Методы отладки функциональных программ

В. Н. Касьянов, email: kvn@iis.nsk.su
Е. В. Касьянова, email: kev@iis.nsk.su
К. А. Кламбоцкий, email: konstantin.klambotsky@gmail.com

Институт систем информатики им. А.П. Ершова СО РАН

Аннотация. *В докладе рассматриваются методы отладки функциональных программ и опыт их использования для отладки Cloud Sisal программ в рамках создаваемой в ИСИ СО РАН системы CPPS поддержки облачного параллельного программирования.*

Ключевые слова: *отладка программ, система CPPS, функциональное программирование, язык Cloud Sisal*.

Введение

Современные подходы к разработке параллельных программ в основном являются архитектурно-ориентированными, когда для достижения эффективной работы создаваемые программы тесно связаны с архитектурами параллельных вычислительных систем, на которых они выполняются и, как правило, разрабатываются. Поэтому требования к квалификации разработчиков параллельных программ весьма высоки, тем более, что протестировать и отладить параллельную программу намного сложнее, чем последовательную, а проблема верификации параллельных программ весьма далека от решения не только практически, но и теоретически.

Более того, в современной вычислительной технике идет постоянная смена архитектурных парадигм, что, в свою очередь, ведет к проблеме переносимости уже разработанных параллельных программ. Приходится постоянно адаптировать уже созданный продукт под изменившиеся аппаратные средства. Это обуславливается тем, что различные параллельные вычислительные системы имеют свойственные только им ресурсные ограничения, которые необходимо учитывать во время разработки программы. Проведение таких адаптаций является весьма интеллектуальной задачей, требующей существенного переписывания параллельных программ и выполнения практически заново их верификации и отладки. В результате адаптированные параллельные программы зачастую содержат новые ошибки и не достигают желаемой и возможной эффективности.

© Касьянов В. Н., Касьянова Е. В., Кламбоцкий К. А., 2022

Поэтому создание методов и технологий архитектурно-независимого параллельного программирования является весьма актуальным. Есть насущная потребность в языковых и инструментальных средствах, которые обеспечат создание и отладку архитектурно-независимых параллельных программ независимо от используемого вычислителя и их корректную адаптацию к разным параллельным вычислительным системам для их эффективного выполнения. Одним из перспективных путей решения указанной проблемы является разработка декларативных средств описания и реализации параллельных вычислений.

Облачная система параллельного программирования CPPS [1, 2], разрабатываемая в ИСИ СО РАН, использует функциональный язык Cloud Sisal для разработки, отладки, верификации и исполнения параллельных программ. В рамках создаваемой системы CPPS прикладной программист будет иметь возможность через браузер создавать, отлаживать и верифицировать Cloud Sisal программу в визуальном стиле и без учета целевого вычислителя, а затем с помощью оптимизирующего кросс-компилятора производить настройку отлаженной программы на тот или другой супервычислитель, доступный ему по сети, с целью достижения высокой эффективности исполнения получаемой параллельной программы, а также передавать построенную программу супервычислителю на счет и получать результаты.

Целью работы, представленной в данном докладе, является исследование существующих методов отладки функциональных программ на языках Haskell, F# и Common Lisp, разработка на их основе методов отладки Cloud Sisal программ и их экспериментальная реализация для представительного подмножества языка Cloud Sisal в виде отдельного веб-модуля на языке JavaScript, легко встраиваемого в систему CPPS и позволяющего пользователю удаленно отлаживать Cloud Sisal программу.

1. Язык Cloud Sisal и система CPPS

Язык Cloud Sisal [3] продолжает традицию предыдущих версий языка SISAL, оставаясь функциональным потоковым языком, ориентированным на написание больших научных программ, и расширяет их возможности средствами поддержки облачных вычислений. Функциональная семантика языка Cloud Sisal гарантирует детерминированные результаты для параллельной и последовательной реализации – то, что невозможно гарантировать для традиционных императивных языков, подобных языкам Фортран или Си. Более того, неявный параллелизм языка снимает необходимость переписывания

исходного кода при переносе его с одного вычислителя на другой. Cloud-Sisal-программа, правильно исполняющаяся на персональном компьютере, будет гарантированно правильно исполняться на любом высокоскоростном параллельном или распределенном вычислителе.

При этом методы аннотированного программирования и конкретизирующих преобразований, поддерживаемые системой CPPS, позволяют в рамках декларативного стиля программирования настраивать процессы адаптации переносимых параллельных программ на классы задач и архитектуру вычислителя при сохранении их корректности, а также получать более эффективный параллельный код за счет использования при адаптации знаний пользователя о задаче, программе и вычислителе, выраженные в аннотациях.

Система CPPS [1, 2] разрабатывается как интегрированная облачная среда программирования на языке Cloud Sisal, которая содержит как интерпретатор, поддерживающий диалоговое взаимодействие с пользователем при построении и отладке программы, так и оптимизирующий кросс-компилятор, осуществляющий построение параллельной программы по её функциональной спецификации.

Интерпретатор и оптимизирующий компилятор системы CPPS используют единое внутреннее теоретико-графовое представление IR функциональных Cloud-Sisal-программ, которое ориентировано на их визуальную обработку и основано на атрибутированных иерархических графах.

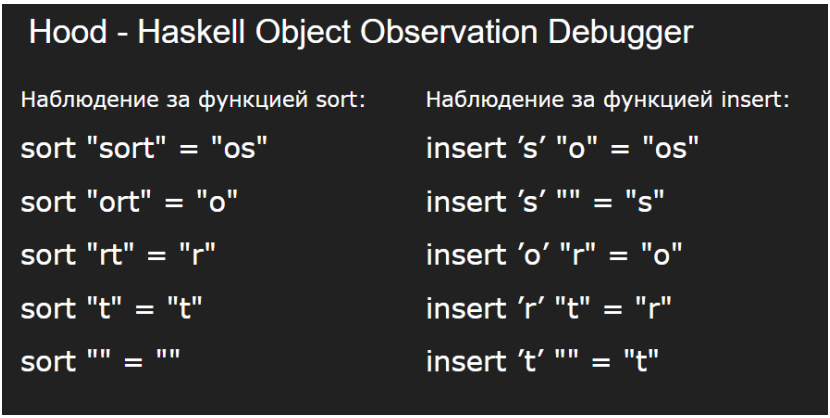

Рис. 1. Пример использования библиотеки Hood

Система CPPS поддерживает построение наглядных изображений графовых внутренних представлений Cloud-Sisal-программ и их использование при конструировании корректных функциональных программ, в том числе для визуальной отладки Cloud-Sisal-программ [1, 4]. Она также поддерживает построение визуальных представлений внутренних структур данных, возникающих в компиляторе при построении параллельных программ, и динамических процессов, возникающих при исполнении построенных параллельных программ, и их использование для управляемой оптимизации с целью повышения рабочих характеристик параллельных программ, получаемых с помощью компилятора по их функциональным спецификациям.

2. Методы отладки для языка Haskell

Простым методом отладки программ на языке Haskell является Debug.Trace.

Debug.Trace.trace: trace :: String -> a -> a

При вызове trace выводит строку в своем первом аргументе, прежде чем вернуть второй аргумент в качестве результата. Преимущество состоит в том, что для отключения и включения трассировки требуется только одна строка комментария. Но стоит заметить, что вследствие работы ленивых вычислений трассировка будет выводиться, только если значение было посчитано в программе.

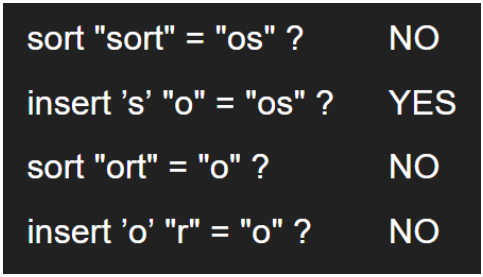

Рис. 2. Пример использования библиотеки Freja

Hood — популярная библиотека для отладки Haskell программ, использует модифицированный метод отладки методом вывода, предлагая пользователю поставить на переменную, выражение или функцию модификатор-наблюдатель (observe), который в течение работы программы будет уведомлять в консольное окно или в файл об этапах вычисления, выводя промежуточный результат. Даже если он не

обновлялся какое-то время, Hood отлично работает с текущей версией языка. Исходный код:

```
import Hugs.Observe
f'  = observe "Informative name for f" f
f x = if odd x then x*2 else 0
```

Результат наблюдения выводит отчет обо всех вызовах f и результатах:

```
Main> map f' [1..5]
[2,0,6,0,10]
```

Freja — компилятор для подмножества Haskell 98. Сессия отладки состоит из последовательности вопросов, на которые отвечает пользователь. Каждый вопрос относится к правильности раскрытия редексов. Пользователь должен ответить «да», если раскрытие является правильным, и «нет» иначе (см. Рис.2 и Рис.3).

В конце отладчик утверждает, раскрытие какого редекса является причиной наблюдаемой ошибки, то есть определение какой функции является неправильным. Первый вопрос всегда начинается с функции main. Если на вопрос о раскрытие редекса функции получен ответ «нет», то следующий вопрос применяется для оценки правой части определения этой функции.

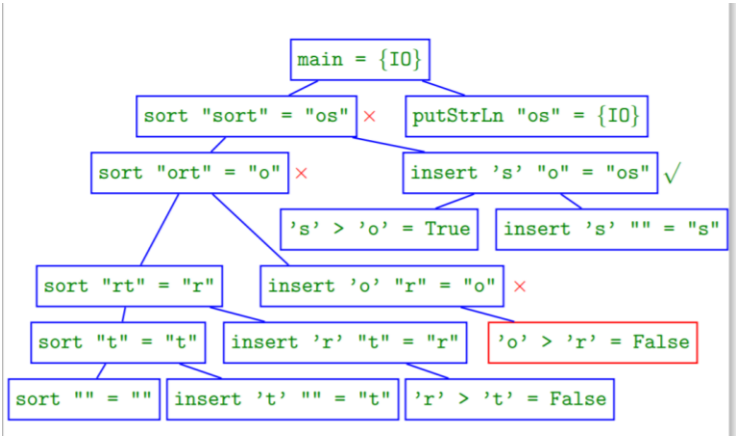

Рис. 3. Визуализация пользовательской сессии библиотеки Freja

Freja может использоваться как обыкновенный отладчик. Ответ «нет» означает «войти внутрь функции», а ответ «да» означает «перейти к следующей функции». Если раскрытие редекса функции является

неправильным, а все сокращения в правой части функции являются правильными, то определение этой функции должно быть неправильным для заданных аргументов.

Система Redex Trail System состоит из модифицированной версии компилятора Haskell и отдельной программы браузера. Программа, скомпилированная для трассировки, выполняется как обычно, за исключением того, что вместо завершения в конце она открывает браузер Redex Trail и показывает вывод программы. Пользователь выбирает выражение и спрашивает браузер о его родительском редексе.

Родительский редекс выражения — это редекс, в результате раскрытия которого получается это выражение. У каждой части редекса снова есть родительский редекс, который браузер показывает по требованию. Все заканчивается на функции main, у которой нет родителя. Отладка с помощью системы Redex Trail работает путем движения от ошибочного вывода или сообщения об ошибке назад вверх к родителям, до обнаружения ошибки.

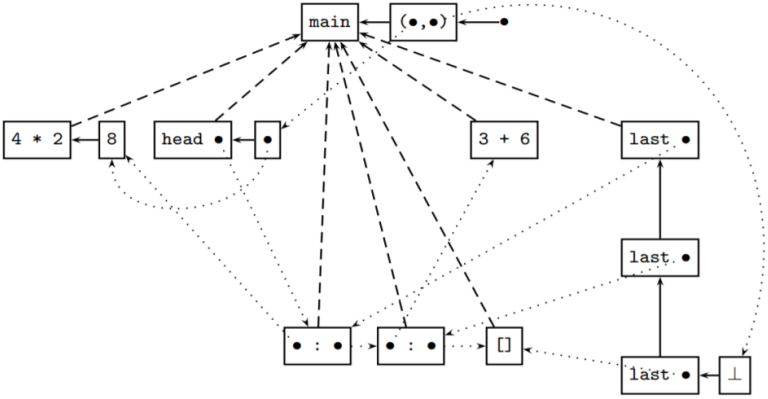

Рис. 4. Redex Trail graph

В основном, система используется следующим образом для обнаружение ошибки в программе

main = let xs = [4*2, 3+6] in (head xs, last xs)

head (x:xs)=x

last (x:xs)= last xs

last [x]=x

Программа прерывается с сообщением об ошибке, и браузер напрямую отображает родительский redex: last[]. Пользователь

обнаруживает, что последняя функция вызывается с пустым списком в качестве аргумента, и запрашивает у браузера родительский редекс last[ ]. Ответ last[3 + 6] проясняет, что определение last указано неверно для списка из одного элемента. Браузер (Рис. 4) также может показать, где в тексте программы последний раз вызывается с помощью пустого списка в уравнении для последнего (x: xs).

Еще один метод, упоминаемый в ряде статей по отладке программ на языке Haskell, — это так называемые trusted функции. Данный метод направлен на сокращение времени отладки программы и заключается в том, что “проверенные функции” отмечаются программистом как trusted. Считается, что они работают правильно и не нуждаются в отладке.

Реализация выглядит следующим образом: используется модификатор “trusted”, который исключает данную функцию из отладки другими методами. Результат функции будет считаться верным, пошаговая отладка не зайдет внутрь данной функции, в логах не будет промежуточного результата, если пользователь поставит другие модификаторы, например, observe на родительский редекс.

3. Методы отладки для языка F#

F# использует отладку методом пошагового выполнения программы и использование точек останова. Во время работы отладчика будут открыты такие окна, как «Модули» (Modules) и «Наблюдения» (Watch). Переменные в окне «Локальные» (Locals) или выражения в окне «Наблюдения» (Watch) будут доступны только тогда, когда отладчик приостановлен в точке останова. Это также известно как «режим прерывания», при котором операции приостанавливаются, и все элементы программы могут быть проверены на правильность.

Как только программа останавливается на какой-то строчке можно нажать F11, чтобы перейти на следующую строку с остановкой на каждой строке. Важно отметить, что условие рассматривается как один шаг, а результат - как отдельный шаг. Если функция является вложенным вызовом, отладка перейдет к самой глубокой вложенной функции в этом вызове. F10 позволяет пройти по функциям избегая входа внутрь каждой функции. что ускоряет отладку таким методом.

Другой распространенный аспект отладки — запуск до установленной точки останова. Эту точку останова можно установить вручную или определить с помощью условия или функции, а также запустить ее в положении курсора или щелкнуть мышью.

Чтобы установить точку останова, нажмите F9 или переключите точку останова в соответствующем раскрывающемся меню в Debug. Также можно нажать слева от номера строки и на поле слева от кода

появится красная точка, указывающая на строку, к которой перейдет отладчик.

Кроме того, отладчик может работать с указанной функцией. Это можно сделать, указав функцию по имени или выбрав ее из стека вызовов.

В режиме отладки можно поставить автоматическую остановку на срабатывание различных исключений (Exceptions). При этом можно указать тип исключений либо останавливаться на любых исключениях.

Когда в отладчике срабатывает пауза, желтая стрелка указывает на следующую строку, которую нужно выполнить. Однако это можно изменить вручную, если требуется пропуск. Пока отладчик находится в режиме прерывания, можно переместить стрелку на нужную строку. Затем отладчик немедленно пропустит все прерывистые строки. Следует отметить один важный аспект: переход назад не приведет к отмене уже выполненных инструкций.

Дополнительные команды отладчика, такие как «Add Watch» и «Go to Disassembly», работают должным образом. Однако, отладчик не распознает выражения F#, выражения должны быть переведены в синтаксис C# во время отладки.

4. Методы отладки для языка Common Lisp

Рассмотрим методы отладки языка Common Lisp. Конечно, в этом языке тоже есть метод отладки выводом (print debugging) Команда print работает, она печатает читаемое (readable) представление своего аргумента. Команда princ фокусируется на эстетике вывода. С помощью нее можно изменять формат вывода для более удобного чтения.

Пример: (with-output-to-string (princ "The buffer is ") (princ (buffer-name)))

Выводит: "The buffer is foo".

Также формат вывода можно менять с помощью функции format, похожую на printf в языке Си, она выводит в консоль текст, при этом специальные выражения заменяются на аргументы после текста.

(defvar a 4)

(format T "2 + 2 = ~d" a)

Выводит: "2 + 2 = 4".

Логирование — следующий этап эволюции после отладки путем вывода. В языке для этого есть различные инструменты. Но давайте рассмотрим библиотеку log4cl. Для начала модификатор log к выражению отправляет в консоль значение выражения:

(defvar *foo* '(:a :b :c))(log:info *foo*)

Выводит: “<INFO> [13:36:49] cl-user () - *FOO*: (:A :B :C)”

Таким образом, можно выводить текст и выражения, при этом использовать специально форматирование

(log:info "foo is " *foo*)

Выводит: ";; <INFO> [13:37:22] cl-user () - foo is *FOO*: (:A :B :C)".

(log:info "foo is ~{~a~}" *foo*)

Выводит: ";; <INFO> [13:39:05] cl-user () - foo is ABC".

В языке Common Lisp также есть функция inspect вывода сложной структуры данных. Например, если есть коллекция данных, то при обычном выводе все запишется в одну строку, без указания индексов. В случае inspect произойдет построчный вывод всех элементов коллекции с указанием индексов.

(inspect *foo*)

Выводит:

The object is a proper list of length 3.

0. 0: :A
1. 1: :B
2. 2: :C

```
(trace factorial)

(factorial 2)
  0: (FACTORIAL 3)
    1: (FACTORIAL 2)
      2: (FACTORIAL 1)
        3: (FACTORIAL 0)
        3: FACTORIAL returned 1
      2: FACTORIAL returned 1
    1: FACTORIAL returned 2
  0: FACTORIAL returned 6
6

(untrace factorial)
```

Рис. 5. След функции (factorial 2)

Функция trace позволяет отслеживать вызовы определенной функции. Например, это будет эффективно для рекурсивных функций (см. Рис. 5).

При этом можно остановить слежение за функцией — (untrace factorial).

Заключение

В процессе отладки программисты пользуются различными методами. Базовыми методами отладки считаются: метод вывода промежуточного результата (print debug) и пошаговое исполнение (breakpoint debug). В связи с недетерминированностью работы функциональных Cloud Sisal программ базовые методы могут оказаться не столь эффективными как для императивных программ.

Поэтому в ходе выполнения данной работы были изучены специальные методы отладки функциональных программ на языках Haskell, F# и Common Lisp и из них были выбраны для отладки Cloud Sisal программ продвинутые методы отладки языка Haskell: Freja — интерактивное раскрытие редексов, Hood — слежение за изменением данных, и Redex Trail System — визуализация дерева раскрытия редексов. На их основе были разработаны методы отладки Cloud Sisal программ и выполнена их экспериментальная реализация для представительного подмножества языка Cloud Sisal.

Предложенные в работе методы представляют интерес для применения в системе CPPS, поскольку они дополняют используемые в системе методы визуальной отладки Cloud Sisal программ на основе их графового представления [2, 4] и опираются на успешную практику применения подобных методов для других функциональных языков.

Список литературы

1. Касьянов, В. Н. Система облачного параллельного программирования CPPS: визуализация и верификация Cloud Sisal программ / В. Н. Касьянов, Д. С. Гордеев, Т. А. Золотухин [и др.]; под ред. В. Н. Касьянова. — Новосибирск: ИПЦ НГУ, 2020. — 256 с.

2. Касьянов, В. Н. Программные средства поддержки дистанционного обучения функциональному программированию / В. Н. Касьянов, А. А. Малышев // Информатика: проблемы, методы, технологии: Материалы XXI Международной научно-методической конференции / под ред. А. А. Зацаринного, Д. Н. Борисова. — Воронеж: ООО «Вэлборн», 2021. — С.1834–1842.

3. Касьянов, В. Н. Язык программирования Cloud Sisal / В. Н. Касьянов, Е. В. Касьянова. — Новосибирск, 2018. — 45 с. — (Препринт/ РАН, Сиб. отд-ние, ИСИ; N181).

4. Касьянов, В. Н. Методы и алгоритмы визуализации графовых представлений функциональных программ / В. Н. Касьянов, Т. А. Золотухин, Д. С. Гордеев // Программирование. — 2019. — № 4. — С.19–27.