**Библиотека переводных публикаций
по функциональному программированию**

Мартин Эрвиг

Побег от Зурга: упражнение в логическом программировании

Martin Erwig. Escape from Zurg: An Exercise in Logic Programming. — 2004

Проект журнала «Практика функционального программирования»

Этот перевод и другие материалы проекта «Библиотечка ПФП» доступны на fprog.ru/lib.

Библиотека переводных публикаций по функциональному программированию

Перевод: Евгений Прохоров

Ревизия: 3206 (2011-01-31)

Сайт проекта: http://fprog.ru/

Побег от Зурга: упражнение в логическом программировании

Мартин Эрвиг
School of EECS, Oregon State University
(e-mail: erwig@cs.orst.edu)

2004

Аннотация

В этой статье мы покажем, как современные функциональные языки, например, Хаскель, могут эффективно использоваться для решения поисковых задач, вопреки широко распространённому мнению, что для подобных задач лучше подходит Пролог.[1]

[1]Оригинал статьи: http://web.engr.oregonstate.edu/~erwig/papers/Zurg_JFP04.pdf

1. Введение

Принято считать, что Пролог — наиболее подходящий язык для решения поисковых задач. Одна из сильных черт Пролога — его встроенный поиск с возвратами (бэктрекинг), способный значительно сократить работу в таких задачах. С другой стороны, Хаскель ([3]) предоставляет мощную систему типов, функции высшего порядка и отложенные (ленивые) вычисления. Мы хотим показать в этой статье, что эти свойства вместе позволяют выражать и решать поисковые задачи на Хаскеле так же легко, как и на Прологе (и, возможно, даже легче). Например, отложенные вычисления облегчают краткое описание пространства поиска, поскольку спецификация бесконечной структуры данных — дерева перебора — может быть записана без зацикливания, так как вычисляется только конечная её часть. Эта идея не нова, она была описана раньше Филом Вадлером ([9]). Тем не менее, переписывание кода в каждой поисковой задаче из-за малейшей мелочи — скучно, ненадёжно, и может отвлекать от самой реализуемой поисковой задачи. Концепция классов типов (далее просто классов) в Хаскеле предлагает простой способ сформулировать решение один раз и повторно использовать его в разных случаях. К тому же типы данных Хаскеля позволяют (и в некоторой степени заставляют) формулировать задачу в адекватной форме.

Альтернативный подход — встраивание Пролого-подобного языка в функциональный язык. Это было продемонстрировано для Хаскеля ([8], [1]) и Scheme ([4]). Однако, наша цель — выразить поисковые задачи функционально, без применения мультипарадигменного подхода.

Пример, который мы будем рассматривать — это домашнее задание из нашего аспирантского курса языков программирования ([2]). Эта задача включалась в практикум программирования на Прологе. Когда мы увидели, что многие студенты испытывают трудности при манипулировании структурами термов в Прологе (уже научившись пользоваться типами данных в Хаскеле) и тратят много времени на отладку, возник вопрос, насколько сложно решить эту задачу на Хаскеле. Это программистское упражнение оказалось успешным, о чём мы и сообщаем в этой статье.

В оставшейся части статьи мы опишем требования к обучению программированию поисковых задач на Хаскеле в главе 2. В главе 3 описан пример задачи. В главе 4 мы покажем решение задачи на Прологе. В главе 5 представлено решение задачи на Хаскеле. Завершают статью выводы, данные в разделе 6.

2. Обучение поисковому программированию на языке Хаскель

Чтобы изучать поисковое программирование на Хаскеле, согласно предлагаемому подходу, студенты должны хорошо понимать основные концепции функ-

ционального программирования, такие как рекурсия, списки и функции высшего порядка. Также студенты должны понимать концепции классов, типов данных и отложенных вычислений, поскольку эти концепции используются для создания модульного решения.

Во-первых, классы используются для разделения общего описания поисковых задач и частной задачи. В частности, мы используем многопараметрические классы для параметризации поисковой задачи типом состояний и типом ходов. Нет необходимости углублённо знать многопараметрические классы. Фактически, класс `SearchProblem` может служить побуждающим примером для введения мультипараметрических классов. Сам класс может быть разработан поэтапно. Вначале можно определить версию с одним параметром (переменной типа), которая параметризуется только типом состояний поиска. Затем, обнаружив, что для некоторых поисковых задач, таких как обсуждаемая здесь, состояния решений не так интересны, как предшествовавшие им ходы, можно обобщить класс до двух параметров (переменных типов).

Во-вторых, в выбранном примере для создания модели приложения используются типы данных. Типы предоставляют более высокоуровневые средства для моделирования приложения, нежели кодирование кусочков информации обычными списками и кортежами. Также инкапсуляция процедуры поиска в класс помогает полностью сфокусироваться на моделировании задачи, поскольку позволяет не возиться с поиском. Это похоже на Пролог, в котором процедура поиска встроена в язык. Тем не менее, в отличие от термов Пролога, типы данных Хаскеля дают типизированное представление, непосредственно указывая на некорректные комбинации ходов и состояний, которые при использовании нетипизированного представления могут потребовать множества усилий на отладку.

В-третьих, знание отложенных вычислений необходимо для понимания того, как в Хаскеле может быть представлено потенциально бесконечное пространство поиска. Простой поиск в ширину реализуется через создание списка состояний путём постоянного добавления к нему последующих состояний. В зависимости от желания и наличия свободного времени на работу с поисковым программированием можно изучить этот аспект глубже. Например, относительно просто обобщить класс, параметризовав создание пространства поиска поисковой стратегией. Также, поскольку поисковая задача изолирована в класс, это изменение не затрагивает модель приложения.

Реализация поисковых задач обсуждается лишь в нескольких книгах по ML и Хаскель. Например, Паульсон описывает реализацию поискового программирования на ML для доказательства теорем ([6]). Раби и Лапальм описывают реализацию алгоритмов бектрекинга на Хаскеле ([7]). Они используют явно определённый алгоритм поиска в глубину и не используют классы для отделения класса задачи от приложений. В частности, они не выделяют отдельный тип для ходов, вследствие чего их подход не подходит для рассматриваемой здесь зада-

чи. Фелляйсен и др. описывают в своей книге ([5]) похожий пример — задачу о трёх миссионерах и каннибалах, пересекающих реку. Этот пример приводится как упражнение по программированию с использованием рекурсии и накапливающих параметров (аккумуляторов), которые очень подробно обсуждаются как средство хранения контекстной информации в рекурсивных функциях.

3. Пример задачи

Задача, которую мы рассмотрим, называется «Побег от Зурга» и сформулирована так:

Базз, Вуди, Рекс и Хэмм убегают от Зурга[2]. Им осталось лишь перейти последний мост, и они будут свободны. Однако мост очень ветхий и сможет одновременно выдержать только двух из них. Также, чтобы перейти мост и не попасть в ловушки и ямы в нём, нужен фонарик. Проблема в том, что у наших четырёх друзей всего один фонарик, а заряда батареи в нём осталось лишь на 60 (шестьдесят) минут. Игрушкам требуется различное время, чтобы перейти мост в одну сторону:

Игрушка	Время
Базз	5 минут
Вуди	10 минут
Рекс	20 минут
Хэмм	25 минут

Так как одновременно на мосту могут находиться только две игрушки, они не могут перейти мост сразу все вместе. Поскольку для перехода через мост им нужен фонарик, кому-то из двоих, перешедших через мост, нужно будет вернуться к оставшимся игрушкам, чтобы отдать им фонарик.

Итак, задача такова: в каком порядке эти четыре игрушки должны пересечь мост (затратив не более 60 минут), чтобы спастись от Зурга?

Попробуйте решить задачку на своём любимом языке программирования.

4. Решение на Прологе

Написание на Прологе программы для решения загадки — это, в принципе, простая задача — по крайней мере, после того как понятен способ ее представления. Как оказалось, в этом и было основное затруднение у многих студентов.

[2] Это персонажи из мультфильма «Игрушечная история 2» (Toy Story 2).

Сложнее всего для студентов было определить подходящее представление термов для состояний поисковой задачи, в данном случае — для положений игрушек на той или другой стороне моста и положения фонарика. В частности, две основные ошибки заключались в использовании слишком сложных структур термов или предикатов, а также в использовании несовместимых термов, а в некоторых случаях даже в смешивании предикатов и термов. Некоторые программы зацикливались. Образец решения показан в следующем фрагменте кода для сравнения с решением на Хаскеле, которое будет разработано в следующей главе.

```
time(buzz,  5).
time(woody,10).
time(rex,  20).
time(hamm, 25).

toys([buzz,hamm,rex,woody]).

cost([],0) :- !.
cost([X|L],C) :-
     time(X,S),
     cost(L,D),
     C is max(S,D).

split(L,[X,Y],M) :-
     member(X,L),
     member(Y,L),
     compare(<,X,Y),
     subtract(L,[X,Y],M).

move(st(l,L1),st(r,L2),r(M),D) :-
     split(L1,M,L2),
     cost(M,D).

move(st(r,L1),st(l,L2),l(X),D) :-
     toys(T),
     subtract(T,L1,R),
     member(X,R),
     merge_set([X],L1,L2),
     time(X,D).

trans(st(r,[]),st(r,[]),[],0).
trans(S,U,L,D) :-
     move(S,T,M,X),
     trans(T,U,N,Y),
     append([M],N,L),
     D is X + Y.
```

```
cross(M,D) :-
     toys(T),
     trans(st(l,T),st(r,[]),M,D0),
     D0=<D.

solution(M) :- cross(M,60).
```

Идея программы на Прологе заключается в представлении промежуточных состояний переходов через мост фактами вида `st(P,L)`, где `L` — список игрушек, находящихся в данный момент на левой стороне моста, а `P` — признак, показывающий положение фонарика (левая или правая сторона).[3]

Предикат `move/4` генерирует ходы в своём третьем аргументе; переход направо генерируется в случае, если фонарик находится на левой стороне, и наоборот; ход также связывает старое состояние (первый аргумент) с полученным новым состоянием (второй аргумент). Последний аргумент выдаёт время, необходимое для хода.

В случае перехода направо время определяется дополнительным предикатом `cost/2`, вычисляющим максимальное время, необходимое группе игрушек. Все возможные группы игрушек, двигающихся направо, генерируются предикатом `split/3`, выдающем списки длины 2 (отсортированные для избежания избыточности, полученной представлением групп игрушек в списках).

При переходе налево есть смысл посылать назад только одну игрушку. Поэтому определение хода в этом случае использует предопределённый предикат `member/2` и вычисляет время, просто просматривая таблицу `time/2`.

Наконец, предикат `trans/4` просто генерирует все возможные переходы через мост вместе с требуемым временем, тогда как предикат `cross/2` формулирует поисковую задачу, задавая начальную и конечную конфигурации пространства поиска.

5. Решение на Хаскеле

Решение на Хаскеле мы получаем в два этапа. Во-первых, мы извлекаем общую структуру поисковой задачи и выражаем её в определении класса типов. Во-вторых, мы представляем программу решения головоломки в виде экземпляра этого класса.

Главные элементы задачи — это *состояние* (представление промежуточной стадии перехода через мост) и *ход* (представление перехода между состояниями, в данном случае — пересечения моста). Поэтому мы определяем класс

[3] Чаще всего в студенческих решениях встречался подход, представляющий две группы игрушек — по каждую сторону моста — но мы обнаружили, что, хотя эта избыточность и может помочь в обдумывании задачи, но сохранение инварианта было главным источником ошибок.

`SearchProblem` с двумя переменными-типами `s` и `m`. Затем мы рассмотрим, какие методы нужны классу `SearchProblem`.

Чтобы построить целиком пространство поиска, начинающееся с некоторого (начального) состояния `s`, нужна функция, определяющая, какие новые состояния могут быть получены из `s`. Вообще говоря, нас интересует не только конечное состояние (более того, в данном примере оно нам уже известно — а именно, все игрушки находятся на другой стороне). Вместо этого нас интересует последовательность ходов, ведущая к этому состоянию.

Поэтому, мы добавили в класс функцию, вычисляющую для состояния список допустимых ходов и новых состояний, к которым они ведут:

```
trans :: s → [(m,s)]
```

Повторно применяя функцию `trans` к листьям дерева перебора можно построить всё пространство поиска. Это пространство представляется элементами типа `Space m s` и строится функцией `space`, которая отображает состояние в список всех узлов пространства поиска. Каждый узел (то есть, состояние) складывается в пару со списком ходов, ведущих к нему.

```
type Space m s = [([m], s)]

space :: s → Space m s
```

Поскольку пространство поиска полностью описывается начальным состоянием и функцией `trans`, функция `space` является производным методом класса `SearchProblem`. Описание `space` (приведенное в листинге далее по тексту) показывает важность использования отложенных вычислений: `space` ссылается на себя (косвенно через `expand`) без условия завершения; при строгом вычислении это определение бы, в общем случае, зациклилось. В данном примере это означает, что в определении функции `trans` достаточно учесть всего два случая; дополнительное определение

```
trans (R,[]) = []
```

не нужно, хотя это условие необходимо в программе на Прологе (сравните первое предложение в предикате `trans`).

Вообще говоря, решения поисковой задачи задаются некоторым подмножеством её состояний. Проверка решения производится предикатом от состояния и порождающих его ходов:

```
isSolution :: ([m],s) → Bool
```

С помощью этого предиката может быть определён другой метод — `solutions`, который имеет тот же тип, что и `space`, и просто возвращает подмножество состояний, удовлетворяющих предикату `isSolution`. Определение класса `SearchProblem` полностью приведено в следующем листинге.

```
type Space m s = [([m],s)]

class SearchProblem s m where

    trans :: s → [(m,s)]
    isSolution :: ([m],s) → Bool
    space, solutions :: s → Space m s

    space s = step ++ expand step
      where step = [ ([m],t) | (m,t) ← trans s ]
            expand ss = [ (ms++ns,t) | (ms,s) ← ss,
                                       (ns,t) ← space s ]

    solutions = filter isSolution · space
```

Определив схему решения, в программе для решения загадки на Хаскеле остается только смоделировать задачу. Самые важные архитектурные решения — это определения типов `BridgePos` и `Move`, так как они определяют экземпляр класса `SearchProblem`. В `BridgePos` мы представляем положение фонарика конструкторами `L` или `R`, а перечисление игрушек, которые находятся на левой стороне моста — списком, точно так же, как и в реализации на Прологе.

Ход — это или переход группы из двух игрушек слева направо, или обратный переход всего одной игрушки справа налево. Оба вида ходов указаны типом `Move`, который определён через предопределённый в Хаскеле тип данных `Either`, имеющий конструкторы `Left` и `Right` для представления «несвязных сумм»[4] типов.

Полностью решение приведено в следующем листинге.

```
data Toy = Buzz | Hamm | Rex | Woody deriving (Eq,Ord,Show)
data Pos = L | R deriving (Eq,Show)
type Group = [Toy]
type BridgePos = (Pos,Group)
type Move = Either Toy Group

toys :: [Toy]
toys = [Buzz,Hamm,Rex,Woody]

time :: Toy → Int
time Buzz  = 5
time Woody = 10
time Rex   = 20
time Hamm  = 25

duration :: [Move] → Int
```

[4] *disjoint sum* — объединение двух непересекающихся множеств (*Прим. ред.*)

```
duration = sum · map (either time (maximum · map time))

backw :: Group → [(Move,BridgePos)]
backw xs = [(Left x, (L, sort (x:(toys \\ xs)))) | x ← xs]

forw :: Group → [(Move,BridgePos)]
forw xs = [(Right [x,y], (R, delete y ys)) |
              x ← xs, let ys=delete x xs, y ← ys, x < y]

instance SearchProblem BridgePos Move where

    trans (L,l) = forw l
    trans (R,l) = backw (toys \\ l)

    isSolution (ms,s) = s == (R,[]) && duration ms ≤ 60

solution :: Space Move BridgePos
solution = solutions (L,toys)
```

Помимо определения типов для представления объектов в программе, самая важная часть — это определение экземпляра класса `SearchProblem`, дающего определения функций `trans` и `isSolution`. Для этого мы определили три вспомогательные функции: `forw` и `backw` для вычисления ходов игрушек и `duration` для вычисления общего времени перехода, в данном случае, для последовательности ходов. Напомним, что функция `(\\)` вычисляет разность двух списков. Определение предиката `isSolution` очевидно. Полный текст решения задачи на Хаскеле приведен выше.

Мы уже упоминали, что класс `SearchProblem` приведенный ранее, реализует простой поиск в ширину. Можно получить обобщение абстрагированием операции добавления сгенерированных новых состояний в список состояний; для этого мы вводим в определения `space` и `solutions` параметр-функцию, контролирующую добавление новых состояний в пространство поиска. Возможная реализация показана в следующем фрагменте кода.

```
type Space m s = [([m],s)]
type Strategy m s = Space m s → Space m s → Space m s

class SearchProblem s m where

  trans :: s → [(m,s)]
  isSolution :: ([m],s) → Bool
  space, solutions :: Strategy m s → s → Space m s

  space f s = expand f (step ([],s))
    where expand f [] = []
          expand f (s:ss) = s:expand f (f (step s) ss)
```

```
        step (ms,s) = [(ms++[m],t) | (m,t) ← trans s]

  solutions f = filter isSolution · space f

dfs = (++)
bfs = flip dfs
```

Чтобы использовать обобщённый класс в нашем примере, нам нужно всего лишь передать экземпляру класса соответствующую поисковую стратегию функции `solutions`:

```
solution = solutions bfs (L,toys)
```

Для этого примера задачи поисковая стратегия не влияет на решение, но для других поисковых задач завершение поиска при использовании `bfs` более вероятно, чем при использовании `dfs`.

Примечание от переводчика:

Так как мультипараметрические классы типов являются расширением стандарта Haskell 98, для выполнения этой программы на GHC или GHCi следует задать в командной строке опцию `-fglasgow-exts`, а в HUGS — опцию `-98`.

Так же неплохо бы указать импорт модуля `List`, который содержит определения нужных программе функций `(\\)` и `sort`.

6. Выводы

Давайте обобщим опыт, полученный от этого упражнения. Большинству студентов понравился характер задачи-головоломки, хотя довольно многие и испытали трудности с ее решением и потратили значительное время на отладку своих программ. Одной из проблем было некорректное использованием термов, приводившее к тому, что интерпретатор лишь отвечал `No`. Еще одной ошибкой было непонимание разницы между термами и предикатами. В некоторой степени за эту ошибку, возможно, ответственны совершенно разные лексические соглашения в Хаскеле и Прологе: в Прологе имена переменных начинаются с прописной буквы, а в Хаскеле — со строчной, тогда как имена конструкторов термов в Прологе начинаются со строчной буквы, а в Хаскеле — с прописной.

Некоторые студенты пытались обойти свои проблемы, кодируя знание о решении в их программах, например, зафиксировав количество ходов вперёд и назад в постановке задачи. Некоторые решения, сданные студентами, были близки к показанному в первом листинге на Прологе, отличаясь, в основном, выбором представления термов и определением предиката перехода. Неправильное представление термов было главной проблемой в программах, которые не запускались вообще или выдавали неверные результаты.

Чтобы получить обратную связь относительно подхода на Хаскеле, мы попросили нескольких выпускников решить задачу также на этом языке. Все они

выдали нам определение класса `SearchProblem`. Те студенты, которые уже получили верные решения на Прологе, сообщили, что решить задачу на Хаскеле было так же легко, как и на Прологе. Другие, чьи решения на Прологе были неидеальны, почувствовали, что написать программу на Хаскеле было легче. Они также сообщили, что система типов помогла в разработке решения и отладке программы.

Из нашего опыта решения данной задачи на обоих языках мы убедились, что благодаря системе типов реализация поисковых задач на Хаскеле оказывается в конечном итоге легче, чем на Прологе. В наибольшей степени поддерживает это впечатление такая важная особенность Хаскеля, как наличие мультипараметрические классов, поскольку она позволяет абстрагировать общую схему решения в класс и повторно использовать его в других задачах.

7. Благодарности

Автор благодарит Маттиаса Фелляйсена за его ценные советы и замечания, которые помогли улучшить эту статью. Также мы весьма благодарны студентам кафедры языков программирования, сообщившим о своём опыте с Хаскелем, Прологом, типами и т. д.

Список литературы

[1] *Claessen K., Ljunglöf P.* Typed logical variables in haskell. — 2000.

[2] *Erwig M.* Cs 581: Programming languages. graduate course. department of computer science, oregon state university. — URL: http://www.cs.orst.edu/~erwig/old/cs581.f01 (дата обращения: 31 января 2011 г.). — 2001.

[3] Haskell 98 Language and Libraries: The Revised Report. — Cambridge University Press2003. — May.

[4] *Haynes C. T.* Logic continuations // — *J. Log. Program.*1987. — June. — Vol. 4. — Pp. 157–176.

[5] How to design programs: an introduction to programming and computing / M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi. — Cambridge, MA, USA: MIT Press, 2001.

[6] *Paulson L. C.* ML for the working programmer (2nd ed.). — New York, NY, USA: Cambridge University Press, 1996.

[7] *Rabhi F., Lapalme G.* Algorithms; A Functional Programming Approach. — 1st edition. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[8] *Spivey M., Seres S.* Embedding Prolog In Haskell. — URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.8710 (дата обращения: 31 января 2011 г.).

[9] *Wadler P.* How to replace failure by a list of successes // Proc. of a conference on Functional programming languages and computer architecture. — New York, NY, USA: Springer-Verlag New York, Inc., 1985. — Pp. 113–128.