Ю. Хан

ОБЗОР СРЕДСТВ ОТЛАДКИ ПРОГРАММ НА ФУНКЦИОНАЛЬНЫХ ЯЗЫКАХ[1]

ВВЕДЕНИЕ

В процессе создания программного обеспечения немаловажную роль играет отладка. По оценкам экспертов, она занимает от половины до двух третей всего времени разработки [1, 2]. Следовательно, необходимы эффективные отладочные средства, приспособленные к используемой парадигме.

Функциональная парадигма вносит свои характерные особенности в процесс отладки. Так, например, порядок выполнения действий в программе может существенно отличаться от того порядка, который можно предположить, читая её исходный текст, особенно в случае «ленивых» вычислений. Некоторые проблемы, часто возникающие в императивных программах, не свойственны функциональным программам (например, использование неинициализированной переменной или утечка памяти).

Таким образом, разные методики отладки имеют различную применимость в зависимости от парадигмы. В данной работе делается попытка проанализировать существующую ситуацию в отношении функционального программирования.

1. СУЩЕСТВУЮЩИЕ СИСТЕМЫ ПРОГРАММИРОВАНИЯ НА ФУНКЦИОНАЛЬНЫХ ЯЗЫКАХ

1.1. Caml и Objective Caml

Caml и его объектно-ориентированное расширение — это диалекты языка ML, разработанные в Национальном исследовательском институте информатики и автоматики (INRIA) (Франция). Поддерживаются платформы

[1] iis@centaur.mailshell.com

Работа выполнена при финансовой поддержке Российского фонда фундаментальных исследований (грант № 04-01-715) и Министерства образования РФ (научная программа «Университеты России», грант № УР.04.01.027)

UNIX, Windows и Macintosh. Существуют две реализации — Caml Lite и Objective Caml.

Язык обладает, помимо чисто функциональных свойств, также некоторыми чертами императивных языков. В частности, в нём есть многократно присваиваемые массивы, циклы и исключительные ситуации.

В Caml Lite основным средством отладки является трассировка. В режиме трассировки при входе в отлаживаемую функцию выводятся её аргументы, а при выходе — результат. Для более детального исследования работы программы авторы рекомендуют использовать отладочный вывод. Также предоставляется пошаговый отладчик, который, однако, реализован только под UNIX [3].

Система Objective Caml включает в себя отладчик, работающий под UNIX и Windows (только в версии под Cygwin). Отладчик предоставляет пользователю традиционные возможности:

- пошаговое исполнение;
- точки останова;
- стек вызовов;
- просмотр (но не изменение) значений переменных.

Также поддерживается удалённая отладка, когда программа и отладчик работают на разных машинах.

Отладчик не имеет собственного интерфейса пользователя. Управление производится через командную строку. Кроме того, пользователь может подключить отладчик к редактору Emacs, что позволит соотносить текущую позицию в программе с её исходным текстом [4].

1.2. Mercury

Язык Mercury, разработанный в Мельбурнском университете, сочетает в себе черты логических и функциональных языков. В нём отсутствуют побочные эффекты, используется строгая типизация. Язык поддерживает модульное программирование. Компилятор работает под UNIX-подобными системами [5].

Система включает в себя отладчик, работающий автономно (с использованием интерфейса командной строки) или в интеграции с редактором Emacs. Поддерживаются следующие возможности:

- пошаговое исполнение;
- просмотр значений переменных;
- стек вызовов;
- точки останова (с возможностью временного отключения);

- декларативная отладка.

В режиме декларативной отладки система задаёт пользователю вопросы о правильности результатов функций, начиная с самой внешней. Если функция даёт (согласно пользователю) неверный ответ, а все вложенные вызовы правильные, отладчик заключает, что ошибка находится в этой функции.

1.3. Scheme

Scheme — упрощённый диалект языка LISP. Были рассмотрены две реализации: EdScheme версии 5.0 (Schemers Inc.) и DrScheme версии 208 (группы PLT). Обе работают под управлением Windows и Macintosh; DrScheme также поддерживает Linux, FreeBSD и Solaris.

В EdScheme средства отладки ограничиваются трассировкой. Пользователь определяет подмножество функций, которые будут трассироваться, и запускает вычисление выражения. После окончания вычислений система отображает дерево вызовов отмеченных функций. В каждом узле отображаются аргументы функции; результат по умолчанию скрыт, но может быть показан по команде пользователя. Также есть режим, когда вычисление приостанавливается при достижении трассируемой функции [6].

DrScheme включает средство Stepper, позволяющее шаг за шагом проследить редукции выражения. На каждом шаге вызов функции разворачивается в её результат либо определение. Цель Stepper'а — скорее обучающая, чем отладочная. Также DrScheme поддерживает трассировку, которая ограничивается выводом всего дерева вызовов пользовательских функций с подставленными значениями аргументов [7].

Следует упомянуть встроенный механизм модульного тестирования. Вместе с определением функции программист может задать выражения и их ожидаемые результаты. При перекомпиляции модуля тесты выполняются, и рядом с ними отображается символ успешного или неуспешного выполнения, что позволяет программисту сразу определить момент, когда изменение программы приводит к нарушению её работы. Методика модульного тестирования подробно описана в книге [8].

Кроме того, система следит за тем, чтобы все пути управления были протестированы; фрагменты программы, которые не покрываются тестами, выделяются цветом.

1.4. Erlang

Язык Erlang разработан в фирме Ericsson для создания надёжных распределённых систем реального времени. Он поддерживает явные конструкции параллелизма, интерфейс к другим языкам программирования, сборку мусора в реальном времени. Компилятор работает под управлением Windows и UNIX-подобными системами.

Среда предоставляет традиционные средства пошаговой отладки — точки останова, пошаговое выполнение, просмотр и изменение значений переменных. Отладчик реализует собственный графический интерфейс пользователя [9].

1.5. Oz

По заявлениям разработчиков, язык Oz объединяет в себе возможности функционального, объектно-ориентированного и логического программирования и позволяет создавать параллельные многопоточные и распределённые системы. Виртуальная машина, исполняющая программы на Oz'е, работает под основными вариантами UNIX и под Windows [9].

Возможности отладчика включают:

- отображение дерева (строго говоря, леса) всех работающих в данный момент потоков программы. Для каждого потока отображается его состояние — работает, заблокирован, завершился успешно, завершился аварийно. Здесь пользователь может выбрать поток для отладки;
- стек вызовов (с значениями параметров функций) для каждого потока;
- пошаговую отладку;
- точки останова;
- просмотр значений переменных;
- удалённую отладку.

Отладчик интегрируется с редактором Emacs.

1.6. Haskell

Haskell — последовательный чисто функциональный язык с отложенными («ленивыми») вычислениями. Существует несколько реализаций, работающих на различных платформах.

Были рассмотрены системы: Buddha, Freja, HOOD и HsDebug.

Buddha [11] и Freja [12] реализуют механизм декларативной отладки. При этом используются различные способы сокращения выводимой информации:

- Freja приводит аргументы функций к наиболее упрощённому виду;
- обе системы выводят длинные структуры только частично, предоставляя пользователю возможность развернуть их при необходимости;
- обе системы не отображают подвыражения, которые в ходе работы программы не вычислялись;
- библиотечные функции считаются правильными и не участвуют в трассировке и отладке.

Поскольку полный след программы может занимать значительный объём памяти, система Freja производит частичную трассировку. При этом сохраняется только та часть следа, на которую хватает памяти, а если процесс отладки выходит за границы этого фрагмента, программа перезапускается с новыми параметрами трассировки.

Другой подход, используемый в системе HOOD (Haskell Object Observation Debugger) [12], состоит в выводе промежуточных значений выражений. Отслеживаемые выражения оборачиваются в функцию, которая эквивалентна тождественной, за исключением того, что она выводит свой аргумент в отладочный вывод.

Наконец, третий вариант — отказ от семантики отложенных вычислений. Система HsDebug заменяет «ленивые» вычисления строгими, что позволяет использовать традиционную методику отладки [13]. Однако такая замена не всегда бывает корректна. В частности, в случае отложенных вычислений программа может использовать бесконечные структуры. Попытка строго вычислить такую структуру приведёт к зацикливанию или исчерпанию свободной памяти. В HsDebug вводятся ограничения по времени для вычисления каждого выражения.

2. МЕТОДИКИ ОТЛАДКИ

Вышеописанные наблюдения позволяют выделить три основных направления.

2.1. Трассировка

При компиляции в программу добавляется код, собирающий информацию о ходе выполнения. После того как программа отработает, программист может использовать собранные данные для нахождения ошибок.

Преимущества:

- *простая реализация*.

Недостатки:

- *отсутствие наглядности*. Часто протокол состоит из серии имён функций с значениями параметров. Соотнесение их с исходным текстом программы возлагается на пользователя;
- *пассивность*. Выполнение и отладка программы разнесены во времени; во время отладки нельзя повлиять на выполнение;
- *замедление*. При больших объёмах отладочной информации её запись может занимать значительное время.

Метод непосредственно применим для строгих языков без параллелизма, в том числе не только функциональных, но и императивных. В случае параллельных программ, где потоки влияют друг на друга, может требоваться сопоставление протоколов отдельных потоков между собой.

2.2. Пошаговое выполнение

Отладка ведётся одновременно с выполнением программы. Как правило, пользователь видит текущее состояние программы и может влиять на степень подробности рассмотрения. Часть отладчиков также позволяют вмешиваться в работу программы, изменяя значения переменных.

Преимущества:

- полная информация о процессе выполнения;
- контроль количества выводимой информации;
- возможность влияния на исполнение программы.

Недостатки:

- метод фокусируется на операционной семантике программы — *из каких шагов состоит* вычисление, а не *что именно* вычисляется;
- реализация требует разработки механизма сопоставления внутреннего представления программы с исходным текстом. В случае отладки программ, скомпилированных в машинный код процессора, дополнительно требуется поддержка со стороны компилятора — сохранение соответствия между именами и адресами переменных, документированное представление данных и т. п.

Пошаговое исполнение применяется к императивным и строгим функциональным языкам.

2.3. Декларативная отладка

Метод декларативной, или алгоритмической, отладки был первоначально разработан для логических языков программирования. Позже его перенесли на функциональные языки с отложенными вычислениями.

В «ленивых» языках вычисления откладываются до того момента, когда их результат непосредственно потребуется, поэтому промежуточные данные могут представлять собой довольно громоздкие выражения, а порядок действий может сильно отличаться от интуитивного. Это считается причиной плохой применимости традиционных пошаговых методов к отладке программ с отложенными вычислениями [14].

Алгоритмическая отладка состоит из двух фаз.

- Пользовательская программа выполняется в специальном режиме трассировки. При этом собирается *след* — дерево вызовов пользовательских функций, где сохраняются аргументы и результаты.
- След обходится в глубину под контролем пользователя, который выносит суждения о правильности или ошибочности результатов применения той или иной функции.

Процесс завершается, когда найдена вершина дерева, давшая неверный результат, в то время как все её потомки вычислились корректно.

Метод имеет те же основные недостатки, что и трассировка — пассивность и замедление.

ЗАКЛЮЧЕНИЕ

Рассмотренные системы демонстрируют три основных подхода к отладке функциональных программ — трассировку, пошаговое выполнение и декларативную (алгоритмическую) отладку. При этом пошаговая отладка преобладает в системах программирования на строгих языках, в то время как для языков с отложенными вычислениями применяется алгоритмическая отладка.

СПИСОК ЛИТЕРАТУРЫ

1. **Brooks F. P. Jr.** The Mythical Man-Month: Essays on Software Engineering. — Boston: Addison–Wesley Professional, 1995. — 322 p.
2. **Spolsky J.** Painless Software Schedules. — http://www.joelonsoftware.com/articles/fog0000000245.html

3. **Weis P.** Frequently asked questions about Caml Light. — http://pauillac.inria.fr/caml/FAQ/index-eng.html
4. **Leroy X. et al.** The Objective Caml system release 3.08. Documentation and user's manual. — http://pauillac.inria.fr/caml/ocaml/htmlman/index.html
5. **Henderson F. et al.** The Mercury User's Guide. — http://www.cs.mu.oz.au/research/mercury/information/doc-release/user_guide_toc.html
6. **The EdScheme** for Windows Knowledge Base. — http://www.schemers.com/
7. **PLT** DrScheme: Programming Environment Manual. — http://download.plt-scheme.org/doc/drscheme/
8. **Beck K.** Extreme Programming Explained: Embrace Change. — Boston: Addison–Wesley Professional, 1999. — 224 p.
9. **Erlang** Debugger User's Guide. — http://www.erlang.se/doc/doc-5.4/lib/debugger-2.3/doc/html/part_frame.html
10. **Lorenz B., Kornstaedt L.** The Mozart Debugger. — http://www.mozart-oz.org/documentation/ozcar/index.html
11. **Pope B.** buddha Version 1.2 User's Guide. — http://www.cs.mu.oz.au/~bjpop/buddha/onlinedocs/UserGuide/UserGuide.html
12. **Nilsson H.** How to Look Busy while Being as Lazy as Ever: The Implementation of a Lazy Functional Debugger // J. of Functional Programming. — 2001. — N 11(6). — P. 629–671.
13. **The Haskell** Object Observation Debugger User Manual. — http://www.haskell.org/hood/documentation.htm
14. **Ennals R., Jones S. P.** HsDebug: Debugging Lazy Programs by Not Being Lazy // ACM SIGPLAN Haskell Workshop'03, August 28th, 2003, Uppsala, Sweden. —2003. — P. 84–87.
15. **Chitil O., Runciman C., Wallace M.** Freja, Hat and Hood — A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs // Proc. of 12th Internat. Workshop on Implementation of Functional Languages 2000, Aachen, Germany, Sept. 4–7, 2000. — Berlin etc.: Springer-Verlag, 2001. — P. 176–193. — (Lect. Notes Comput. Sci.; N 2011).