Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Бакрадзе Лиана Георгиевна

Вывод типов для языка R в интегрированной среде разработки IntelliJ IDEA

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
к. ф.-м. н. Булычев Д. Ю.

Рецензент:
Тузова Е. А.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Liana Bakradze

Type inference for language R in the IntelliJ IDEA integrated development environment

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
Dmitri Boulytchev

Reviewer:
Ekaterina Tuzova

Saint-Petersburg
2015

Оглавление

Введение

В настоящее время многие организации являются источниками большого количества данных. Это могут быть данные коммерческих компаний о продажах, статистика использования программных продуктов и сервисов, финансовые и медицинские данные или информация, получаемая с мощных астрономических телескопов.

Все эти данные нуждаются в дальнейшей обработке и анализе. С этой целью используется язык R [11, 13, 2], который является стандартом для программного обеспечения, занимающегося статистической обработкой и анализом данных. Этот язык предназначен для интерактивного использования, поэтому он позволяет, например, указать только отличающиеся от параметров по умолчанию значения при вызове функции или обратиться к элементам коллекции, указав только часть их имени (*partial matching*). В то же время R является популярным языком высокого уровня. На этом языке написано значительное количество программ, выполняющих сложные статистические вычисления или строящих различного вида графики. В "The Comprehensive R Archive Network"[1] и Bioconductor[2], самых популярных репозиториях с библиотеками на языке R, на сегодняшний день собрано более 7000 библиотек.

Язык R является динамически типизированным, то есть все проверки типов осуществляются только в момент исполнения программы. Динамическая природа языка R обеспечивает программам, написанным на этом языке, большую гибкость. В то же время это означает, что программист должен запустить свою программу, чтобы узнать о возможных ошибках.

Статический анализ позволяет узнать о наличии в программе ошибок без её запуска. *Вывод типов* является важной частью статического анализа и позволяет, например, проверить соответствие типов переданных в функцию значений типам её параметров. Кроме того, инстру-

[1] http://cran.r-project.org

[2] http://www.bioconductor.org

менты статического анализа могут быть встроены в интегрированные среды разработки (*Integrated Development Environment*). Информация, получаемая с помощью таких встроенных статических анализаторов, используется также в различных преобразованиях исходного кода программ (*refactorings*) и в автодополнении (*completion*).

Платформа IntelliJ, разработанная компанией JetBrains[3], позволяет создавать кросс-платформенные интегрированные среды разработки, включающие текстовый редактор, интеграцию с системами контроля версий, отладчик и инфраструктуру для поддержки различных языков (синтаксический и лексический анализаторы, подсветку синтаксиса). На базе этой платформы, например, уже созданы среды разработки для Java (*IntelliJ IDEA*[4]), Python (*PyCharm*[5]), JavaScript (*WebStorm*[6]). Поддержку других языков программирования можно реализовать на базе платформы IntelliJ, создав подключаемый модуль (*плагин*). Также платформа позволяет легко встраивать инструменты статического анализа, которые называются инспекциями исходного кода (*inspections*).

На базе платформы IntelliJ для другого динамического языка Python[7], который, как и R, широко используется в сфере анализа данных, создана среда разработки PyCharm. В эту среду встроена система типов, которая используется в различных инспекциях и преобразованиях исходного кода, а также в автодополнении. Однако для языка R попыток предложить подобную систему типов до сих пор сделано не было. Поэтому даже самые популярные среды разработки для языка R (RStudio[8], StateEt[9]) не имеют никаких встроенных инструментов статического анализа.

В настоящее время в компании JetBrains разрабатывается плагин для языка R, подключаемый к IntelliJ IDEA. В этом плагине уже реализована функциональность по подсветке синтаксиса исходного кода и

[3]https://www.jetbrains.com
[4]https://www.jetbrains.com/idea/
[5]https://www.jetbrains.com/pycharm/
[6]https://www.jetbrains.com/webstorm/
[7]https://www.python.org
[8]http://www.rstudio.com
[9]http://www.walware.de/goto/statet

навигации по коду, строится синтаксическое дерево и существует возможность запуска программ на языке R. Также ведётся разработка отладчика и компоненты, ответственной за интерактивное отображение документации к функциям и установку пакетов. Цель данной работы заключается в том, чтобы с помощью введения системы типов улучшить поддержку для языка R в этом плагине.

Для достижения данной цели были поставлены следующие задачи:

- предложить систему типов для языка R;
- в плагине для интегрированной среды разработки IntelliJ IDEA реализовать подсистему, которая производит анализ исходного кода без его выполнения с целью получения и дальнейшего использования информации о типах;
- для апробации разработанной подсистемы реализовать инспекцию, выявляющую несоответствие реальных типов ожидаемым;
- протестировать реализованную инспекцию на реальных проектах на языке R.

1. Статические анализаторы для языка R

В [16] рассматриваются различные сложности и ограничения, возникающие при статическом анализе языка R, а также описывается реализованная на языке R библиотека codetools[10]. Данная библиотека используется в "The Comprehensive R Archive Network" (*CRAN*), самом популярном репозитории с библиотеками на языке R, для поиска ошибок при выкладывании новых библиотек в репозиторий.

Библиотека codetools позволяет определять следующие дефекты в программах на языке R:

- использование переменных и функций, которые нигде не были определены;
- определение переменных, которые нигде не используются;
- несоответствие вызовов функций их определению.

Проверять на наличие дефектов можно как отдельные функции, так и целые библиотеки. Также codetools обладает широкими возможностями для настройки типов дефектов, о которых пользователь будет получать предупреждения.

К сожалению, дефекты, которые находит данная библиотека, в большинстве случаев не приводят к ошибкам при исполнении программы, а являются особенностями реализации. Тем не менее, вызов функции без необходимого параметра, который приведёт к остановке программы, не будет определён как дефект библиотекой codetools.

В [4] упоминается ParseR, инструмент для статического анализа языка R. Он написан на Java и является частью большого проекта, в котором на основе большого набора программ на языке R исследовалось применение различных возможностей языка.

ParseR позволяет проводить четыре различных статических анализа для программ на языке R:

- нахождение переменных, ссылающихся на одни и те же объекты;

[10]http://cran.r-project.org/web/packages/codetools/index.html

- переопределение объекта из более широкой области видимости;
- нахождение переменных, которые нигде не используются;
- нахождение конфликтов области видимости.

В реализации также есть возможность подключить собственный статический анализатор, тем не менее не известно, чтобы этот инструмент использовался где-либо, помимо проекта, для которого он был написан.

2. Подходы к выводу типов в динамических языках

Под *выводом типов* (*type inference*) понимается процесс получения типов переменных путём анализа их использования в исходном коде. Вывод типов достаточно давно применяется в функциональных языках. Хиндли[6] и Милнер[10] независимо друг от друга предложили метод для вывода типов во время компиляции, который используется в таких функциональных языках, как ML[11], Haskell[12], OCaml[13].

Языки с динамической типизацией обычно медленнее языков со статической, так как на каждую операцию во время исполнения программы приходится проверять правильность типов. Этим фактом вызвано желание многих исследователей добавить в динамические языки систему типов, чтобы улучшить производительность.

В данной работе будет рассмотрено несколько существующих систем получения информации о типах для популярных динамических языков. Как уже было отмечено ранее, для языка R попыток реализовать такую систему сделано не было.

2.1. Введение статической системы типов

В [14] описывается система Starkiller, предназначена для статического вывода типов выражений в программах на языке Python. Эта система была разработана, чтобы облегчить генерацию нативного кода по программам на языке Python. Алгоритм, который использует Starkiller для вывода типов, основан на алгоритме, предложенном в [1] для языка Self.

В ходе своей работы используемый в Starkiller алгоритм вывода типов строит граф потока данных, который моделирует поведение всех значений во входной программе. Каждой переменной или выражению во входной программе ставится в соответствие вершина графа. С каж-

[11] http://en.wikipedia.org/wiki/ML_(programming_language)
[12] https://www.haskell.org
[13] http://caml.inria.fr

дой вершиной ассоциируется множество типов, которые может принимать выражение во время исполнения. В самом начале все такие множества пусты, за исключением тех, которые содержат константы: в множества типов для таких вершин записывается тип соответствующей константы. Вершины, в свою очередь, соединены ограничениями, которые моделируют поток данных между ними. Ограничение — это направленное ребро, которое означает, что в множестве типов входящей вершины должны содержаться по крайней мере типы из исходящей вершины. На Рис.1 показан пример построенного графа для простой программы.

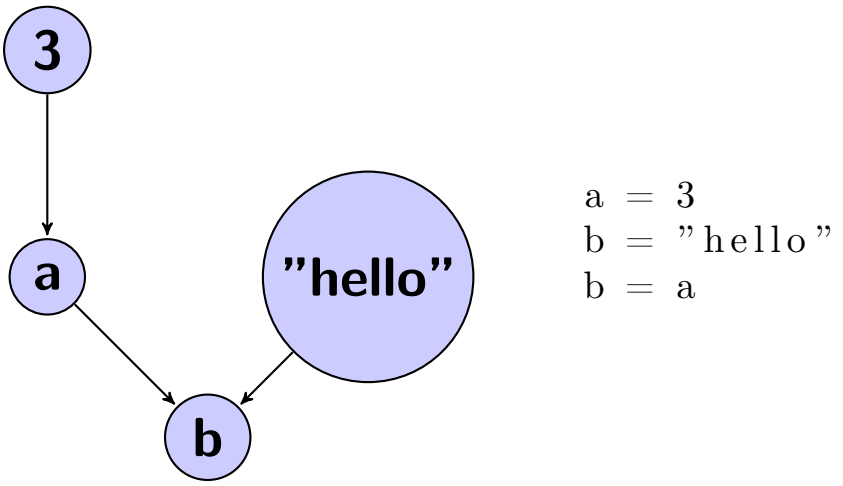


Рис. 1: Пример построения графа для программы

Используемый в Starkiller алгоритм нечувствителен к потоку управления, что означает, что типы присваиваются переменным, а не конкретным обращениям к ним.

PyPy[14] — реализация интерпретатора для языка Python, написанная на подмножестве этого языка под названием RPython. Основным преимуществом этого интерпретатора является его производительность. В [8, 12] описывается разработанная для RPython статическая система типов. Для этой системы типов приводится описание с использованием методов формальной логики: грамматика для всех типов и правила переписывания для их вывода.

К сожалению, RPython накладывает очень серьёзные ограничения

[14]http://pypy.org

на язык Python, отказываясь от многих его динамических возможностей:

- переменные не могут менять тип в ходе исполнения;
- сложные типы должны быть гомогенными, то есть все коллекции должны содержать элементы только одного типа;
- функции не могут быть определены внутри других функций;
- запрещено использование глобальных переменных;
- запрещено множественное наследование;
- запрещено использование метапрограммирования.

В [15] отмечается, что последствием гибкости, которую предоставляют динамические языки программирования, является отсутствие возможности статически обнаружить такие ошибки, как передача параметров неправильного типа при вызове функции. Была сделана попытка решения этой проблемы для языка Ruby[15]. Авторами [15] была реализована система для статического вывода типов для этого языка — *Diamondback Ruby*. Типы в данной системе выводятся с помощью генерации и последующего разрешения различных ограничений на них. Ограничения, например, могут указывать на наличие определённых методов у класса. Реализованная система была протестирована на нескольких десятках небольших проектов на языке Ruby и помогла обнаружить несколько ошибок.

2.2. Динамические методы

К сожалению, существующие системы для статического получения информации о типах для динамических языков показывают плохие результаты на программах, в которых активно используется метапрограммирование: изменение структуры классов во время исполнения или

[15]https://www.ruby-lang.org/en/

интерпретация текстовых строк как исходного кода. Можно выделить группу методов, которые основаны на идее динамического получения информации, необходимой для вывода типов в программах, использующих динамические возможности метапрограммирования.

В [3] предложен подход динамического получения информации о типах для динамического языка Ruby. Идея этого подхода состоит в следующем:

1. Всем сущностям, для которых выводятся типы (поля классов, аргументы и возвращаемое значение методов), ставятся в соответствие специальные объекты-обёртки.

2. Каждый раз, когда поток исполнения обращается к любой из этих сущностей, на основе её типа генерируются и запоминаются внутри обёрток специальные ограничения. Такие ограничения могут, например, указывать, от какого класса должен наследоваться класс аргумента метода, или содержать информацию о наличии у класса опеделённого метода.

3. После того, как исполнение программы завершилось, для всех сущностей подбирается тип, удовлетворяющий всем сохранённым ограничениям. Если такой тип найти не удалось, то генерируется ошибка.

Кроме теоретического обоснования корректности предложенного подхода, авторы [3] также представили его реализацию — систему *Rubydust*. Эта система была протестирована на нескольких небольших проектах на языке Ruby и помогла обнаружить одну реальную ошибку.

В [5] описывается похожий динамический подход к получению информации о типах для языка Squeak[16], основанный на пошаговом исполнении участков кода.

Надо отметить, что из-за использования метапрограммирования описанные динамические методы влияют на производительность программ, поэтому получать информацию с их помощью следует тогда,

[16]http://www.squeak.org/

когда время работы несущественно: во время отладки программы или выполнения тестов. Среди недостатков таких методов также нужно выделить их зависимость от конкретного пути исполнения программы (если в качестве источника информации о типах используются тесты, то от полноты тестового покрытия), поэтому выведенные с их помощью типы могут получиться излишне конкретными и не учитывать некоторых случаев.

Динамические методы могут быть встроены в интегрированные среды разработки. Например, PyCharm, среда разработки для языка Python, использует полученную по время отладки приложения информацию в системе для вывода типов.

2.3. Эвристические методы

Необычный подход к выводу типов для динамического языка Python предлагается в проекте *MINO* [17]. Авторы этого проекта отмечают, что большинство существующих систем вывода типов для языка Python используют методы формальной логики, однако такой подход обычно налагает ограничения на множество используемых конструкций типизированного языка и требует некоторых предположений о типизируемых программах.

В проекте MINO предлагается вместо методов формальной логики использовать машинное обучение. При таком подходе задача типизации программы рассматривается как задача классификации с многими классами, где каждому выражению требуется присвоить один из нескольких типов. Идея данного подхода основана на том, что информацию о типах в программе можно извлечь не только из её формальной семантики. В качестве простого примера можно привести использование имени i для переменных, которые являются счётчиками циклов и, следовательно, являются целыми числами.

К сожалению, с помощью такого подхода могут быть выведены только самые простые типы: авторы проекта ограничиваются примитивными типами и общим типом для всех объектов, что не позволяет различать типы объектов разных классов.

3. Использование модулей-заглушек

Преимуществом популярных динамических языков (Python, Ruby, JavaScript, R) является большое количество библиотек: к примеру, в PyPi [17], крупнейшем репозитории библиотек на языке Python, собрано более 55000 библиотек. Чтобы облегчить использование сторонних библиотек, средам разработки необходимо знать не только точные определения содержащихся в них функций, но и типы аргументов и тип их возвращаемого значения.

Извлечение такой информации — непростая задача, несмотря на возможности инструментов статического анализа. Во-первых, внутри библиотечных функций может использоваться метапрограммирование, что, как уже отмечалось в разделе 2.2, негативно сказывается на качестве результатов статического анализа. Во-вторых, очень часто в стандартной библиотеке и в сторонних билиотеках, которые занимаются обработкой изображений, видео или звука, некоторые функции реализуют на языке C ради большей производительности, делая таким образом тела этих функций недоступными для анализа (эта проблема известна также как *foreign code interaction*). В-третьих, даже из документации бывает сложно понять типы аргументов и возвращаемого значения функции, потому что для большинства динамических языков нет общепринятых методов для записи типов.

Для решения этой проблемы для некоторых динамических языков было предложено создавать специальные модули-заглушки, которые представляют собой определения функций, дополненные информацией, из которой инструменты статического анализа могут вывести типы этих функций.

DefinitelyTyped[18] — репозиторий, содержащий модули-заглушки для нескольких сотен популярных JavaScript[19]-библиотек. Эти модули написаны на языке TypeScript[20] — диалекте JavaScript, добавляющем сре-

[17] https://pypi.python.org
[18] http://definitelytyped.org
[19] https://en.wikipedia.org/wiki/JavaScript
[20] http://www.typescriptlang.org

ди прочего систему типов. Типы указываются прямо в коде посредством опциональных аннотаций, также можно выносить определения объектов и функций с указанием типов в отдельные файлы (*type defenitions*). При этом любая программа на JavaScript является корректной программой на TypeScript. Наличие таких модулей позволяет получить очень хорошую поддержку в средах разработки при использовании библиотек на языке JavaScript, для которых есть файлы с определением типов в репозитории DefenitelyTyped (как это делается, например, в Misrosoft Visual Studio[21] и в WebStorm[22]).

PyCharm, среда разработки для языка Python, использует свой собственный репозиторий с модулями-заглушками (*skeletons*)[23]. Эти модули представляют собой синтаксически правильные определения функций на языке Python. Типы в специальном виде, также предложенном разработчиками PyCharm, указываются в комментариях с документацией в формате Sphinx [24]. Пользователи также имеют возможность самостоятельно подключать такие модули непосредственно в среде разработки.

Для языка Python существует также другая инициатива по сбору модулей-заглушек, которые могут быть использованы в инструментах статического анализа, — проект typeshed[25]. Пока в репозитории этого проекта содержатся модули-заглушки для немногим более 20 библиотек, однако в настоящее время проект активно развивается. В качестве источника информации о типах используются опциональные аннотации типов, описанные в одном из стандартов языка Python[26].

Во всех упомянутых репозиториях собраны модули-заглушки, которые вручную написаны либо разработчиками инструментов статического анализа, либо пользователями библиотек, либо их создателями.

[21]https://www.visualstudio.com/
[22]https://www.jetbrains.com/webstorm/
[23]https://github.com/JetBrains/python-skeletons
[24]http://sphinx-doc.org
[25]https://github.com/JukkaL/typeshed
[26]https://www.python.org/dev/peps/pep-3107/

4. Система типов для языка R

Система типов состоит из множества типов и правил их назначения различным конструкциям языка, то есть правил вывода типов.

Рассмотренные в разделе 2.1 статические системы типов для различных динамических языков предназначались, прежде всего, для использования в интерпретаторе в целях улучшения его производительности. Поэтому такие системы типов не должны отвергать корректные программы. Обычно это свойство системы типов доказывается на основе формальной записи системы типов с помощью методов формальной логики.

Предлагаемая в данной работе система типов для языка R предназначена для использования в интегрированной среде разработки, то есть на исполнение программ она никак не влияет. Поэтому, если данная система типов отвергнет какую-то корректную программу, это не будет критичным недостатком среды разрабоки.

Рассматриваемая система типов должна быть, прежде всего, гибкой. В неё должно быть легко вносить исправления, вызванные сообщениями об ошибках от пользователей или изменениями в новых версиях языка. В реальности правила для вывода типов меняются каждый раз, когда обнаруживается какой-то новый вид ошибки, которую следует показывать пользователю среды разработки.

Система типов состоит из множества типов и правил их вывода. В связи с описанными особенностями сред разработки в данной работе не приводится формальная запись правила вывода типов, а рассматриваются только предлагаемые в данной работе типы для языка R и контекст их использования.

Гомогенные коллекции

Гомогенные коллекции являются основным типов данных языка R и называются векторами (*vector*). Они могут быть параметризованы одним из атомарных типов языка R: *complex, numeric, character, integer, raw, logical, null*. К примеру, строковый литерал ”hello” с точки зрения

языка R является вектором единичной длины, параметризованным типом *character*.

На атомарных типах определён линейный порядок: если создать вектор, содержащий два числа и строку, то все элементы будут трактоваться как строки, так как тип character имеет больший приоритет, чем числовые типы.

К элементам векторов можно обращаться по индексу, получая элемент того типа, которым вектор параметризован.

В предложенной системе типов отождествляется вектор с атомарным типом, которым он параметризован. Таким образом, в системе типов содержится семь типов, соответствующих атомарным типам.

Гетерогенные коллекции

Гетерогенные коллекции в языке R представляются списками (*list*). Списки представляют собой коллекцию атрибутов различного типа, поддерживающих обращение либо по индексу, либо по необязательному имени. Структура списка может быть изменена во время исполнения программы.

Рассмотрим такой пример:

```
x <- foo()
x$bar <- "buz"
```

Вторая инструкция в этом примере превращает `x` независимо от его типа в список, содержащий атрибут `bar`. Тем не менее, функция `foo` могла вернуть список, тогда у x останутся и остальные атрибуты списка, о которых может не быть информации, то есть необходим тип списка, означающий, что известна только часть его структуры.

Таким образом, в предложенной системе типов есть два различных типа для списков: тип с частичным знанием структуры списка и тип с полным знанием структуры списка.

S3 и S4 классы

В языке R существует две объектные модели: S3 и S4 классы. Модель S3 считается устаревшей, тем не менее согласно исследованиям [4] она до сих пор используется гораздо чаще. В предложенной системе типов поддерживаются обе этих модели.

S3-классы на самом деле представляют собой метки, которые используются для определения, какая именно реализация обобщённой функции будет использована для объекта. Ниже приведён пример создания и использования S3-класса:

```
print.A <- function(x) ...
print.B <- function(x) ...
x <- foo()
class(x) <- "A"
print(x)
```

В данном примере определяются две различных реализации обобщённой функции `print` — для класса A и для класса B. Затем переменной `x` присваивается класс A, поэтому при вывозе функции `print` для `x` будет использована реализация `print.A`.

В предложенной системе типов для S3-классов нет отдельного типа. S3-классы поддерживаются с помощью возможности добавить к любому из существующих типов метку.

S4-классы являются аналогами классов языка C++. При создании классов указываются его атрибуты вместе с их типами, причём при записи значения в атрибут класса во время исполнения проверяется, что тип этого значения соответствует типу атрибута, указанному при создании класса. Для S4-классов возможно наследование, в том числе множественное. В предложенной системе типов для S4-классов есть отдельный тип.

Функции

Основная цель системы типов — находить ошибки в программе. В языке R возвращаемое значение функций часто зависит от точного значения строкового или числового литерала, который передаётся в качестве параметра. Ошибки в таких литералах хотелось бы находить с помощью системы типов.

Рассмотрим пример:

```
x <- vector("integr", 3)
```

В данном примере при помощи функции `vector` создаётся вектор длины три. При этом тип вектора, который вернёт эта функция, передаётся в виде строки. В примере в этом литерале допущена опечатка, и найти такого рода ошибку в большой библиотеке — не самая тривиальная задача, поэтому хотелось бы показывать их в среде разработки.

Надо заметить, что функция `vector` является одним из часто используемых конструкторов типа вектор, основного типа данных языка R, поэтому необходимо понимать, вектор какого именно атомарного типа она возвращает. Такие функции описываются в теории зависимых типов [9].

В нашем случае тип функции содержит не только типы её параметров и тип возвращаемого значения, но и множество правил, которые задают зависимость типа возвращаемого значения от типов и точных значений параметров.

Тип-объединение

Рассмотрим пример:

```
if (x) {
    y <- 1
} else {
    y <- "str"
}
sin(y)
```

В данном примере существует путь исполнения программы, при котором возникнет ошибка, так как в функцию `sin` будет передана строка. Для обнаружения таких ошибок в предложенной системе типов были введены типы-объединения (*union types*)[7].

Тип-объединение представляет собой контейнер, содержащий множество типов, которые может принимать переменная во время исполнения.

Универсальный тип

Ввиду всех упомянутых в данной работе проблем, возникающих при выводе типов для динамических языков, для некоторых встречающихся в программе выражений вывести тип нельзя. Таким выражениям присваивается универсальный тип (*top*), который означает, что множество значений этого выражения ничем не ограничено.

Пустой тип

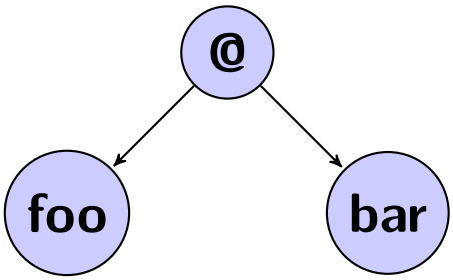
В некоторых случаях возможно точно определить, что во время исполнения возникнет ошибка. Например, при попытке обратиться к несуществующему атрибуту S4-класса. Таким выражениям присваивается пустой тип (*bottom*), который означает, что у данного типа нет представителей.

5. Реализация

5.1. Система вывода типов для языка R

Для получения информации о типах в подключаемом модуле для платформы IntelliJ, который осуществляет поддержку языка R, реализована специальная подсистема. Данная подсистема написана так же, как и платформа IntelliJ, на языке Java.

Подключаемый к IntelliJ IDEA модуль для каждого файла с исходным кодом на языке R строит абстрактное синтаксическое дерево. На вход системе для вывода типов приходит запрос на получение типа определённого узла этого дерева. В зависимости от того, какой конструкции языка соответствует этот узел, тип этого узла выводится из типов других узлов этого дерева.



foo@bar

Рис. 2: Пример абстрактного синтаксического дерева для выражения

Например, для вывода типа выражения `foo@bar` (это выражение означает обращение к атрибуту S4-класса) на Рис.2 будет сделан запрос на получение типа его потомка — узла `foo`. Если этот тип — S4-класс, то система вернёт тип атрибута `bar`, иначе вернёт пустой тип.

Вывод типов в среде разработки должен быть чувствителен к потоку управления, так как подсказки, которые получает пользователь от среды разработки, зависят от конкретного места в коде. Таким образом, рассматриваемая система вывода типов должна выводить типы для каждого конкретного обращения к переменной. Для решения этой задачи был реализован отдельный статический анализатор.

Данный анализатор находит все достигающие определения

(`reaching definitions`)[27] для данного обращения к переменной: записи значений в эту переменную, такие, что для них существует путь исполнения программы, при котором эта переменная не перезаписывается до этого обращения к ней. В таком случае типом переменной в этом обращении к ней будет тип-объединение всех типов из построенного множества достижимых определений.

Данный анализатор не только даёт чувствительность к потоку управления, но и позволяет определить необязательные параметры функций, так как для них существует путь исполнения, при котором для всех обращений к переменной с тем же именем в множестве достигающих определений не содержится объявления параметра функции.

Надо заметить, что рассматриваемая в данной работе система должна по-разному вести себя в отношении пользовательских и библиотечных функций. Для библиотечных функций из-за вопросов производительности нецелесообразно проводить глубокий анализ их тела, к тому же из-за проблемы, описанной в разделе 3, их тело может быть недоступным для анализа.

Описанная проблема решалась при помощи специальных модулей-заглушек, также описанных в разделе 3. В данной работе такие модули представляют собой определения функций, дополненные комментариями, содержащими специальные аннотации.

Так как для языка R нет общепринятых методов записи даже простых типов, формат аннотаций также был предложен в данной работе. Такие аннотации являются опциональными. В них можно указывать все типы, описанные в разделе 4. Всего существует четыре различных видов аннотаций:

1. `@type` позволяет явно указать тип параметра функции.

2. `@return` служит для явного указания возвращаемого значения функции.

3. `@optional` позволят перечислить необязательные параметры функции.

[27] http://en.wikipedia.org/wiki/Reaching_definition

4. `@rule` используется для задания правил, описанных в разделе 4.

Как уже отмечалось ранее, обычно модули-заглушки пишутся вручную, однако в данной работе они были сгенерированы. Генерация этих модулей происходит в несколько этапов:

1. Специальный скрипт на языке R получает список установленных библиотек.

2. Для каждой из библиотек этот же скрипт получает список содержащихся в ней функций.

3. Для каждой из функций этот скрипт получает её определение, документацию к ней и примеры использования, которые также можно получить динамически.

4. Далее производится анализ документации на предмет указания в ней информации о типах параметров и возвращаемого значения функций. На основе этой информации генерируются аннотации для этой функции.

5. Если в определении функции содержится её тело, то с помощью отдельных статических анализов определется тип возвращаемого значения из тела функции и необязательные параметры. Полученная информация также добавляется к сгенерированным ранее аннотациям.

6. Правильность сгенерированных аннотаций проверяется с помощью примеров. Если аннотации проходят проверку, то они вместе с определением функции добавляются в модуль-заглушку для данной библиотеки.

7. Полученные модули-заглушки добавляются в среду разработки в виде подключенной библиотеки.

Сгенерированные модули-заглушки для библиотек, входящих в стандартную поставку интерпретатора, поставляются вместе с подключаемым модулем, тем не менее они могут быть сгенерированы отдельно

и для сторонних библиотек. Для проверки работоспособности данного метода генерации модулей-заглушек были сгенерированы такие модули для стандартной библиотеки языка R (*base*). В результате были сгенерированы аннотации для 45% функций, содержащихся в этой библиотеке.

Благодаря тому, что в качестве источника информации о типах функций в библиотеках выступают аннотации, вывод типов библиотечных функций никак не влияет на производительность системы получения информации о типах.

Вывод типов является дорогостоящей операцией, поэтому выведенные типы для всех выражений кэшируются.

5.2. Встроенный статический анализатор

Платформа IntelliJ позволяет легко встраивать статические анализаторы для разных языков. Такие встроенные статические анализаторы называются инспекциями (*inspections*)[28].

Для апробации разработанной системы для вывода типов в плагине для платформы IntelliJ, осуществляющем поддержку для языка R, была реализована инспекция. Данная инспекция сообщает пользователю о несоответствии ожидаемых типов реальным.

Испекция обходит все выражения и проверяет их тип. Если тип какого-то из выражений оказывается ненаселённым (пустой тип), то также показывается соответствующее сообщение об ошибке.

При обходе вызовов функций, имеющихся в построенном абстрактном синтаксическом дереве, инспекция находит определение вызываемой функции и посылает системе для вывода типов запрос на получение типа этой функции, также отправляются запросы на получение типов всех передаваемых в функцию значений. Все полученные типы передаются специальной компоненте, которая проверяет их соответствие друг другу. В случае несоответствия инспекция генерирует сообщение об ошибке и показывает его пользователю.

[28]https://www.jetbrains.com/idea/help/code-inspection.html

6. Тестирование

Тестирование описанной в разделе 5.2 инспекции проводилось в два этапа.

Сначала были вручную написаны тестовые программы на языке R, содержащие все основные конструкции языка R в различных комбинациях. В части тестовых программ были специально допущены и размечены вручную ошибки. Таким образом, было собрано специальное тестовое множество, на котором можно отслеживать работоспособность инспекции после внесения изменений в саму инспекцию или систему для вывода типов.

Затем инспекция тестировалась на реальных проектах на языке R, чтобы убедиться в отсутствии большого количества ложных срабатываний, то есть неправильно найденных ошибок в корректном коде. В тестовое множество было включено несколько популярных проектов на языке R, взятых с Github [29], веб-сервиса для хранения программных проектов. Также в тестовое множество вошло несколько недавно обновлённых библиотек с CRAN[30], крупнейшего репозитория для библиотек на языке R.

Для всех включенных в тестовое множество проектов инспекция была запущена для всех файлов с исходным кодом на языке R.

В таблице 1 представлены результаты тестирования. Под “ошибками” понимается наличие такого пути исполнения программы, который приведёт к её аварийному завершению.

[29] https://github.com

[30] http://cran.r-project.org

Проект	Строк кода	Ошибки	Ложные срабатывания
httr	4375	0	1
readr	1440	0	0
reconstructr	615	0	0
rprime	1137	3	0
selfie	673	0	0
signals	5487	2	1

Таблица 1: Результаты тестирования

Все выбранные для тестирования проекты широко используются. Проекты, взятые с CRAN, были проверены описанным в разделе 1 статическим анализатором *codetools*. Поэтому предполагалось, что в них не содержится ошибок. Тем не менее, инспекция обнаружила несколько настоящих ошибок в этих проектах. Количество ложных срабатываний у реализованной инспекции невелико.

Заключение

В рамках данной работы достигнуты следующие результаты:

- предложена система типов для языка R;
- реализована подсистема, которая в плагине для интегрированной среды разработки IntelliJ IDEA производит анализ исходного кода на языке R без его выполнения с целью получения и дальнейшего использования информации о типах;
- реализован встроенный в среду разработки IntelliJ IDEA статический анализатор, выявляющий несоответствие реальных типов ожидаемым;
- инспекция протестирована на реальных проектах на языке R.

Список литературы

[1] Agesen Ole. The Cartesian Product Algorithm - Simple and Precise Type Inference of Parametric Polymorphism. — 1995.

[2] Cribari-Neto Francisco, Zarkos Spyros G. R: Yet another econometric programming environment // Journal of Applied Econometrics. — 1999. — Vol. 14. — P. 319–329. — URL: http://www.interscience.wiley.com/jpages/0883-7252/.

[3] Dynamic Inference of Static Types for Ruby / David An, Avik Chaudhuri, Jeffrey Foster, Michael Hicks // Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11). — ACM, 2011. — P. 459–472.

[4] Evaluating the Design of the R language : Rep. / Purdue University ; Executor: Leo Osvald Jan Vitek Floreal Morandat, Brandon Hill.

[5] Haupt, Perscheid, Hirschfeld. Type Harvesting: A Practical Approach to Obtaining Typing Information in Dynamic Programming Languages // Proceedings of the 2011 ACM Symposium on Applied Computing. — New York, NY, USA : ACM, 2011. — P. 1282–1289. — URL: http://doi.acm.org/10.1145/1982185.1982464.

[6] Hindley R. The Principal Type-Scheme of an Object in Combinatory Logic // Trans. Amer. Math. Soc. — 1969. — December. — Vol. 146. — P. 29–60.

[7] Igarashi Atsushi, Nagira Hideshi. Union Types for Object-oriented Programming // Proceedings of the 2006 ACM Symposium on Applied Computing. — SAC '06. — New York, NY, USA : ACM, 2006. — P. 1435–1441. — URL: http://doi.acm.org/10.1145/1141277.1141610.

[8] Maia Eva, Moreira Nelma, Reis Rogério. A Static Type Inference for Python.

[9] Martin-Löf Per, Sambin Giovanni. Intuitionistic type theory. Studies in proof theory. — Napoli : Bibliopolis, 1984. — ISBN: 88-7088-105-9. — URL: http://opac.inria.fr/record=b1093069.

[10] Milner Robin. A theory of type polymorphism in programming // Journal of Computer and System Sciences. — 1978. — Vol. 17. — P. 348–375.

[11] R Development Core Team. The R language definition // R Foundation for Statistical Computing. — http://cran.r-project.org/doc/manuals/R-lang.html.

[12] Rigo Armin, Pedroni Samuele. PyPy's Approach to Virtual Machine Construction // Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. — OOPSLA '06. — New York, NY, USA : ACM, 2006. — P. 944–953. — URL: http://doi.acm.org/10.1145/1176617.1176753.

[13] Ripley Brian D. The R Project in Statistical Computing // MSOR Connections. The newsletter of the LTSN Maths, Stats & OR Network. — 2001. — February. — Vol. 1, no. 1. — P. 23–25. — URL: http://ltsn.mathstore.ac.uk/newsletter/feb2001/pdf/rproject.pdf.

[14] Salib Michael. Faster than C: Static type inference with Starkiller // in PyCon Proceedings, Washington DC. — SpringerVerlag, 2004. — P. 2–26.

[15] Static Type Inference for Ruby / Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, Michael Hicks // Proceedings of the 2009 ACM Symposium on Applied Computing. — SAC '09. — New York, NY, USA : ACM, 2009. — P. 1859–1866. — URL: http://doi.acm.org/10.1145/1529282.1529700.

[16] Tierney Luke. Code analysis and parallelizing vector operations in

R // Computational Statistics. — 2009. — Vol. 24, no. 2. — P. 217–223. — URL: http://EconPapers.repec.org/RePEc:spr:compst:v:24:y:2009:i:2:p:217-223.

[17] Tu Stephen. MINO: Data-driven approximate type inference for Python. — MIT CSAIL. URL: http://people.csail.mit.edu/stephentu/papers/mino.pdf.