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 М. А. Буряков
Руководитель, ст. преп. 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. 32
Краткая характеристика всего документа, основной результат работы, список ключевых слов

Abstract

32 pages
Brief description of the work, main result, keywords

Оглавление

Список обозначений

АБГШ	аддитивный белый гауссовский шум
AST	абстрактное синтаксическое дерево
MPS	Metaprogramming system
GADT	Generalized algebraic datatypes

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 приписанными 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 поддерева. Аннотации типов, как правило, непосредственно указывают типы вершин AST.

Переменная типа (ти́повая переменная) -

Введение

Большая часть современных языков программирования имеют статическую систему типов для классификации конструкций, использующихся при описании программы (переменные, выражения, функции), по типами 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 в 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.

Требуется разработать систему 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 некоторм 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 [tapl, 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 [tapl, 8.2.2; 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 [tapl] 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

2. Системы типов в язык программирования.

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 их двух этапов: сначала каждой вершине AST программы приписывается некоторый тип, затем производится проверка корректности программы вместе с приписанными типами.

Замечание. Как правило, тип требуется не каждой вершине, а только выражениям языка. В дальнейшем будем это опускать, и, говоря, что вершине приписывается тип, будем подразумевать вершины, соответствующие выражениям.

Замечание. Этап вычисления типов теоретически является необязательным - программиста может все типы аннотировать явно. Но это оказывается крайне ее 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.

2.1. Простой вариант системы типов содержит конечное число типов, которые встроены в систему 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.

2.2. 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 объявиться 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.

2.3. Подтипы в их наиболее простом виде добавляют одно отношение частичного порядка над типами - 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. 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.

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 тоже 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 кассов можно создать несколько удобных языков, генерируемых в минималистичный язык/API движка типсистемы.

Глава 3

Язык описания систем типов

3.1 Парадигма языка

В качестве парадигмы языка, описывающего систему типов, предлагается использовать императивный подход. В сравнении с декларативным подходом программирования в ограничениях императивный подход предоставляет более простое описание для языков с простым алгоритмом типизации, но плохо структурированной иерархией типов, а именно такие языки чаще всего создаются в платформах для разработки языков.

Правила предлагается записывать в виде небольших фрагментов кода, выполняющего операции вида "прочитать тип вершины"или "записать тип вершины". Для вычисления типов правила запускаются в порядке, определяемом на основе метаинформации, прилагающейся к правилам. Например, правило может выполняться только при соблюдении некоторых условий. Если условия сводятся к тому, что к моменту выполнения уже должно быть выполнено некоторое другое правило, то это знание может быть использовано для предварительной сортировки правил. Метаинформация, прилагающаяся к правилу, частично может быть выведена из кода правила методом статического анализа, что представляет собой перспективу для дальнейших исследований.

3.2 Правила типизации

Каждое правило вывода типов объявляется для одного концепта (конструкции языка). В процессе вывода к каждой вершине этого концепта будет применён соответствующий экземпляр правила.

Также автором предложен способ описания систем типов с помощью локальных правил вывода. Локальное правило вывода применяется ко всем фрагментам AST, подходящим под шаблон - критерий применимости правила. Результатом применения правила к фрагменту AST является внесение изменений в репозиторий типов в отношении вершин из этого фрагмента. 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, необходимой 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.

Если многократная запись в репозиторий типов запрещена, то тип вершины AST считается вычисленным, как только произведена запись типа для этой вершины. Если же разрешена многократная запись, то тип вершины может считаться вычисленным, только если все правила, производящие запись, уже выполнены.

3.3 Репозиторий типов

Язык для виртуальной машины, выводящей типы, будет позволять производить произвольные вычисления с типами и другими объектами. Логично потребовать, чтобы все вычисления были чистыми. Все обращения к типизируемому коду, обращения состоянию машины и изменения состояния машины производятся только через специальные команды. Основной командой, изменяющей состояние машины, является команда присваивания типа вершине. Машина в качестве состояния хранит типы, присвоенные вершинам. Если этой вершине уже был присвоен какой-то тип, то эта ситуация рассматривается не как конфликт, а как конъюнкция

требований. Типы на данной этапе рассматриваются не как окончательные типы вершин, а как требования к этим типам. Поэтому когда требования пересекаются, вызывается специальная операция конъюнкции, возвращающая результирующее требование. Так как сами типы-требования определяются пользователем, то и процедура конъюнкции также должна быть определена пользователем. Заметим, что описанная схема принципиально отличается от создания уравнений тем, что в данном случае хранятся требования к одному типу, и правила работы с требованиями (конъюнкцию) определяет пользователь. После того, как все требования сформулированы и объединены, для каждой вершины выбирается конкретный окончательный тип.

Автором предложен обобщённый алгоритм вывода типов на основе 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. В зависимости от настроек запись типа может быть либо одноразовой, либо многоразовой. Многоразовая запись типов представляет собой моноид, определяющий начальное значение типа перед первой записью, а также операцию разрешения конфликта, который возникает при повторной записи типа. Например, для системы типов Хиндли-Милнера единицей моноида является свободная переменная типа, а операцией умножения — структурная унификация двух типовых деревьев.

3.4 Ти́повые переменные

3.5 Операции над типами

Некоторые языки программирования при описании их системы типов используют понятие отношений на типами. Наиболее распространённым отношением является отношение подтипа, однако в языке могут существовать 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 на типами содержит количество аргументов этой операции и тип результата. Реализаций же может быть много для разных шаблонов аргументов. Реализация n-арной операции содержит n шаблонов аргументов и функцию, непосредственно производящую вычисление.

Когда необходимо вычислить значение некоторой операции над типами для конкретных аргументов, среди всех реализаций этой операции выбираются те, шаблоны которых соответствуют аргументам. Если такая реализация ровно одна, то вызывается данная реализация. 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 моноида, которая предполагает возврат значения по умолчанию, если подходящих реализаций нет, и запуск всех реализаций с дальнейшей свёрткой, если подходящих реализаций несколько.

Глава 4

Правила вывода типов для языка Haskell

Каждое правило вывода типов объявляется для одного концепта (конструкции языка). В процессе вывода к каждой вершине этого концепта будет применён соответствующий экземпляр правила. Рассмотрим основные концепты упрощённого языка Haskell и правила для них.

4.1 Объявление алгебраических типов

#схема# Здесь нужно выставить типы конструкторам. Конструктор с n аргументами будет иметь функциональный тип с n аргументами, причём аргументы и результат копируются непосредственно из объявления.

Замечание 4.1.1. Так как алгебраические типы могут иметь параметры, то конструкторы могут иметь полиморфный тип атваризатрав. 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 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.

4.2 Лямбда-выражение

#схема#: лямбда-выражение lambda, его аргумент arg, тело функции body. Правило на псевдокоде:

argType = new fresh type variable; bodyType = new fresh type variable; typeof(arg) := argType; typeof(body) := bodyType; typeof(lambda) := "argType -> bodyType";

Замечание 4.2.1. Цикл жизни переменных. Переменные типов могут использоваться для двух целей [tapl, 22.2]: представление полиморфных типов и использование в промежуточных вычислениях в методах type inference. Если не различать переменные, предназначенные для той или иной цели, то можно получить важное преимущество: при создании переменной не нужно указывать, для какой цели она будет использована (будет ли её значение вычислено, или над ней появится квантор всеобщности). Это весьма важно, так как при создании переменной часто сложно знать её дальнейшую судьбу. 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. Если это происходит, то во всех местах, где встречается эта переменная, она может быть заменена на своё значение (что и будет сделано немедленно или отложенно - в зависимости от деталей реализации), поэтому все переменные можно считать пока ещё свободными. Если же значение переменной так и не будет определено, значит она может принимать любые значения (важно как можно быстрее понять, что переменная осталась свободной). В конце текущего этапа вычислений переменная будет перемещена (в реализации - скопирована) в список переменных некоторого полиморфного типа. Отметим, что перемещение переменных, оставшихся свободными является особенностью конкретной реализации.

Замечание 4.2.2. Переменные и type kinds. При создании переменной уже известен её type kind. Для поддержки type kinds необходимо при создании переменной указывать её type kind, чтобы можно было вычислить type kind для любого типа, в том числе содержащего переменные.

Замечание 4.2.3. Привязка переменных к контекстам. Принцип локальности подсказывает, что переменные типов 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 переменная не была разрешена, она становится частью полиморфного типа некоторой вершины, то логично привязывать её к той вершине, частью полиморфного типа которой она может стать.

Замечание 4.2.4. Присваивание типов и resolve-процедура. Наиболее важной частью правил являются присваивания типов. При этом правило может присваивать тип не только той вершине, для которой создан конкретный экземпляр правила, но и вершинам в её поддереве, к которым заранее указан путь из этой вершины (запретить присваивание типов произвольно найденным вершинам предложено в [#нет публикации#, Elovkov D.]). В результате одной и той же вершине тип может присваиваться несколько раз различными правилами. Если это происходит, запускается resolve-процедура, вычисляющая конъюнкцию требований к типу.

Resolve-процедура принимает на вход вершину и два типа: старый, который уже был присвоен вершине, и новый, попытка присваивания которого вершине привела к вызову resolve-процедуры. В общем случае resolve-процедура должна задаваться в виде набора правил, но для данной конкретной задачи её можно задать единовременно, так как здесь resolve-процедура представляет собой процедуру унификации. Унификация двух алгебраических типов для различных типов возвращает ошибку, а для одинаковых запускает унификацию для соответствующих параметров. Аналогично для функциональных типов и применений типов. Если вызвана унификация свободной переменной и не 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 через другую преследует несколько целей. Во-первых, при дальнейшем добавлении классов типов новая переменная будет совмещать в себе классы обеих исходных переменных. Во-

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

4.3 Правило для всех выражений

Для каждого выражения expr применим правило: aType = new fresh type variable; typeof(expr) := aType;

Это правило позволяет правильно вывести тип тогда, когда других правил нет. Например, для объявления undefined = undefined В остальных случаях это правило не влияет принципиально на вывод типов.

Замечание 4.3.1. Type kinds и resolve-процедура. Вышеописанное правило может быть полезно при добавлении проверок type kinds. Так как известно, что тип любого выражения должен иметь kind *, то это можно задать непосредственно при создании переменной. Дополнительно потребуется ввести проверку совпадения type kinds у аргументов resolve-процедуры.

4.4 Применение функции к аргументу

#схема#: выражение применения функции к аргументу app, функция fun, аргумент arg. argType = new fresh type variable; resType = new fresh type variable; typeof(arg) := argType; typeof(app) := resType; typeof(fun) := "argType -> resType";

4.5 Ссылка на переменную лямбда-выражения

#схема#: ссылка ref на переменную decl. Так как все упоминания переменной должны иметь одинаковый тип, логично приравнять тип ссылки типу объявления. aType = new fresh type variable; typeof(ref) := aType; typeof(decl) := aType;

4.6 Ссылка на объявление выражения или конструктора

Как было указано выше, объявления выражений или конструкторов (далее - объявления, bindings) имеют потенциально полиморфный тип, 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.

4.7 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 из 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.

4.8 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 = new fresh type variable; typeof(letExpr) := aType; typeof(innerExpr) := aType;

Заменчание 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 = let g y z = ([x,y],z) in (g ( ) ( ), g 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.

4.9 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 ти-

пов (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 туже переменную) к типам ветвей.

Глава 5

Описание системы типов языка Java

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 алогритмы 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.