Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Санкт-Петербургский национальный исследовательский
Академический университет Российской академии наук»
Центр высшего образования

Кафедра математических и информационных технологий

Прошев Семен Александрович

Интерактивная отладка программ на языке R в интегрированной среде разработки IntelliJ IDEA

Магистерская диссертация

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Омельченко А. В.

Научный руководитель:
Тузова Е. А.

Рецензент:
Власовских А. С.

Санкт-Петербург
2016

Оглавление

Список терминов и сокращений

API набор готовых классов, функций и структур данных, предоставляемый приложением для использования во внешнем программном обеспечении. 6, 10, 25

IDE интегрированная среда разработки. Набор программных средств, используемый программистами для разработки программного обеспечения. Обычно включает в себя текстовый редактор, средства автоматизации сборки и отладчик. 2, 4, 5, 7, 17, 18, 21, 23, 25, 26

Плагин независимо компилируемый программный модуль, динамически подключаемый к основной программе и предназначенный для расширения и/или использования её возможностей. 5, 7, 13

Стек вызовов структура данных, хранящая информацию о вызванных подпрограммах текущей программы. Обычно под подпрограммами подразумеваются функции. Стек состоит из элементов, называемых фреймами. Каждый фрейм хранит данные о конкретной подпрограмме. 14, 15, 17, 19

Введение

В настоящее время в среде компьютерных технологий широкую популярность обрели статистика и машинное обучение – дисциплины, сопряженные с большим количеством данных и требующие соответствующих возможностей от вычислительных систем. Поддержка обработки данных большого объема существует как на уровне устройств, так и на уровне языков программирования.

Примером такого языка является R [15] – мультипарадигменный интерпретируемый язык программирования. Язык R поддерживает широкий спектр статистических и численных методов и обладает хорошей расширяемостью с помощью пакетов. В базовую поставку включен основной набор пакетов, а всего их доступно около 8000 [7].

Кроме того, R позволяет создавать программы с использованием других языков [11] и имеет встроенную поддержку визуализации результатов работы в виде графиков, позволяющую оценить и сохранить итоги вычислений. Графики создаются на основе пользовательских данных и могут быть как отображены на экране, так и сохранены на диск в одном из доступных форматов: PNG, SVG, TIFF, BMP и др.

Удобство пользования языком определяется не только его возможностями, но также сообществом и инструментами, осуществляющими его поддержку. Зачастую в наукоемких задачах и разработке программного обеспечения используются разные языки, и в таком случае весьма удобно вести разработку в одном контексте, некоторой экосистеме. Таким образом, оказывается востребованным инструмент, помогающий разрабатывать программы на R в контексте знакомых и частоиспользуемых сред разработки (IDE). Такая поддержка позволит помочь программисту сконцентрироваться на самой программе нежели на ее запуске и низкоуровневых командах по ее отладке.

Отладка программы – один из этапов разработки программного обеспечения, в ходе которого ошибки обнаруживают, локализуют и устраняют. В процессе поиска ошибки разработчики обычно анализируют значения переменных и путь, по которому выполнялась программа.

Существуют два способа отладки программного обеспечения. Прежде всего, отладка может вестить с помощью журналирования, т.е. вывода текущего состояния программы в файл или на экран. Второй способ – это использование отладчика. Отладчики позволяют исполнять программы пошагово и просматривать содержимое переменных на каждом шаге. Основным термином отладки является точка останова – место в программном коде, в котором разработчик хочет приостановить работу приложения. Это позволяет не исполнять пошагово программу целиком, а только останавливаться в интересующих местах. В работе пойдет речь именно об отладчике для программ на языке R.

Продукт с открытым исходным кодом IntelliJ [4] [3] является платформой для интегрированных сред разработки. Возможности платформы, разработанной компанией

JetBrains [5], могут быть расширены за счет установки плагинов [6]. Разработчикам плагинов платформа предоставляет базовые абстракции и интерфейсы, которые способствуют как упрощению разработки, так и гарантии ожидаемого для пользователя поведения.

Целью данной работы является разработка визуального отладчика на базе платформы IntelliJ, а также интеграция графического вывода в IDE.

Для достижения вышеуказанной цели были поставлены следующие задачи:

- анализ возможностей языка по отладке;
- разработка способа взаимодействия с интерпретатором;
- реализация отладчика;
- разработка способа поддержки графиков в IDE;
- интеграция выбранного решения в IDE.

В главе 1 рассматриваются способы реализации отладчиков, особенности языка R, касающиеся отладки и написания программ на нескольких языках. Также упоминаются текущие недостатки сред разработки RStudio [8] и StatET [13], объясняются преимущества платформы IntelliJ. В главе 2 указываются возможности языка R по отладке, происходит объяснение выбранного способа взаимодействия с интерпретатором, а также упоминаются требования, предъявляемые платформой к отладчикам программ. Кроме того, в этой главе приведен разработанный процесс отладки программ, а также дополнительные возможности, реализованные в отладчике. В главе 3 описывается способ взаимодействия программ на языке R и C/C++, объясняются текущие возможности R по реализации собственного устройства графического вывода, а также приводится разработанное и реализованное решение по интеграции графиков в IDE.

1. Обзор предметной области

1.1. Способы реализации отладчиков

Все языки программирования тем или иным образом предоставляют базовые возможности по отладке программ. Чаще всего эти возможности выражены либо в низкоуровневых командах, которые нужно вводить в консоли, либо в виде некоторого API на этом же языке. Данные способы не очень удобны для программиста, поскольку они отвлекают от написания основной программы, но с помощью вышеуказанных инструментов реализуются более интерактивные визуальные отладчики.

Обычно для подобной отладки реализуется посредник между низкоуровневым отладчиком и его визуальным аналогом. Посредник умеет преобразовывать команды пользователя в низкоуровневые составлящие, а также конвертировать данные, поступающие из консоли или API, в данные, готовые к отображению. Дополнительно он может предоставлять более широкие возможности, основываясь на функциональности имеющегося отладчика. Например, это могут быть условные точки останова и исполнение дополнительных выражений во время отладки. В случае консольного отладчика, посредник реализует запуск процесса с отладчиком, взаимодействие с ним посредством потоков ввода-вывода и его завершение. Если же отладка осуществляется с помощью API, то посредник встраивает свой код в запускаемую программу.

Дополнительно стоит отметить, что иногда между посредником и визуальным отладчиком открывается сетевое соединение. Делается это в случае, когда отладчик реализован на языке, отличном от языка программы. Данное решение получается наиболее универсальным в контексте передачи данных от одного компонента другому и обратно.

1.2. Язык R

В языке R отладка реализована с помощью команд интерпретатору [1], API не предусмотрен. Эти команды позволяют переключаться в режим отладки с последующим пошаговым исполнением программы, дают возможность узнать текущее положение в коде, а также список доступных переменных в окружении, их тип и содержимое.

Как отмечалось ранее, язык R позволяет создавать программы с использованием других языков программирования. На такие программы накладываются дополнительные ограничения в связи с тем, что части, написанные на разных языках, могут разделять между собой данные. Например, поскольку в R реализовано автоматическое управление памятью [12], то часть программы, реализованная не на R, должна самостоятельно уменьшать или увеличивать счетчик ссылок на объекты.

1.3. Обзор сред разработки на языке R

Наиболее распространенной средой разработки на языке R является RStudio. IDE разрабатывается с 2009 года, является кросс-платформенной, ее исходный код открыт [9].

RStudio поддерживает базовые потребности по работе с исходным кодом: поиск, переходы вперед и назад, а также переходы к функции или файлу. Кроме того, среда может выступать в качестве менеджера пакетов, делая возможным их установку и удаление.

RStudio позволяет отлаживать программы, а также визуализировать их графический вывод, но отладку трудно назвать интерактивной. Точки останова могут быть установлены только перед запуском программы, и их нельзя убрать или деактивировать во время ее исполнения. Этого можно было избежать с помощью условных точек останова, но в данной IDE такой функциональности нет.

Существует плагин для платформы Eclipse [2] под названием StatET, позволяющий работать с программами на языке R на базе этой платформы. StatET разрабатывается с 2005 года, его исходный код открыт [14].

StatET имеет широкие возможности по редактированию программ на языке R, по работе с интерпретатором, отображению данных, а также по работе с пакетами.

StatET позволяет производить отладку программ, но для этого необходимо самостоятельно запускать интерпретатор. Причем полностью программу отладить невозможно, можно лишь запускать отдельные функции или инструкции. Кроме того, StatET требует устанавливать дополнительное программное обеспечение для корректной работы.

1.4. Платформа IntelliJ

Платформа IntelliJ лежит в основе многих популярных сред разработки, а именно IntelliJ IDEA, PyCharm, WebStorm, PhpStorm и других. Базовые возможности платформы по работе с исходным кодом довольно широки. Кроме стандартных возможностей по созданию, удалению и редактированию файлов, платформа позволяет выполнять поиск и замену с использованием регулярных выражений, делать вертикальное выделение, а также работать с различными системами контроля версий. Функциональность платформы может быть расширена с помощью плагинов, устанавливаемых из общедоступного репозитория либо из сторонних источников.

Грамотное использование архитектуры платформы и ее абстракций позволяет упростить разработку поддержки того или иного языка. Такой подход подталкивает к более активному переиспользованию кодовой базы и поддержанию поведения, характерного для сред разработки на этой платформе.

Подобного рода общий фундамент дает возможность реализовать экосистему раз-

работки на разных языках, тем самым сильнее погрузив в свои возможности конкретного пользователя и улучшив его производительность.

2. Отладка программ на языке R

2.1. Возможности языка R по отладке

Отладка программ на языке R ведется в режиме, называемом *Browser* [10]. Этот режим переключает интерпретатор в пошаговое исполнение. Во время такого исполнения запуск текущей инструкции и переход к следующей осуществляется командой *n*. После каждого шага интерпретатор сообщает результат текущей инструкции и местоположение следующей. Список переменных в текущем окружении может быть просмотрен с помощью вызова функции *ls*, тип каждой из них можно узнать с помощью функции *typeof*. Для того, чтобы узнать значение, достаточно ввести имя переменной.

Обычно этот режим запускается при вызове функции, помеченной как отлаживаемая. Пользователи редко запускают *Browser* вручную. Пометить функцию как отлаживаемую можно с помощью команды *debug*. Данная команда сообщает интерпретатору, что пользователь хочет видеть ее пошаговое исполнение тоже.

Типовая отладка функции приведена в листинге 1.

Листинг 1: Отладка функции

```
1  > foo <- function(x) { # ввод функции
2  + x + 1
3  + }
4  > debug(foo) # пометка функции, как отлаживаемой
5  > foo(c(1:5)) # запуск функции
6  debugging in: foo(c(1:5)) # интерпретатор сообщает ее имя и включает режим Browser
7  debug at #1: {
8      x + 1
9  }
10 Browse[2]> n # пошаговое исполнение
11 debug at #2: x + 1 # интерпретатор указывает номер строки внутри функции и
       выражение, которое соответствует этой строке
12 Browse[2]> ls() # просмотр текущего окружения
13 [1] "x"
14 Browse[2]> typeof(x) # просмотр типа переменной x
15 [1] "integer"
16 Browse[2]> x # просмотр значения переменной x
17 [1] 1 2 3 4 5
18 Browse[2]> n
19 exiting from: foo(c(1:5))
20 [1] 2 3 4 5 6 # результат функции
```

2.2. Взаимодействие с интерпретатором

Поскольку в R отсутствует API для отладки программ, отладчик самостоятельно запускает и настраивает интерпретатор, анализирует его ответы. Для взаимодействия с интерпретатором был выбран стандартный способ общения с процессами, поддерживаемый всеми операционными системами, – потоки ввода-вывода. Отладчик читает сообщения из потока вывода и записывает команды в поток ввода.

В таком способе взаимодействия есть свои сложности: необходимо понимать, закончился ли ответ интерпретатора, а также определять тип произошедшего события.

Решению первой проблемы способствует режим *Browser*, который характеризуется более явным суффиксом ответа, его формат – *Browse[число]* (см. листинг 1). Данный режим включается сразу после запуска интерпретатора, чтобы на протяжении всего исполнения окончания ответов были однородны.

Типы возможных сообщений не описаны в документации языка, поэтому в ходе работы были исследованы и учтены все сообщения, встречающиеся при отладке большинства программ. На основе данного анализа была введена собственная классификация типов сообщений и способ их распознавания. Ниже приведена упомянутая классификация, а примеры сообщений представлены в приложении А.

- *PLUS* — интерпретатор ожидает продолжения ввода, данный тип ответа может быть получен при вводе многострочной функции;
- *EMPTY* — пустой ответ, данный тип ответа может быть получен при присвоении нового значения переменной;
- *DEBUGGING_ IN* — начало отладки функции;
- *DEBUG_ AT* — информация о следующей инструкции;
- *START_ TRACE_ BRACE*, *START_ TRACE_ UNBRACE* — вход в функцию, обработанную функцией *trace*;
- *CONTINUE_ TRACE* — вход в ту же функцию, из которой только что вышли;
- *EXITING_ FROM* — окончание отладки функции;
- *RECURSIVE_ EXITING_ FROM* — окончание отладок нескольких функций, данный тип ответа может быть получен, если на протяжении нескольких функций, каждая из них в последней инструкции вызывала следующую;
- *RESPONSE* — непустой ответ, данный тип ответа может быть получен в случае запроса текущего окружения, типа или значения переменной.

2.3. Особенности отладки программ на языке R

Как интерпретатор, так и сама отладка имеют некоторые особенности. В интерпретаторе, запущенном из консоли, включен интерактивный режим, в то время как при запуске его в качестве отдельного процесса этот режим отключен. Интерактивный режим включает сохранение отладочной информации для указания позиций в функциях, а также перенаправляет графический вывод на экран. При отключении интерактивного режима отладочная информация теряется, а графический вывод копится в памяти и в конце работы сохраняется в формате PDF. Во избежание потери отладочной информации в настройку интерпретатора входит вызов команды *options(keep.source=TRUE)*, который изменяет значение флага *keep.source* на *TRUE*.

По особому отлаживаются однострочные функции и циклы, а также функции, переданные в качестве аргумента функциям высшего порядка.

Листинг 2: Отладка однострочной функции

```
1 > foo <- function(x) x + 1
2 > debug(foo)
3 > foo(c(1:5))
4 debugging in: foo(c(1:5))
5 debug: x + 1 # местоположение внутрии функции не указано
6 Browse[2]> n
7 exiting from: foo(c(1:5))
8 [1] 2 3 4 5 6 # цикл отладки сокращен
```

При отладке однострочной функции интерпретатор не сообщает местоположение внутри нее, а также сокращает цикл ее отладки (см. листинг 2). В типовом случае, чтобы отладить функцию с одним выражением, необходимо выполнить три команды (вызов функции, вход внутрь нее и непосредственно исполнение самого выражения), в случае однострочной функции – две (запуск функции, исполнение выражения).

В случае отладки функции, переданной в качестве аргумента функции высшего-порядка, интерпретатор не сообщает ее имя (см. листинг 3). Данная проблема была преодолена за счет использования функции *trace*. Эта функция позволяет устанавливать обработчики на вход и выход из функции. Поскольку в R нет перегрузки функций по количеству аргументов, для обнаружения вызванной функции достаточно установить обработчик, выводящий в поток ее имя на входе (см. листинг 4). Для упрощения взаимодействия с интерпретатором, такой обработчик устанавливался на все, без исключения, функции. Также стоит отметить, что функция *trace* меняет тело своего аргумента, что требует дополнительной обработки при отображении переменных.

Листинг 3: Отладка функции-аргумента функции высшего порядка

```
> foo <- function(x) {
+ x + 1
+ }
> debug(foo)
> sapply(c(1:5), foo)
debugging in: FUN(X[[i]], ...)  # имя функции не указано
debug at #1: {
    x + 1
}
Browse[2]> n
debug at #2: x + 1
Browse[2]> n
exiting from: FUN(X[[i]], ...)
debugging in: FUN(X[[i]], ...)
debug at #1: {
    x + 1
}
 ...
Browse[2]> n
debug at #2: x + 1
Browse[2]> n
exiting from: FUN(X[[i]], ...)
[1] 2 3 4 5 6
```

В листинге 4 сперва определяется сама функция, а затем функция, которая будет вызываться при входе в исходную. Через вызов *trace* происходит ее установка. Далее демонстрируется, что тело функции действительно меняется. Функция помечается как отладочная, начинается ее отладка, во время которой вызывается обработчик, печатающий ее имя.

Листинг 4: Отладка и трассировка функции-аргумента функции высшего порядка

```
> foo <− function(x) {
+ x + 1
+ }
> foo_enter <− function() { print("foo") } # обработчик входа в функцию
> trace(foo, foo_enter)
[1] "foo"
> foo # тело функции изменено
Object with tracing code, class "functionWithTrace"
Original definition :
function(x) {
x + 1
}
> debug(foo)
> sapply(c(1:5), foo)
debugging in: FUN(X[[i]], ...)
debug: {
    .doTrace(foo_enter(), "on entry")
    {
        x + 1
    }
}
Browse[2]> n
debug: .doTrace(foo_enter(), "on entry")
Browse[2]> n
Tracing FUN(X[[i]], ...) on entry
[1] "foo" # обработчик вывел имя функции
debug: {
    x + 1
}
Browse[2]> n
debug at #2: x + 1
 ...
```

2.4. Требования к отладчику со стороны платформы

Платформа предъявляет определенные требования к плагинам, в частности к тому, как должен быть устроен отладчик программ.

Обработку пользовательских действий платформа берет на себя, при этом она требует, чтобы был реализован наследник класса *XDebugProcess*, которому будут пе-

реданы пользовательские команды.

К пользовательским командам относятся следующие действия:

- *registerBreakpoint*, *unregisterBreakpoint* – установка, отключение или удаление точки останова;
- *stepOver* – исполнение текущего выражения и переход к следующему в текущем окружении. Если после запуска данной команды произошел вызов какой-либо функции, отладчик в ней не останавливается. Исключением является наличие точки останова внутри этой функции;
- *stepInto* – вход внутрь функции и начало ее пошагового исполнения. Отладчик останавливается на первой инструкции данной функции;
- *stepOut* – окончание отладки функции. Функция исполняется до конца и отладчик останавливается на инструкции, которая идет после вызова данной функции;
- *resume* – продолжение исполнения до следующей точки останова;
- *runToPosition* – продолжение исполнения до указанного места в программе либо до точки останова, в зависимости от того, что наступило раньше;
- *stop* – остановка процесса отладки и завершение исполнения программы.

В свою очередь, платформа ожидает от наследника класса *XDebugProcess* текущий путь выполнения программы. Обычно этот путь называют стеком вызовов или же просто стеком. Каждый элемент стека указывает на место в программном коде, верхний элемент стека может сообщить о значениях переменных. Иногда остальные элементы тоже могут сообщить информацию об окружении в соответствующих им местах.

2.5. Процесс отладки

В текущем разделе приводится описание процесса отладки, разработанного на основе вышеуказанных инструментов.

В начале запускается и настраивается процесс с интерпретатором языка R. В настройку входит включение режима *Browse*, а также изменение значения флага *keep.source*.

Затем происходит анализ запускаемой программы. В ходе этого анализа строится древовидная структура, описывающая положение функций в программе, а также отношения внешняя-внутренняя функция в случае вложенных функций (см. листинги 5, 6). Данная структура используется при конвертации данных, поступающих от

интерпретатора, в данные для платформы. Как упоминалось ранее, интерпретатор сообщает только имя и номер строки внутри функции, но не сообщает, в каком месте программы была объявлена эта функция. Тем не менее, необходимо отображать пользователю текущее положение в коде, причем с учетом вложенных и переопределенных функций. На основе стека программы и структуры, вычисленной на данном этапе, можно однозначно установить местоположение.

Листинг 5: Пример программы с вложенной и переопределенной функциями

```
foo <- function(x) {
    bar <- function(y, z) { # вложенная функция
        y + z
    }

    x + bar(x, x)
}

print(foo(c(1:5)))

foo <- function(x) { # переопределение функции foo
    x + 1
}

print(foo(c(1:5)))
```

Листинг 6: Дерево функций

```
# start - строка начала определения функции, end - конца
# отсчет начинается с 0

"foo"
    |- start: 0, end: 6
        |- "bar"
            |- start: 1, end: 3
    |- start: 10, end: 12
```

Следующим шагом идет отправка программы интерпретатору. В ходе работы было установлено, что если посылать программу так, как если бы ее в консоли вводил пользователь, то была бы невозможна отладка циклов. В таком случае, после ввода тела цикла, интерпретатор сразу его выполняет, не останавливаясь на каждой итерации (см. листинг 7). Поэтому код программы отправляется как отдельная функция, после чего запускается ее отладка (см. листинг 8).

Листинг 7: Запуск цикла на верхнем уровне

```
Browse[1]> for (i in 1:5) {
+ print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Листинг 8: Запуск цикла внутри функции

```
> foo <- function() { # определение функции с циклом
+ for (i in 1:5) {
+ print(i)
+ }
+ }
> debug(foo)
> foo()
debugging in: foo()
debug at #1: {
    for (i in 1:5) {
        print(i)
    }
}
Browse[3]> n
debug at #2: for (i in 1:5) { # интерпретатор остановился перед циклом
    print(i)
}
Browse[3]> n
debug at #3: print(i) # интерпретатор остановился на первой итерации цикла
Browse[3]> n
[1] 1
debug at #3: print(i) # интерпретатор перешел ко второй итерации цикла
 ...
Browse[3]> n
[1] 5
exiting from: foo()
```

Отладчик начинает посылать команды интерпретатору, чтобы тот шаг за шагом

исполнял программу. На каждом шаге отладчик вычисляет текущее положение в программе, и если оно соответствует некоторой точке останова, исполнение приостанавливается. В этот момент в среде разработки подсвечивается местоположение, а также отображается стек программы и переменные в текущем окружении. В случае условных точек останова, сперва проверяется истинность введенного условия. По построению, такие точки останова получаются динамическими – нет сложностей с их добавлением, отключением или удалением.

На описанном подходе основана команда *resume*, т.к. подход полностью соответствует ее поведению. Несмотря на то, что поведение других команд отличается, подход может быть к ним адаптирован. В случае команд *stepOver* и *stepOut* дополнительно требуется отслеживать размер стека. Если пользователь исполняет команду *stepOver*, останавливаться нужно не только в случае точки останова, но и в случае если размер стека вернулся на первоначальную величину или же стал меньше. Для команды *stepOut* нужно останавливаться в случае, если размер стека сократился. Команда *runToPosition* добавляет временную точку останова, которая удаляется по достижению, процесс исполнения такой же. Команда *stepInto* не проверяет текущее местоположение на совпадение с точкой останова, ей достаточно сделать единственный шаг и остановиться после этого.

2.6. Архитектура отладчика

Технически отладчик состоит из двух частей (рис. 1). Первая часть, наследник класса *XDebugProcess*, обрабатывает пользовательские действия, связанные с установкой и отключением точек останова. Кроме того, эта же компонента занимается обработкой команд, непосредственно касающихся процесса отладки:

- stepOver;
- stepInto;
- stepOut;
- resume;
- runToPosition;
- stop.

Вторая часть представляет из себя более низкоуровневый отладчик. Эта компонента умеет передавать интерпретатору команду на исполнение следующей инструкции, обрабатывать его сообщения и обновлять отладочную информацию. Она же предоставляет доступ к стеку вызовов. Каждый элемент стека хранит соответствующее

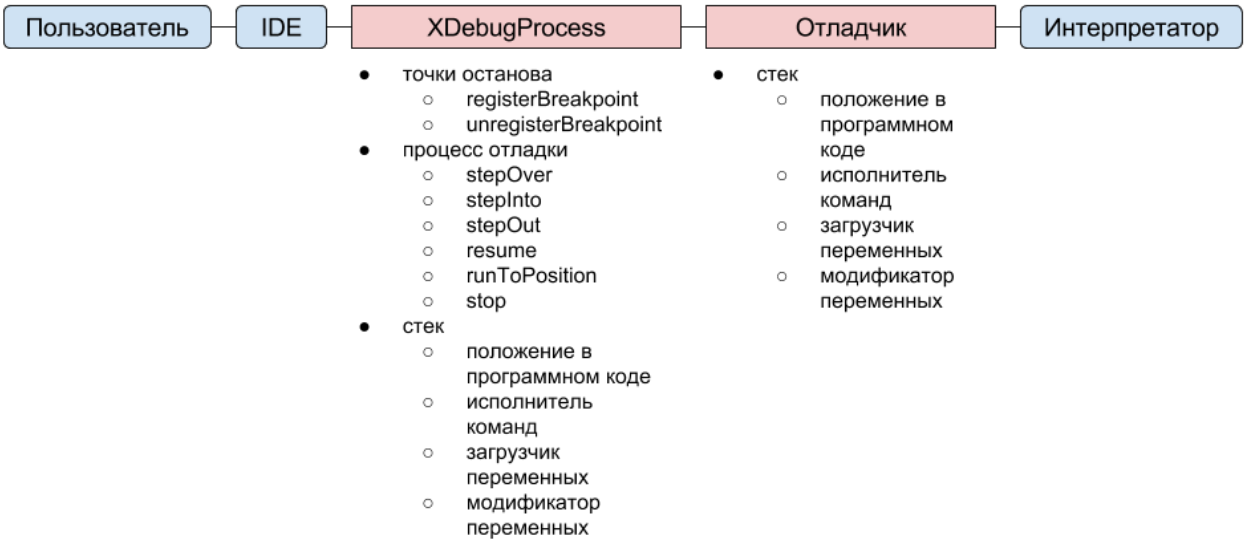


Рис. 1: Архитектура отладчика

положение в программном коде (имя функции и номер строки в ней), а также позволяет загрузить список переменных и их значения. Верхнему элементу стека доступно исполнение какой-либо команды, в том числе обновление значения указанной переменной.

Наследник класса *XDebugProcess* адаптирует данные возможности низкоуровнего отладчика для использования их в IDE. Так, например, он преобразует имя функции и номер строки в строку файла с исходным кодом. Для этого он использует стек вызовов и структуру программы, вычисленную на этапе запуска. Там, где не требуется преобразование структур данных, эта компонента встраивает переданные объекты в IDE, оборачивая их в экземпляры требуемых классов.

2.7. Дополнительные возможности

Кроме необходимых, в отладчике были реализованы дополнительные возможности, позволяющие сделать процесс более удобным. В частности, поскольку язык R является интепретируемым, а не компилируемым, упрощается реализация исполнения сторонних выражений, которые не являются частью программы. Это позволяет поддержать условные точки останова, упомянутые ранее. Также благодаря указанному устройству языка, была реализована возможность изменять значение переменных во время исполнения, данная функциональность доступна только для текущих переменных.

В контексте отладки на базе платформы IntelliJ присутствуют такие пользовательские действия, как *add watch* и *evaluate expression*. *Add watch* – это команда, позволяющая указать выражение, результат которого пользователь хочет видеть при каждой приостановке исполнения. *Evaluate expression* – это команда, позволяющая выполнить

введеное выражение в текущем месте исполняемой программы. Обе возможности поддерживаются в отладчике, причем, как в случае *add watch*, так и в случае *evaluate expression* стоит учитывать, что выражение может быть некорректным для места, в котором оно было вызвано.

Для сокращения объема данных, передаваемых между отладчиком и интерпретатором, была реализована ленивая загрузка окружения, а именно списка переменных, их типов и значений. Эта информация загружается только по требованию пользователя, причем она доступна не только для последнего элемента стека, но и нижеследующих (см. листинг 9).

Листинг 9: Загрузка окружения

```
> x <− c(1:3) # определение переменной на верхнем уровне
> foo <− function(x) { # определение функции на верхнем уровне
+ x + 1
+ }
> debug(foo)
> sys.nframe() # команда выводит номер текущего фрейма в стеке
[1] 0
> foo(c(1:5))
debugging in: foo(c(1:5))
debug at #1: {
    x + 1
}
Browse[2]> n
debug at #2: x + 1
Browse[2]> sys.nframe()
[1] 1
Browse[2]> ls(sys.frame(0)) # список переменных на верхнем уровне
[1] "foo" "x"
Browse[2]> ls(sys.frame(1)) # список текущих переменных
[1] "x"
Browse[2]> sys.frame(0)$x # значение переменной на верхнем уровне
[1] 1 2 3
Browse[2]> sys.frame(1)$x # значение текущей переменной
[1] 1 2 3 4 5
Browse[2]> typeof(sys.frame(0)$x) # тип переменной на верхнем уровне
[1] "integer"
Browse[2]> typeof(sys.frame(1)$x) # тип текущей переменной
[1] "integer"
Browse[2]> typeof(sys.frame(0)$foo) # тип переменной на верхнем уровне
[1] "closure"
Browse[2]> typeof(sys.frame(1)$foo) # тип текущей переменной
[1] "NULL" # поскольку foo нет в текущем фрейме, sys.frame(1)$foo вернет NULL
```

3. Интеграция графического вывода в IDE

3.1. Взаимодействие R и C/C++

Язык R позволяет загружать динамические библиотеки на C/C++ во время исполнения программы. Возможности загруженной библиотеки становятся доступны сразу, она лишь должна удовлетворять двум требованиям. Во-первых, интерфейс библиотеки должен быть реализован на языке C. Во-вторых, поскольку исполнение происходит в контексте интерпретатора, библиотека должна быть написана с учетом этого контекста.

Основным термином взаимодействия библиотеки и интерпретатора является тип *SEXP*. Все объекты в R, передаваемые в библиотеку, являются представителями только этого типа, поэтому функции из интерфейса могут принимать и возвращать только *SEXP*-объекты. Технически, *SEXP* это указатель, аналог *void** в C. В целом, базис взаимодействия указан в заголовочных файлах *R.h*, *Rinternals.h* и *Rdefines.h*, которые поставляются вместе с дистрибутивом языка.

Библиотека загружается с помощью команды *dyn.load*, функция из нее может быть вызвана с помощью встроенных функций *.Call* или *.External*. В первом случае каждый передаваемый объект сопоставляется с каждым параметром вызываемой функции. В случае *.External* все аргументы объединяются в один набор и единым целым передаются в библиотеку (см. листинги 10, 11).

Листинг 10: Пример кода на C

```
1  SEXP convolve_c(SEXP a, SEXP b)
2  # определение функции, которая будет вызвана с помощью .Call
3  # каждый переданный объект сопоставится с отдельным параметром
4  {
5   ...
6  }
7  SEXP convolve_e(SEXP args)
8  # определение функции, которая будет вызвана с помощью .External
9  # объекты будут переданы в виде одного набора
10 # функция должна будет извлечь их самостоятельно
11 {
12  ...
13 }
```

Листинг 11: Вызов кода на C из R

```
> dyn.load("libmylib.so") # загрузка библиотеки, в качестве аргумента передается путь к
    ней
> .Call("convolve_c", a, b) # вызов функции convolve_c с аргументами a и b
...
> .External("convolve_e", a, b)
...
```

Необходимо также учитывать, что в R реализовано автоматическое управление памятью: освобождать память не нужно, интерпретатор сделает это самостоятельно. При создании R-объекта на стороне библиотеки надо сообщить интерпретатору, что этот объект удалять не следует. Делается это с помощью макроса *PROTECT*. В противовес ему существует макрос *UNPROTECT*, который принимает в качестве параметра целое число (см. листинг 12). *PROTECT* и *UNPROTECT* работают по принципу стека: *PROTECT* кладет на верх стека поступивший объект, тогда как *UNPROTECT* снимает со стека количество объектов, равных переданному числу.

Листинг 12: Пример работы с R-объектами на C

```
SEXP ab = PROTECT(allocVector(REALSXP, 2));
# создание и защита от удаления вектора длиной 2 для вещественных чисел
REAL(ab)[0] = 123.45;
REAL(ab)[1] = 67.89;
UNPROTECT(1);
# исключение самого последнего объекта из списка защиты
# в данном случае выделенного ранее вектора
```

3.2. Возможности языка R по реализации устройств графического вывода

R имеет встроенные способы генерации графиков на основе пользовательских данных, причем эти графики могут быть отображены как на экране, так и сохранены в виде одного из доступных форматов: PDF, SVG, PNG, JPEG, TIFF и др.

Ко всему прочему, R позволяет реализовывать и использовать собственные устройства графического вывода. Базис графического вывода описан в заголовочных файлах *GraphicsEngine.h* и *GraphicsDevice.h*, которые тоже поставляются вместе с дистрибутивом языка. В них содержатся структуры данных и функции, позволяющие описывать и регистрировать новое устройство, переключаться между запущенными экземплярами или же останавливать их работу.

Основным термином графического вывода является структура данных *DevDesc*.

DevDesc хранит в себе указатели на функции, обрабатывающие события графического вывода. Например, это могут быть функции, рисующие линию, ломаную, текст, круг или же прямоугольник. Кроме того, эти функции могут вызываться, когда устройство активировано, деактивировано или закрыто. Под активацией подразумевается, что весь последующий графический вывод будет направлен на это устройство, под закрытием – деактивация устройства и его удаление из списка доступных.

На основе экземпляра структуры *DevDesc* создается объект *GEDevDesc*, которым оперирует непосредственно сам интерпретатор. В *GEDevDesc* может храниться история графических событий, отправленных на устройство. Имея экземпляр структуры *GEDevDesc*, библиотека может зарегистрировать данное устройство, как готовое к выводу, а также активировать его. Пример создания, регистрации и активации собственного устройства графического вывода приведен в листинге 13.

Листинг 13: Создание, регистрация и активация устройства графического вывода

```
1  pDevDesc desc = new DevDesc; # создание дескриптора устройства
2
3  # ниже дескриптор устройства инициализируется
4
5   ...
6
7  desc->deviceSpecific = NULL;
8  desc->displayListOn = TRUE;
9
10  ...
11
12 pGEDevDesc device = GEcreateDevDesc(desc);
13 # на основе дескриптора создается устройство, которым оперирует интерпретатор
14 GEaddDevice2(device, "MY_DEVICE");
15 # устройство добавляется в список устройств
16 # оно будет фигурировать под именем MY_DEVICE
17
18 Rf_selectDevice(Rf_ndevNumber(device->dev));
19 # устройство становится активным, активация происходит по номеру
```

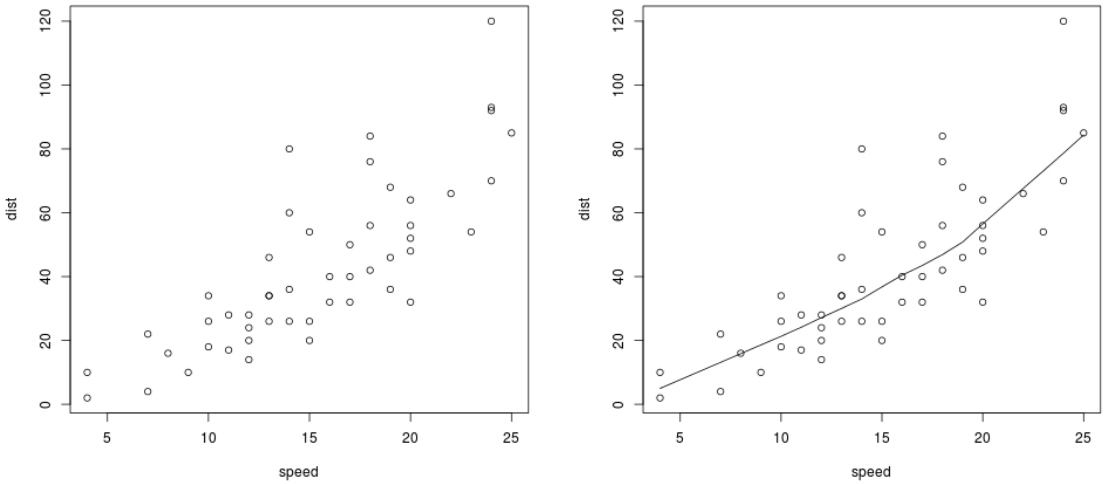
3.3. Интеграция графического вывода в IDE

Одной из задач интерактивной отладки программ на языке R была интеграция графиков непосредственно в среду разработки. Необходимо было разработать и реализовать способ сохранения графического вывода и его отображения в среде. Причем требовалось учесть, что иногда графики могут обновляться во время работы про-

граммы. Этот процесс представлен листингом 14: при вызове первой команды будет отображен график 2a, последующая команда обновит график, и он станет выглядеть, как график 2b.

Листинг 14: Отрисовка графика в два этапа

```
1 > plot(cars)
2 > lines(lowess(cars))
```



(a) Первоначальный график (b) График после обновления

Непосредственное использование встроенных устройств в таком случае не помогает: они не сохраняют график до тех пор, пока не началась отрисовка нового графика, поэтому потребовалась разработка дополнительного устройства.

Разработанное в ходе работы устройство берет за основу встроенный графический вывод, сохраняющий данные в формате PNG. При инициализации дескриптора устройства создается фоновое PNG-устройство, с которого берутся настройки, связанные с выводом изображения, например, гамма-коррекция. В качестве функций, которые требуют реализации, подставляются функции, вызывающие таковые у фонового устройства. Соответственно, все вызовы, поступающие главному устройству, переадресовываются фоновому. Тем не менее, это не позволяет устранить недостаток встроенных в R устройств.

Чтобы поддержать сохранение графика и всех его обновлений, необходимо получать события начала и конца отрисовки элементов графика. К счастью, R предоставляет такую возможность. В структуре данных *DevDesc* может быть установлен обработчик под названием *mode*. Функции *mode* в качестве аргумента R передает 0 или 1. В случае начала процесса рисования – 1, в случае окончания – 0. Таким образом, необходимо при вызове этой функции с аргументом 0 делать сохранение теку-

щего графика, что может быть осуществлено только с помощью закрытия фонового устройства. В начале отрисовки следующих элементов графика, фоновое устройство будет создано заново, на него будет скопирована история графических событий с главного устройства.

Итого был реализован следующий процесс получения и обработки графического вывода:

1. создание фонового устройства с копированием истории графический событий;
2. перенаправление всех графических событий фоновому устройству;
3. закрытие фонового устройства после отрисовки элементов графика;
4. переход к первому пункту при следующей отрисовке.

После того, как устройство скомпилировано в динамическую библиотеку и загружено, необходимо сообщить интерпретатору, что данное устройство будет использоваться по умолчанию. Подобного рода конфигурация может быть выполнена с помощью вызова команды *options(device=my_ constructor)*, где *my_ constructor* – функция, создающая и активирующая устройство. Загрузка и установка собственного устройства в качестве устройства по умолчанию приведены в листинге 15.

Листинг 15: Загрузка собственого устройства в качестве устройства по умолчанию

```
> dyn.load("libdevice.so")

> device_init_r <- function() { .Call("device_init_c", "snapshots") }
# интерфейсом библиотеки является функция device_init_c,
# создающая, регистрирующая и активирующая устройство.
# device_init_c в качестве аргумента принимает путь,
# по которому будут сохраняться графики

> options(device=device_init_r)
# установка функции, вызываемой интерпретатором для создания устройства
```

После того, как графический вывод был перехвачен и сохранен на диск, необходимо на стороне IDE узнать о новом графике, доступном для отображения.

В платформе реализована виртуальная файловая система, упрощающая работу с различными файловыми системами и предоставляющая дополнительные возможности. Посредством API виртуальной файловой системы можно установить обработчик событий внутри какой-либо директории. Платформа будет сообщать обработчику, когда файл создан, модифицирован или удален. Таким образом, достаточно следить за событиями в директории с графиками и своевременно отображать их.

Заключение

В рамках данной работы достигнуты следующие результаты:

- реализован отладчик программ на языке R, предоставляющий следующие возможности:
  - динамические точки останова;
  - отображение типов и значений переменных;
  - отладка функций (в т.ч. вложенных и переопределенных);
  - интерпретация выражений во время отладки (*add watch*, *evaluate expression*, условные точки останова);
  - отладка циклов;
- произведена интеграция графического вывода программ в IDE.

Список литературы

[1] Debugging tools in R. — URL: http://www.noamross.net/blog/2013/4/18/r-debug-tools.html.

[2] Eclipse. — URL: https://eclipse.org/.

[3] IntelliJ GitHub. — URL: https://github.com/JetBrains/intellij-community.

[4] IntelliJ IDEA. — URL: https://www.jetbrains.com/idea/.

[5] JetBrains. — URL: https://www.jetbrains.com/.

[6] JetBrains plugin repository. — URL: https://plugins.jetbrains.com/.

[7] R packages. — URL: https://cran.r-project.org/web/packages/.

[8] RStudio. — URL: https://www.rstudio.com/.

[9] RStudio GitHub. — URL: https://github.com/rstudio/rstudio.

[10] R's environment browser. — URL: https://stat.ethz.ch/R-manual/R-devel/library/base/html/browser.html.

[11] R's system and foreign language interfaces. — URL: https://cran.r-project.org/doc/manuals/R-exts.html#System-and-foreign-language-interfaces.

[12] R's memory management. — URL: http://adv-r.had.co.nz/memory.html.

[13] StatET. — URL: http://www.walware.de/goto/statet.

[14] StatET GitHub. — URL: https://github.com/walware/statet.

[15] Язык R. — URL: https://www.r-project.org/.

A. Примеры типов сообщений интерпретатора

```
> foo <- function() {
# тип: PLUS
+ sapply(c(1:2), bar)
# тип: PLUS
+ }
> bar <- function(x) {
# тип: PLUS
+ x + 1
# тип: PLUS
+ }
> foo_enter <- function() print("foo")
# тип: EMPTY
> bar_enter <- function() print("bar")
# тип: EMPTY
> trace(foo, foo_enter)
# тип: RESPONSE
[1] "foo"
> trace(bar, bar_enter)
# тип: RESPONSE
[1] "bar"
> debug(foo)
# тип: EMPTY
> debug(bar)
# тип: EMPTY
> foo()
# тип: DEBUGGING_IN
debugging in: foo()
debug: {
    .doTrace(foo_enter(), "on entry")
    {
        sapply(c(1:2), bar)
    }
}
Browse[2]> n
# тип: DEBUG_AT
debug: .doTrace(foo_enter(), "on entry")
Browse[2]> n
# тип: START_TRACE_BRACE
Tracing foo() on entry
```

```
[1] "foo"
debug: {
    sapply(c(1:2), bar)
}
Browse[2]> n
# тип: DEBUG_AT
debug at #2: sapply(c(1:2), bar)
Browse[2]> n
# тип: DEBUGGING_IN
debugging in: FUN(X[[i]], ...)
debug: {
    .doTrace(bar_enter(), "on entry")
    {
        x + 1
    }
}
Browse[3]> n
# тип: DEBUG_AT
debug: .doTrace(bar_enter(), "on entry")
Browse[3]> n
# тип: START_TRACE_BRACE
Tracing FUN(X[[i]], ...) on entry
[1] "bar"
debug: {
    x + 1
}
Browse[3]> n
# тип: DEBUG_AT
debug at #2: x + 1
Browse[3]> n
# тип: CONTINUE_TRACE
exiting from: FUN(X[[i]], ...)
debugging in: FUN(X[[i]], ...)
debug: {
    .doTrace(bar_enter(), "on entry")
    {
        x + 1
    }
}
Browse[3]> n
# тип: DEBUG_AT
```

```
debug: .doTrace(bar_enter(), "on entry")
Browse[3]> n
# тип: START_TRACE_BRACE
Tracing FUN(X[[i]], ...) on entry
[1] "bar"
debug: {
    x + 1
}
Browse[3]> n
# тип: DEBUG_AT
debug at #2: x + 1
Browse[3]> n
# тип: RECURSIVE_EXITING_FROM
exiting from: FUN(X[[i]], ...)
exiting from: foo()
[1] 2 3
```

В. Исходный код

Исходный код отладчика и интеграции графического вывода в IDE доступен по ссылке https://github.com/ktisha/TheRPlugin.

Исходный код устройства графического вывода расположен по адресу https://github.com/sproshev/TheRPlugin_Device.