерживается,

и на данный момент систему типов Хиндли-Милнера в Spoofax реализовать невозможно.

## Глава 4

# Язык описания систем типов

### 4.1 Парадигма языка

Языки, допускающие лишь некоторый ограниченный набор операций, не позволяющий выразить достаточно сложные вычисления, подходят лишь для заранее очерченного круга задач. Поэтому логично составлять язык для виртуальной машины, выводящей типы, так, чтобы позволить пользователю производить произвольные вычисления с типами и другими объектами. Для этих целей язык описания систем типов допускает включение в определённых местах произвольных фрагментов кода на языке общего назначения (Java с расширениями).

Альтернативным вариантом мог бы являться полностью декларативный подход к описанию языков, при котором не допускается использование тьюринг-полного императивного языка для описания вычислений. Но такой полностью декларативный подход к описанию систем типов имеет ряд недостатков.

Во-первых, системы типов большинства разрабатываемых предметно-ориентированных языков описываются в терминах последовательных вычислений. В результате такие, как правило, простые системы типов будут требовать значительных усилий, чтобы их описать на декларативном языке. Перевод в обратную сторону — из декларативного описания в спецификации языка в описание с использованием

императивного языка — представляется менее затратным относительно сложности самой системы типов, так как та или иная декларативная семантика спецификации системы типов используется для языков общего назначения с достаточно сложной системой типов, при этом разработчик, взявшийся за непростую задачу описания языка общего назначения в универсальной среде описания языков, в любом случае будет тратить достаточно много времени на осмысление семантики описания системы типов этого языка.

Во-вторых, если язык описаний систем типов не будет тьюринг-полным, то в какой-то момент найдётся система типов, требующая таких вычислений, которые невозможно выразить на этом языке. Если же этот язык сможет описать сколь угодно сложные вычисления, то этот язык окажется достаточно сложным по структуре и семантике, и будет весьма непростым для изучения. Поэтому в данной работе принято решение использовать язык общего назначения (Java) для описания сколь угодно сложных вычислений, и специально разработанный предметно-ориентированный язык для оперирования понятиями, специфичными для задачи вывода и проверки типов.

## 4.2 Репозиторий типов

Язык для виртуальной машины, выводящей типы, позволяет производить произвольные вычисления с типами и другими объектами. Логично потребовать, чтобы все вычисления, производимые в процессе вывода и проверки типов, не имели побочных эффектов. Это означает, что все обращения к типизируемому коду, обращения к состоянию машины, выводящей типы, и изменения состояния машины производятся только через специальные команды. Состояние машины содержит типы, присвоенные вершинам. Они хранятся в виде репозитория типов. Репозиторий типов представляет собой хранилище, сопоставляющее каждой вершине абстрактного синтаксического дерева (AST) вычисленный для неё тип. Взаимодействие с репозиторием типов осуществляется через две операции: *чтение типа* и *запись типа*.

### 4.2.1 Аспекты репозитория

Кроме непосредственно типов, может понадобиться хранить результаты промежуточных вычислений. Их тоже удобно хранить привязанными к

определённой вершине абстрактного синтаксического дерева, так же как и типы, в репозитории. Для этой цели введена возможность присваивать одной и той же вершине абстрактного синтаксического дерева одновременно несколько «типов», различая их по специальным меткам, *аспектам репозитория*.

Аспекты репозитория объявляются отдельно. Чтение из репозитория и запись из репозитория производится с указанием аспекта репозитория, по которому должно быть произведено чтение или запись, и вершины AST, к которой привязана читаемая или привязывается записанная информация. В большинстве практических случаев по одному из аспектов репозитория хранятся собственно типы выражений языка, а другие аспекты объявляются для вспомогательных целей. Примером осмысленного дополнительного аспекта может быть аспект, по которому для некоторой вершины записывается максимальный допустимый тип выражения, которое могло бы стоять на месте этой вершины. Такой аспект чрезвычайно полезен для автоматического дополнения кода в интегрированной среде разработки.

Также множественность аспектов полезна и с точки зрения совместимости. Если различные разработчики независимо друг от друга создали системы типов для одного языка, и есть необходимость использовать обе эти системы, то они могут работать, не мешая друг другу, если будут использовать различные аспекты репозитория. При этом каждая система получит как бы свой репозиторий, и эти «репозитории» можно считать независимыми.

Особенно полезно одновременное использование различных систем типов может быть, если в рамках модели системы типов осуществляются различные проверки, не относящиеся непосредственно к типам так, как они описаны в спецификации языка, но, подобно системам типов, производящие более сложные статические проверки, в том числе с использованием внешних инструментов.

Таким образом, систему типов можно использовать в том числе как интерфейс взаимодействия внешних инструментов статического анализа с интегрированной средой разработки.

В дальнейшем для простоты изложения любые данные, записываемые в репозиторий по некоторому аспекту, будем называть типами по этому аспекту.

### 4.2.2 Многократная запись в репозиторий

При объявлении аспекта он может быть определён либо как аспект с однократной записью в репозиторий, либо как аспект, поддерживающий многократную запись.

Аспекты с однократной записью запрещают запись в репозиторий для вершины, которой уже приписан тип по данному аспекту. Если же тип по данному аспекту вершине не приписан, то запрещена операция чтения. Таким образом, текущий тип вершины по данному аспекту может находиться в двух состояниях: ещё не записан или уже записан. При этом, если система типов детерминирована, то операция чтения типа вершины по данному значению, если она возможна, вернёт всегда одно и то же значение.

Аспекты, разрешающие многократную запись, не накладывают подобных ограничений на операции записи и чтения — запись и чтение типа по этому аспекту всегда считается допустимым. В таком случае возникает два вопроса.

1. Что вернёт операция чтения, если до этого записи не производилось?
2. Что останется в репозитории после операции записи, если в репозитории уже был записан тип для этой вершины?

Ответ на эти вопросы можно предоставить, если при объявлении аспекта, разрешающего многократную запись, указать моноид, содержащий значение типа по этому аспекту перед первой записью и ассоциативную функцию, вычисляющую новое значение на основе значения, которое было в репозитории до этого, и значения, переданного в качестве параметра в функцию записи.

*Например, для системы типов Хиндли-Милнера единицей моноида является свободная переменная типа, а операцией умножения — структурная унификация двух типовых деревьев.*

*#Пример: Конъюнкция требований#*

## 4.3 Правила типизации

Основу описания систем типов на разработанном языке составляют локальные правила типизации. Правило типизации состоит из *шаблона*

*AST*, по которому определяется применимость правила, специальных *аннотаций* и фрагмента программного кода, непосредственно осуществляющего необходимые для вывода и проверки типов действия, — *действующей процедуры правила*.

Каждое правило применяется к тем фрагментам *AST*, которые соответствует шаблону этого правила. Если правило может быть применено к некоторому фрагменту (поддереву) *AST*, создаётся экземпляр этого правила, привязанный к корневой вершине этого поддерева, которая становится *корневой вершиной экземпляра*. Затем у созданного экземпляра правила вызывается действующая процедура. Результатом работы этой процедуры, а значит и результатом применения правила к фрагменту *AST*, является внесение изменений в репозиторий типов в отношении вершин из этого фрагмента.

Шаблоном может служить, к примеру, конкретная конструкция языка. Например, в большинстве императивных языков будет правило, применимое к операциям присваивания. Для каждого присваивания в типизируемой программе будет создан экземпляр этого правила. Действующая процедура правила прочтает из репозитория тип переменной из левой части присваивания, прочтает тип выражения из правой части и сравнит эти типы на совместимость.

Как видно из примера, правило может обращаться к репозиторию типов с запросами на чтение или запись. Но если запросы производятся по аспекту с одноразовой записью, то перед тем, как производить чтения типа некоторой вершины, этот тип должен быть записан в репозиторий в процессе выполнения другого правила. Это требование формирует достаточно строгие ограничения на порядок запуска экземпляров правил.

### 4.3.1 Аннотации правил

Требования к порядку применения правил содержатся в самих правилах в виде специального вида аннотаций. *Минимальный набор аннотаций*, позволяющий запускать правила в правильном порядке, состоит из одной функции, отвечающей на вопрос, может ли экземпляр правила быть выполнен в данный момент (достаточно ли уже вычислено информации для корректной работы данного экземпляра правила).

Но такой минимальный набор не всегда приводит к эффективной работе. Если появятся циклические зависимости, при которых ни одно правило не может быть выполнено, то это выяснится только в процессе выполнения правил. Также неэффективно после завершения работы каждого эк-

земпляра правил опрашивать все оставшиеся правила, чтобы выбрать из них то, которое должно быть выполнено следующим.

Чтобы определять порядок выполнения правил заранее, до начала их выполнения, используются *дополнительные аннотации зависимостей* правил. В начале, после создания всех необходимых экземпляров, каждый экземпляр предоставляет список вершин, тип которых должен быть в репозитории на момент запуска этого экземпляра. Также каждый экземпляр предоставляет список вершин, тип которых будет записан в репозиторий в результате работы этого экземпляра. На основе этих данных составляется направленный двудольный граф, вершинами которого являются, с одной стороны, вершины AST (по некоторому аспекту) и, с другой стороны, экземпляры правил. Дуга графа, идущая из экземпляра правила в вершину AST, означает, что это правило записывает (или, если дуга пунктирная, то предположительно записывает) в репозиторий тип этой вершины. Дуга графа, идущая из вершины AST в экземпляр правила, означает, что этому правилу требуется (или, если дуга пунктирная, то предположительно требуется) тип этой вершины.

# Пример графа - рисунок #

### 4.3.2 Подбор правил

В большинстве ситуаций не требуется вычислять и проверять типы для всего проекта. Например, компиляция крупных проектов происходит отдельными файлами или модулями. Но часто файл тоже является достаточно большой единицей типизации. Например, это может происходить при редактировании текста в IDE, когда при небольшом изменении кода в редакторе нет необходимости перевычислять типы всех выражений в файле. Также возможна ситуация, когда требуется вычислить тип одной конкретной вершины (например, эта необходимость возникает при генерации некоторых высокоуровневых языков). В этом случае нужно создавать и запускать только те экземпляры правил, которые участвуют в выводе типа нужной вершины.

Для решения задачи подбора правил используются *обратные аннотации зависимостей* правил. Обычных (прямых) аннотаций зависимостей правил для этой цели недостаточно, так как прямые аннотации зависимостей вычисляют по экземпляру правила вершины AST, используемые этим правилом, то есть строят двудольный граф зависимостей со стороны экземпляров правил. А в ситуации, когда экземпляры правил ещё не со-



зданы, и требуется найти те из них, которые нужно создать, то же граф должен строиться со стороны вершин AST.

### 4.3.3 Альтернативные правила

Каждое правило типизации содержит небольшой фрагмент алгоритма типизации. В результате последовательного выполнения экземпляров правил в некотором порядке результат работы должен получиться такой, как если бы работал алгоритм типизации, не разбитый на правила.

Однако в алгоритме типизации могут быть описаны исключительные ситуации, которые запускаются при некотором условии и в которых действия полностью отличаются от основной линии действия алгоритма. Такие ситуации могут описаны на разработанном языке с помощью альтернативных правил.

Для того, чтобы описать принцип работы замещающих правил, вернёмся к модели выполнения правил. Сначала создаются необходимые экземпляры правил. Затем на основе аннотаций определяется некоторая информация о порядке выполнения экземпляров. После этого экземпляры правил запускаются в требуемом порядке (окончательно порядок определяется уже во время выполнения). В конце все правила должны быть выполнены.

Альтернативные правила, если срабатывает условие их запуска, отменяют необходимость в выполнении тех правил, которые они замещают. Если же альтернативные правила не понадобились, то они автоматически отменяются при выполнении основных правил.

Использование альтернативных удобно, если требуется отдельно обрабатывать ситуации, когда между правилами образуется циклическая зависимость. Образование «затора», когда не может быть запущено ни одно правило, будет условием запуска альтернативных правил, а сами альтернативные правила в результате своей работы отменяют некоторые из заблокированных правил, разбив таким образом образовавшийся цикл.

### 4.3.4 Автоматическая генерация аннотаций

Локальные правила вывода применяются в определённом порядке, так, чтобы информация, необходимая для работы некоторого правила, была занесена в репозиторий типов правилами, применёнными до этого.

Для того, чтобы корректно определять порядок выполнения правил, необходим способ определения для каждого правила информации, необ-

ходимой для работы этого правила, и информации, записываемой в репозиторий в результате работы этого правила. Если правила описываются пользователем на языке общего назначения, то они могут быть снабжены соответствующими аннотациями, если же правила генерируются из предметно-ориентированного языка, то аннотации могут быть получены на основе статического анализа кода.

Метаинформация, прилагающаяся к правилу, частично может быть выведена из кода правила с помощью статического анализа, что представляет собой перспективу для дальнейших исследований.

## 4.4 Операции над типами

Некоторые языки программирования при описании их системы типов используют понятие отношений над типами. Наиболее распространённым отношением является отношение подтипа, однако в языке могут существовать и несколько отношений. Например, в языке Java 2 используются 8 отношений. В проектируемой системе предлагается описывать отношения с помощью более общей концепции операций с типами. В зависимости от целей использования бинарное отношение можно задать в виде бинарной булевой функции или в виде функции от первого аргумента, возвращающей последовательность допустимых вторых аргументов.

Так как система проектируется так, чтобы допустить возможность расширения набора используемых типов, то отношения также должны объявляться расширяемо. Для этого предложено разделять объявление операций над типами от объявлений реализации этих операций.

Объявление операции над типами содержит количество аргументов этой операции и тип результата.

Реализаций одной операции может быть несколько для разных шаблонов аргументов.

Реализация  $n$ -арной операции содержит  $n$  шаблонов аргументов и функцию, непосредственно производящую вычисление.

Если реализации не пересекаются по шаблонам аргументов, то операция является функцией (возможно, частично определённой). Если же к некоторым аргументам могут быть применимы одновременно несколько реализаций, то возникает вопрос о том, которая из них должна быть применена.

В противном случае возможны 2 разные политики вычисления в зависимости от вычисляемой операции. Политика приоритетов предполагает

некорректное завершение вычисления, если нет ни одной подходящей реализации. В случае же нескольких подходящих реализаций вызывается та из них, которая имеет наибольший приоритет.

Альтернативой политики приоритетов является политика равноправных реализаций. Операция определяет коммутативный моноид, содержащий значение по умолчанию, возвращаемое для тех аргументов, для которых не нашлось ни одной подходящей реализации, и бинарную функцию, по которой производится свёртка результатов работы всех подходящих реализаций, если подходящих реализаций несколько.

## 4.5 Типовые переменные

*Типовые переменные* могут использоваться для двух целей [1, раздел 22.2]: представление полиморфных типов и использование в промежуточных вычислениях во время вывода типов.

В разработанном языке описания систем типов при создании переменной не требуется указывать, с какой из вышеуказанных целей будет использоваться переменная (будет ли её значение вычислено, или над ней появится квантор всеобщности). После создания переменной она является свободной, то есть её значение не определено. В дальнейшем её значение может стать конкретным типом или типом, выраженным через другие переменные. Это может произойти через операцию присваивания значения типовой переменной. После того, как переменной присвоено значение, она становится зависимой переменной, и в каждом месте, где она используется, она заменяется на своё значение. На практике разумно заменять упоминания переменной на её значение не непосредственно в тот момент, когда переменная перестаёт быть свободной, а лениво, при первом запросе к её упоминанию. В результате такой замены все переменные, упоминания которых можно наблюдать в репозитории, являются свободными.

Если же значение переменной так и не будет определено в процессе вывод типов, то есть после выполнения всех экземпляров правил вывода, которые могли присвоить её значение, она всё равно осталась свободной, значит она может принимать любые значения. Как только об этом становится известно, переменная перемещается в список переменных некоторого полиморфного (содержащего переменную) типа.

Принцип локальности подсказывает, что типовые переменные чаще используются вблизи того места в дереве, где они были объявлены. Поэтому логично, создавая переменную, привязывать её к определённому

месту в дереве. А так как, если переменной не был присвоен тип, она становится частью полиморфного типа некоторой вершины, то логично привязывать её к той вершине, частью полиморфного типа которой она может стать.

## 4.6 Иерархический контекст

Чтобы язык было удобнее расширять, правила типизации должны быть локальными, то есть экземпляр правила должен обращаться только к корневой вершине экземпляра или к её непосредственным подвершинам. Крайне неудобно, если правила содержат код, идущий вверх по дереву до нахождения определённой конструкции языка. Однако получение информации из вышележащих вершин часто необходимо для эффективной работы системы типов.

Для решения этой проблемы используется способ передачи информации на большие расстояния через контекст `[#Elovkov D.#]`. Контекст, подобно репозиторию типов, предоставляет операции чтения и записи для некоторой вершины и может содержать много аспектов. Но, в отличие от репозитория типов, значение, записанное в репозиторий типов для некоторой вершины, может быть прочитано для любой вершины из её поддерева. Операция чтения возвращает список всех значений, записанных для всех её надвершин, включая саму вершину.

Для вывода типов оказывается вовсе не обязательно запускать экземпляры правил, которые записывают значение в контекст, если эти значения впоследствии не будут использованы. Если, к примеру, в некотором месте производится чтение из контекста для некоторой вершины, а затем из прочитанного списка берётся последний элемент, соответствующий ближайшей надвершине, для которой производилась запись в контекст, то можно запустить лишь тот экземпляр правила, который записывает этот последний элемент, и опустить все вышележащие записи. Чтобы корректно определить, какие экземпляры правил запустить необходимо, а какие можно и не запускать, используются специального вида *аннотации контекста*, которые могут быть записаны вручную или сгенерированы с помощью статического анализа кода.

## Глава 5

# Правила вывода типов для языка Haskell

На вышеизложенном языке в качестве иллюстрации его возможностей реализована система типов части языка Haskell. Из конструкций и возможностей языка взяты:

1. лямбда-выражение,
2. применение функции к аргументу,
3. конструкция `let` с полиморфными объявлениями внутри,
4. алгебраические типы данных,
5. полиморфные типы,
6. сравнение с образцом.

Каждое правило вывода типов объявляется для одной конструкции языка. В процессе вывода к каждой вершине этой конструкции будет применён соответствующий экземпляр правила.

Для описания типов языка используются две конструкции.

1. Конкретный (мономорфный) тип. Это может быть алгебраический тип, функциональный тип и даже тип, параметризованный переменной (переменная рассматривается как временно свободная). При-

меры конкретных типов приведены на рисунке ?? (красной волнистой линией подчёркнуты переменные).

2. Полиморфный тип (первого ранга). Содержит конкретный тип, но может синтаксически связывать некоторые переменные квантором всеобщности (обрамляющим конкретный тип). Заметим, что не обязательно все переменные должны быть связаны квантором — некоторые переменные могут остаться синтаксически не связанными. Возможна также ситуация, когда квантор не будет связывать ни одной переменной — важна сама возможность связать переменные квантором. Примеры конкретных типов приведены на рисунке ?? (красной волнистой линией подчёркнуты переменные, не связанные кванторами).

Для языка Haskell объявлены два основных аспекта репозитория типов (о вспомогательных аспектах см. #про циклы#, #где хранятся переменные#).

Аспект `haskell.expression` предназначен для хранения типов выражений языка. Типы выражений языка являются конкретными типами, то есть не содержат переменных, связанных квантором. Этот аспект разрешает многократное присваивание. Значением по умолчанию является свободная переменная (универсальный тип). Операцией, которая запускается при повторном присваивании типа одной и той же вершине, для этого аспекта является операция унификации.

Унификация двух алгебраических типов для различных типов возвращает ошибку, а для одинаковых запускает унификацию для их соответствующих параметров. Аналогично для функциональных типов и применений типов. Если вызвана унификация свободной переменной (заметим, что все переменные являются свободными, так как сразу после присваивания типа переменной она заменяется на своё значение) и типа, не являющегося переменной (алгебраического типа данных, функционального типа, применения типов), то переменная перестаёт быть свободной и ей присваивается конкретное значение. Если же вызвана унификация двух свободных переменных, то они обе выражаются через новую переменную. Создание новой переменной вместо выражения одной переменной не только выглядит более симметрично, но немного упрощает алгоритм. Во-первых, при использовании классов типов (которые могут быть добавлены в язык в качестве расширения) новая переменная будет совмещать в себе классы, содержащиеся в обоих исходных переменных. Во-вторых,

```
data Bool = False | True
data Maybe a = Just a | Nothing
```

Рис. 5.1. Объявления алгебраических типов данных

новая переменная создаётся привязанной к минимальной общей надвершине вершин, к которым привязаны исходные переменные. #где хранятся переменные# #схема, пример с мономорфизмом#

Аспект `haskell.declaration` предназначен для хранения типов `let`-объявлений и объявлений верхнего уровня (заметим, что объявления верхнего уровня синтаксически не отличаются от объявлений внутри выражения `let`, поэтому в дальнейшем будем называть их объявлениями). Для поддержки `let`-полиморфизма типы объявлений являются полиморфными типами, то есть начинаются с квантора всеобщности. При этом, если объявление содержится внутри `let`-выражения, то кванторами могут быть связаны не все переменные, а если объявление глобально, то все переменные должны быть связаны кванторами (впрочем, это условие является не требованием, а свойством алгоритма и может быть нарушено при расширении языка, например, при добавлении параметризованных модулей). Данный аспект не допускает многократного присваивания, так как в системе типов Haskell единицей типизации является объявление, поэтому тип объявления может вычисляться только в случае необходимости.

Рассмотрим основные конструкции упрощённого языка Haskell и правила для них.

## 5.1 Объявление алгебраических типов

Пример объявления алгебраических типов `Bool` и `Maybe` приведён на рис. 5.1. Для каждого алгебраического типа нужно выставить типы его конструкторам. Конструктор с  $n$  аргументами будет иметь функциональный тип с  $n$  аргументами, причём аргументы и результат копируются непосредственно из объявления.

Правило для данной конструкции языка будет выполнять следующие действия.

Для каждого конструктора объявляемого типа:

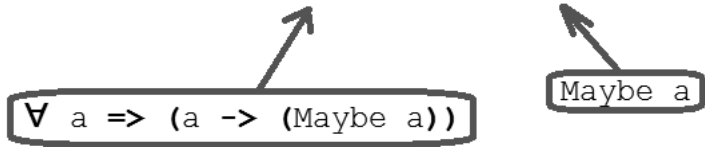
1. скопировать объявляемый тип со всеми параметрами (в рассматриваемых примерах это `Bool` и `Maybe a`) в локальную переменную

```
data Bool = False | True
```



(a) Тип Bool

```
data Maybe a = Just a | Nothing
```



(b) Тип Maybe

Рис. 5.2. Типы конструкторов

```
result;
```

2. создать функциональный тип, результатом которого будет **result**, а аргументами — аргументы конструктора;
3. записать получившийся тип в репозиторий типов в качестве типа конструктора (рис. 5.2).

Замечание 4.1.1. Так как алгебраические типы могут иметь параметры, то конструкторы могут иметь полиморфный тип. Для удобного представления потенциально полиморфных типов в виде AST создан концепт `PolymorphicType`, состоящий из дерева обычного типа (возможно, содержащего переменные) и списка использованных переменных. Введение особой конструкции для потенциально полиморфных типов является лишь удобным способом организовать привязку (binding) переменных типа.

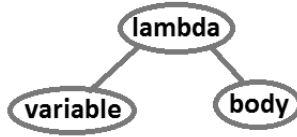
Замечание 4.1.2. В текущей версии типизируемого языка не поддерживаются классы типов. Все переменные типов не имеют ограничений.

Замечание 4.1.3. Проверка `type kinds` не влияет на вывод типов в корректно составленной программе. Поэтому не будем включать эту проверку в правила вывода типов. Для проверок, от которых не зависит основной



$$\lambda x \rightarrow x$$

(a) Тестовое представление



(b) Структура синтаксического дерева

Рис. 5.3. Лямбда-выражение

процесс вывода типов, будем создавать отдельные правила, чтобы обеспечить модульность системы типов (например, проверки можно в некоторых ситуациях отключать для ускорения вычисления типов, если требуется не найти все ошибки типов, а вычислить конкретный тип. Проверка `type kinds` потребует добавить дополнительные поля в концепт переменной типа, это будет впоследствии уточнено.

## 5.2 Лямбда-выражение

Правило для лямбда-выражения, согласно алгоритму Хиндли-Милнера, должно создавать уравнения унификации, связывающие тип переменной, тип возвращаемого значения и тип самой лямбда-функции. Чтобы на разработанном языке системы типов связать типы разных вершин, присвоим этим вершинам типы (по аспекту `haskell.expression`), содержащие одну и ту же переменную.

Правило типизации для лямбда-выражения будет выглядеть следующим образом:

```
argType = new fresh type variable;
bodyType = new fresh type variable;
typeof(arg) := argType;
typeof(body) := bodyType;
```

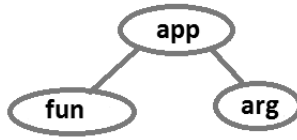


Рис. 5.4. Применение функции к аргументу

```
typeof(lambda) := "argType -> bodyType";
```

### 5.3 Применение функции к аргументу

Применение функции к аргументу (рис. 5.4) является, наряду с лямбда-абстракцией, основной конструкцией лямбда-исчисления. Правило типизации для этой конструкции выглядит аналогичным образом.

```
argType = new fresh type variable;
resType = new fresh type variable;
typeof(arg) := argType;
typeof(app) := resType;
typeof(fun) := "argType -> resType";
```

### 5.4 Ссылка на переменную лямбда-выражения

На рисунке 5.5 на переменную  $x$  (параметр лямбда-функции) имеется одна ссылка.

В общем случае ссылок на одну переменную (её упоминаний в теле функции) может быть несколько. Так как все эти упоминания переменной должны иметь одинаковый тип, логично приравнять тип ссылки типу объявления.

```
aType = new fresh type variable;
typeof(ref) := aType;
typeof(decl) := aType;
```

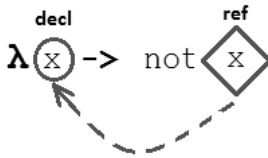


Рис. 5.5. Ссылка на переменную лямбда-выражения

Заметим, что подобная запись правила вывода означает, что вывод типов возможен в обе стороны: требования к типу объявления переменной влияют на тип ссылки на эту переменную и наоборот. Таким образом, все ссылки на эту переменную (все её упоминания) получат одинаковый тип — тип этой переменной.

## 5.5 Ссылка на объявление выражения или конструктора

Как было указано выше, объявления выражений или конструкторов (далее - объявления, *bindings*) имеют полиморфный тип, записываемый в аспекте `haskell.declaration`, то есть содержат, помимо собственно типа, список переменных типа, которые могут принимать произвольное значение. При этом ссылка на объявление будет иметь такой же тип, но некоторые переменные в нём могут быть конкретизированы. Заметим, что в разных ссылках на одно и то же объявление переменные могут быть конкретизированы по-разному. Подобное поведение можно осуществить, копируя переменные из полиморфного типа в общее пространство переменных. Новые переменные создаются привязанными к ближайшей вершине с потенциально полиморфным типом.

## 5.6 Объявление выражения

В объявлении выражения производится обратная процедура: переменные копируются из общего пространства переменных в список переменных полиморфного типа. При этом захватываются только те переменные, которые привязаны к вершине не выше рассматриваемой.

Замечание 4.7.1. О порядке выполнения правил и окончательно определённых типах. Так как объявления выражений являются частью видимого извне интерфейса модуля, то запросы об их типе могут приходить из различных источников. Поэтому тип объявлений должен быть определён корректно или ещё не определён – промежуточные вычисления не должны сказываться на этом типе. Для того, чтобы этого достичь, необходимо запускать копирование типов после того, как тип самого выражения окончательно определён. Окончательно определённым тип можно считать тогда, когда становится понятным, что все переменные, привязанные к рассматриваемой вершине или ниже, уже не будут конкретизированы. Фактически, окончательная определённость типа даёт возможность произвести захват всех свободных переменных, привязанных к поддереву, и поставить над ними квантор всеобщности. Конкретные алгоритмы, позволяющие установить момент, когда тип окончательно определён, и запустить правило, копирующее переменные, требуют отдельного исследования графов зависимостей вершин и правил.

## 5.7 Выражение `let`

Свойство выражения `let`, которое ещё не учтено другими правилами, заключается в том, что тип выражения `let` совпадает с типом выражения, стоящего после `in`.  $aType = \text{new fresh type variable}; \text{typeof}(\text{letExpr}) := aType; \text{typeof}(\text{innerExpr}) := aType;$

Замечание 4.8.1. О мономорфизме и окончательно определённых типах. Выражение `let` имеет одну примечательную особенность: внутри его объявлений могут находиться ссылки на переменные объемлющих функций. В результате может оказаться, что тип некоторого объявления внутри `let` зависит от переменных, привязанных к вышележащим вершинам. В этом случае тип этого значения этого объявления, после того как он будет признан окончательно определённым, будет зависеть от этих переменных. Это не помешает скопировать переменные, привязанные к рассматриваемому объявлению, в полиморфный тип, который также будет зависеть от переменных, привязанных к вышележащим вершинам. В конечном итоге эти переменные могут быть конкретизированы, тогда полиморфизма по ним не будет. Или, если они не будут конкретизированы, они скопируются в некоторый вышележащий полиморфный тип.

Пример (классический, из Haskell wiki).  $f\ x = \text{let } g\ y\ z = ([x,y],z) \text{ in } (g\ () , g\ \text{True}\ ())$  Данный пример корректно типизировать невозможно, так как

тип функции `g` связан (зависит от общей переменной типа) с типом переменной `x`. Из использования `g` в выражении `g()` следует, что переменная `x` имеет тип `()`, а из использования `g` в выражении `g True` следует, что переменная `x` имеет тип `Bool`. А разные алгебраические типы не совместимы друг с другом – `resolve`-процедура, пытающаяся провести унификацию, возвращает ошибку.

## 5.8 Выражение `case` и сопоставление с образцом

Для образцов правила типизации аналогичны правилам для выражений: ссылки на конструкторы копируют в локальный контекст тип конструктора, применение конструктора к аргументу (тоже образцу) аналогично применению функции к аргументу. Заметим, что количество аргументов у конструктора в образце должно совпадать с количеством аргументов в объявлении конструктора. И обеспечить это должна система типов (особенно, если иметь в виду расширение `GADTs`). Один из способов, которым это можно обеспечить, заключается в дополнительном присваивании образцу нужного алгебраического типа. В результате, если количество аргументов недостаточно, то этот тип будет конфликтовать в выведенным функциональным.

После того, как выведены типы образцов (окончательно определены), их можно скопировать на тип сравниваемого с образцами выражения. Если не сойдутся – ошибка.

Тип самого выражения `case` можно определить, приравняв его (присвоив одну и ту же переменную) к типам ветвей.

## Глава 6

# Описание системы типов языка Java

Описание языка Java на разработанном языке выполнено в соответствии со спецификацией языка [##].

### 6.1 Правила для констант

Для числовых и строковых литералов правила выглядят максимально просто: для каждого вида литералов указывается его тип. Приведём примеры литералов в Java и их типов.

2	int
0x7FFFFFFF	int
2l	long
0.0f	float
1.0	double
2.0E10	double
3.3D	double
'a'	char
'\u0036'	char
"Hello!"	String
false	boolean
null	nulltype

## 6.2 Выражения `.class`

В спецификации языка Java рассматривают 3 случая.

1. Если тип `C`, стоящий слева от `.class`, является классом, интерфейсом или массивом, то тип всего выражения будет иметь вид `Class<C>`.
2. Если тип `p`, стоящий слева от `.class`, является примитивным типом, то тип всего выражения будет иметь вид `Class<C>`, где `B` — результат применения `boxing conversion` к типу `p`.
3. Выражение `void.class` имеет тип `Class<Void>`.

В реализации каждому из перечисленных случаев соответствует отдельное правило. Единственное отличие от спецификации заключается в том, что случай 1 в реализации из соображений наглядности разбит на два варианта: для массивов и для классов с интерфейсами.

## 6.3 Правила для `this`

### 6.3.1 `unqualified this`

#чтение последнего элемента из контекста#

### 6.3.2 `qualified this`

#проверка, что указанный класс содержится в контексте#

## 6.4 Операции

Возможности разработанного языка позволяют выразить на нём отношения и преобразования над типами (`conversions`) таким же образом, как отношения описываются в спецификации языка [##]. Рассмотрим это на примере присваивания (`assignment conversion`).

В спецификации языка указывается что `assignment conversion` может состоять из:

- `identity conversion`,

- widening primitive conversion,
- widening reference conversion,
- boxing conversion, затем возможно widening reference conversion,
- unboxing conversion, затем возможно widening primitive conversion,

Далее отдельно указывается, что после вышеперечисленных преобразований возможно применение unchecked conversion, но только в том случае, если цепочка преобразований не будет содержать двух разных ссылочных типов, не связанных отношением подтипа.

Также отдельно оговаривается случай присваивания переменным типов `byte`, `Byte`, `short`, `Short`, `char`, `Character` констант типов `byte`, `short`, `char`, `int`.

Для записи на разработанном языке придётся слегка изменить формулировку, не меняя общего принципа изложения.

Объявляется отношение `java.assignmentConversion`, использующее политику приоритетов при разрешении конфликтов с перекрытием реализаций. Двумя аргументами операции являются типы левой и правой частей оператора присваивания, а результатом — специальная конструкция, указывающая, возможно ли присваивание, и, если возможно, то какая последовательность элементарных преобразований потребуется.

Затем объявляются реализации.

1. Если типы совпадают, то не потребуется никаких преобразований.
2. Если оба типа примитивные, то вызвать операцию `java.wideningPrimitiveConversion`.
3. Если оба типа ссылочные, то вызвать операцию `java.wideningReferenceConversionWithUnchecked`.
4. Если тип левой части примитивный, а правой — ссылочный, то вычислить результат операции `java.unbox` от типа правой части, затем вызвать операцию `java.wideningPrimitiveConversion`.
5. Если тип левой части один из типов `byte`, `Byte`, `short`, `Short`, `char`, `Character`, а правой — один из типов `byte`, `short`, `char`, `int`, то вернуть указание о том, что присваивание разрешается только для констант в определённом диапазоне.



6. Если тип левой части ссылочный, а правой — примитивный, то вычислить результат операции `java.box` от типа правой части, затем вызвать операцию `java.wideningReferenceConversionWithUnchecked`.

Операция `java.wideningReferenceConversionWithUnchecked` проверяет, возможно ли приведение типов с использованием операций `widening reference conversion`, а затем `unchecked conversion`, но только в том случае, если цепочка преобразований не будет содержать двух разных ссылочных типов, не связанных отношением подтипа.

Операция `java.box` переводит тип `int` в `Integer` и так далее.

Операция `java.unbox` является обратной к операции `box` (реализации обеих операций объявляются в виде общей расширяемой таблицы).

Операция `java.unbox` является обратной к операции `box` (реализации обеих операций объявляются в виде общей расширяемой таблицы).

##

# Заключение

Создан прототип движка, выполняющего правила в нужном порядке в соответствии с условиями, указанными в правилах. Реализован набор правил, осуществляющий вычисление и проверку типов для системы Хиндли-Милнера с *let*-полиморфизмом (упрощённая версия языка Haskell). При этом расширение Haskell, вводящее обобщённые алгебраические типы данных (GADTs) сводится к добавлению к базовому набору двух новых правил. Тип *Integer* и другие предопределённые в языке типы также вводятся в виде расширений языка, добавляющих новые правила системы типов.

Нетривиальной задачей является моделирование декларативной по своей природе системы Хиндли-Милнера в виде императивных правил. Однако следует заметить, что системе Хиндли-Милнера могут соответствовать различные алгоритмы вычисления типов, например, алгоритм *W* и алгоритм *M* [3]. При корректной программе эти алгоритмы приведут к одинаковому результату, но при наличии ошибок сообщения об ошибках будут разными. Различие между алгоритмами *W* и *M* заключается в порядке решения уравнений, и может быть выражено именно при императивном подходе.

Также создан набор правил, осуществляющий вычисление типов для упрощённой версии языка Java (без параметрического полиморфизма и перегрузки методов).

Основным критерием качества разрабатываемой системы описания правил является возможность описывать правила в тех же терминах, которые использовались при разработке языка. Для языка Haskell такими терминами являются уравнения над типами, а для языка Java – отношения и операции над типами. Соответственно, язык описания правил можно считать пригодным для описания систем типов Haskell и Java, если в

правилах для языка Haskell правила будут состоять в основном из кода, воспринимаемого как уравнения, а правила для Java будут оперировать отношениями и операциями над типами.

Разработана система языковых и библиотечных конструкций, позволяющая записывать алгоритмы вычисления и проверки типов в виде правил, структурно повторяющих пункты документации, по крайней мере для двух различных языков программирования. В качестве иллюстрации реализованы упрощённые версии алгоритмов вычисления и проверки типов языков Haskell и Java. Разработанная система после некоторой технической доработки может быть использована для описания языков в платформах разработки языков, таких как JetBrains MPS.

# Литература

- [1] Pierce, B. C. Types and Programming Languages / Benjamin C. Pierce.  
— Cambridge, MA, USA: MIT Press, 2002.