JFP **14** (3): 253–261, May 2004.
DOI: 10.1017/S0956796804005040 Printed in the United Kingdom

EDUCATIONAL PEARL

Escape from Zurg: an exercise in logic programming

MARTIN ERWIG

School of EECS, Oregon State University, Corvallis, OR 97331, USA
(*e-mail:* erwig@cs.orst.edu)

Abstract

In this Pearl we illustrate with an example that modern functional programming languages like Haskell can be used effectively for programming search problems, in contrast to the widespread belief that Prolog is much better suited for tasks like these.

1 Introduction

It is a common belief that Prolog is *the* language of choice to solve search problems. One strong point of Prolog is its built-in backtracking ability, which can save considerable work in handling search problems. On the other hand, Haskell (Peyton-Jones, 2003) provides a powerful type system, higher-order functions, and lazy evaluation. We want to illustrate in this paper that these elements taken together make it as easy in Haskell (and maybe even easier) to express and solve search problems as it is in Prolog. For example, lazy evaluation facilitates the concise description of the search space because the specification of an infinite data structure – the search tree – can be written down without running into nontermination as long as only a finite part of it is processed. This idea is not new; it has been described by Wadler (1985) before. However, rewriting this encoding for every search problem from scratch is tedious, error-prone, and can distract from the very search problem that is to be implemented. In Haskell, the concept of type classes offers a clean way to formulate the solution once and reuse it in different instances. Additionally, Haskell data types allow (and enforce to some degree) formulation of the problem in an adequate fashion.

An alternative approach is to embed a Prolog-like language into a functional language. This has been demonstrated in Seres & Spivey (1999) and Classen & Ljunglö (2000) for Haskell, and in Haynes (1987) for Scheme. However, our goal is to express search problems functionally without resorting to a multi-paradigm approach.

The example that we want to consider is a homework problem that we have given in a graduate level course on programming languages (Erwig, 2001). The problem was one of several exercises to practice programming in Prolog. After observing

that many students had problems manipulating term structures in Prolog (after already having learned to use data types in Haskell) and spending a lot of time on debugging, the question arose whether it would be as difficult to develop a solution for this problem in Haskell. This programming exercise worked well, and we report the result in this paper.

In the rest of this paper we will describe the requirements for teaching the programming of search problems in Haskell in Section 2. The example problem is described in Section 3. In Section 4 we show an example solution in Prolog. Section 5 presents the Haskell solution to the problem. Conclusion given in Section 6 complete this paper.

2 Teaching search programming in Haskell

To teach search programming in Haskell as proposed here, students should already have a solid understanding of essential functional programming concepts, such as, recursion, lists, and higher-order functions. In addition, students have to understand type classes, data types, and lazy evaluation, because these are used to create the modular solution.

First, a type class is used to separate the generic description of search problems from a particular problem instance. In particular, we make use of multi-parameter type classes to parameterize a search problem by the type of states and the type of moves. Multi-parameter type classes need not necessarily be known in advance. In fact, the `SearchProblem` class can serve as a motivating example to introduce multi-parameter type classes. The class itself can be developed in steps. Initially, a single-parameter version can be defined that is parameterized only over the type of search states. Then, recognizing that for some search problems, like the one discussed here, the solution states are not as interesting as the moves that lead to them, a generalization to two type parameters can be performed.

Second, data types are employed in the chosen example to create a model of the application. Data types provide a higher-level means of modeling the application than encoding all the information pieces by plain lists and tuples. Again, the encapsulation of the general search process in a type class is helpful since it allows us to focus completely on modeling the application because we do not have to deal with the search. This situation is similar in Prolog where the search procedure is built into the language. However, compared to Prolog terms, Haskell data types provide as a typed representation immediate feedback about illegal combinations of moves and states, which otherwise can cause a lot of debugging effort in untyped respresentations.

Third, knowledge of lazy evaluation is required to understand how the potentially infinite state space can be described in Haskell. A simple breadth-first search is implemented by creating a list of states through repeated appending of successor states. Depending on the focus and available time for dealing with search programming one might want to discuss this aspect in more depth. For example, it is relatively easy to generalize the class by parameterizing the construction of the search space by a search strategy. Again, since the search problem is isolated in a type class, this discussion does not affect the modeling of applications.

The implementation of search problems is discussed only in a few ML or Haskell textbooks. For example, Paulson (1996) describes the implementation of search programming in ML in the context of a theorem prover. Rabhi & Lapalme (1999) describe how to implement backtracking algorithms in Haskell. They employ an explicitly defined depth-first search algorithm and do not use type classes to separate the problem class from the applications. In particular, they do not distinguish a separate type of moves, which makes the described approach inappropriate for the example problem discussed here. Felleisen *et al.* (2001) describe in their book a similar example, the problem of three missionaries and cannibals crossing a river. This example is given as an exercise in the context of programming with generative recursion and accumulators, which are discussed in great depth as a means to retain context information in recursive function definitions.

3 The example problem

The problem to be solved was called "Escape from Zurg" and reads as follows:

> Buzz, Woody, Rex, and Hamm have to escape from Zurg.[a] They merely have to cross one last bridge before they are free. However, the bridge is fragile and can hold at most two of them at the same time. Moreover, to cross the bridge a flashlight is needed to avoid traps and broken parts. The problem is that our friends have only one flashlight with one battery that lasts for only 60 minutes (this is not a typo: **sixty**). The toys need different times to cross the bridge (in either direction):
>
> | Toy | Time |
> |---|---|
> | Buzz | 5 minutes |
> | Woody | 10 minutes |
> | Rex | 20 minutes |
> | Hamm | 25 minutes |
>
> Since there can be only two toys on the bridge at the same time, they cannot cross the bridge all at once. Since they need the flashlight to cross the bridge, whenever two have crossed the bridge, somebody has to go back and bring the flashlight to those toys on the other side that still have to cross the bridge.
>
> The problem now is: In which order can the four toys cross the bridge in time (that is, in 60 minutes) to be saved from Zurg?
>
> [a] These are characters from the animation movie *Toy Story 2*.

Try to solve the problem in your favorite language.

4 A Prolog solution

Writing a Prolog program for solving the riddle is in principle a rather straightforward task – at least once it has been figured out how to represent the problem.

https://doi.org/10.1017/S0956796804005040 Published online by Cambridge University Press

As it turned out, this last aspect seemed to have been the major reason that quite a few students had problems with the assignment. The most difficult part for the students was to find an appropriate term representation for the states of the search problem, that is, the position of toys on either side of the bridge and the position of the flashlight. In particular, the two prevailing mistakes were to use too complex term structures or predicates and to use terms inconsistently, or even in a few cases to confuse predicates and terms. Several programs did not terminate. The reference solution is shown in Figure 1 to serve as a comparison with the Haskell solution to be developed in the next section.

The idea of the Prolog program is to represent an intermediate state of a bridge crossing by facts of the form `st(P,L)` where `L` is a list giving the toys that are currently on the left side of the bridge and where `P` is a flag that indicates the position (left or right side) of the flashlight.[1] The predicate `move/4` generates movements in its third argument; a movement to the right is generated if the flashlight is on the left side of the bridge and vice versa; `move` also relates the old state (first argument) to the newly reached state (second argument). The last argument gives the time required for the move. In the case of a movement to the right, the time is determined by the additional predicate `cost/2` that computes the maximum time needed by a group of toys. Any such possible group of toys to move to the right is computed by the predicate `split/3`, which computes lists of length 2, which are sorted to avoid redundancy caused by representing groups of toys as lists. In a move to the left it makes only sense to send back one toy. Therefore, the definition of `move` for that case uses the predefined `member/2` predicate and computes the time by simply looking into the table `time/2`. Finally, the `trans/4` predicate basically generates all possible bridge crossings together with the required time whereas the `cross/2` predicate formulates the search problem by giving the initial and final configuration of the search space.

5 The Haskell solution

We give the Haskell solution in two steps. First, we extract the general structure of the search problem and capture it in the definition of a type class. Second, we present the program for solving the puzzle as an instance of that class.

The main elements in the problem are a *state* (to represent intermediate stages of bridge crossings) and *moves* (to represent transitions between states, which are in this case bridge crossings). Therefore, we have defined a type class `SearchProblem` with two parameter types `s` and `m`. Next, we consider what member functions are needed for the class `SearchProblem`.

To build the complete search space starting from some (initial) state `s`, a function is needed that describes which new states can be reached from `s`. In general, it is not only the final state that is of interest (in fact, we know this state already in the

[1] In fact, more common among the students' solutions was the approach to represent *two* groups of toys on both sides of the bridge, but we found that although this redundancy might help to think about the problem, keeping the invariant was a common source of errors.

https://doi.org/10.1017/S0956796804005040 Published online by Cambridge University Press

```
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).
```

Fig. 1. Prolog solution for the Zurg riddle.

given example, namely all toys on the other side). Rather, the sequence of moves that lead to this state is needed, too. Therefore, we haved added a function to the type class that computes for a state a list of possible moves and the new states to which they lead:

```
trans :: s -> [(m,s)]
```

https://doi.org/10.1017/S0956796804005040 Published online by Cambridge University Press

```
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
```

Fig. 2. The `SearchProblem` type class.

By repeatedly applying `trans` to the fringe of a search tree the complete search space for a problem can be constructed. This search space is represented by an element of type `Space m s` and is constructed by a function `space` that maps a state to a list of all nodes of the search space. Each node (that is, state) is paired with the list of moves that lead to it.

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

space :: s -> Space m s
```

Since the search space is completely represented by the initial state and the function `trans`, the function `space` is a derived member function in the class `SearchProblem`. The definition of `space` makes essential use of lazy evaluation: without a terminating condition `space` refers (indirectly through `expand`) to itself; under strict evaluation this definition would, in general, not terminate. For the present example this means that it is enough to consider just two cases in the definition of `trans` (see Figure 3); an additional definition

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

is *not* needed, although such a condition was required in the Prolog program (cf. the first clause for the predicate `trans`).

In general, the solutions to a search problem are given by a subset of its states. The decision is made by a predicate on states and their generating moves:

```
isSolution :: ([m],s) -> Bool
```

With this predicate another class member `solutions` can be defined, which has the same type as `space` and which simply yields the subset of states that are considered solutions by the predicate `isSolution`. The definition of the type class `SearchProblem` is summarized in Figure 2.

Having defined the solution schema, the Haskell program for solving the riddle requires only the modeling of the problem. The most important design decisions are the definitions of the types `BridgePos` and `Move` because they are related by

https://doi.org/10.1017/S0956796804005040 Published online by Cambridge University Press

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

Fig. 3. Haskell solution for the Zurg riddle.

an instance definition for the class `SearchProblem`. In `BridgePos` we represent the position of the flashlight by a constructor `L` or `R` and the toys that are on the left side of the bridge by a list, just like in the Prolog implementation. A move is either a move of a group of toys from left to right or a backward move from right to left by just one toy. Both kinds of moves are captured by the type `Move`, which is defined through an `Either` data type, which is predefined in Haskell and which contains the constructors `Left` and `Right` to represent disjoint sum types.

Apart from defining types for representing the objects in the program, the main part is the instance definition of the `SearchProblem` type class, which means to give a definition for `trans` and `isSolution`. To this end, we have defined three auxiliary functions: `forw` and `backw` for computing toy moves and `duration` for computing the total time of a crossing, that is, for a sequence of moves. Note that the function `(\\)` computes the difference of two lists. The definition of the `isSolution` predicate is obvious. The complete Haskell solution is shown in Figure 3.

https://doi.org/10.1017/S0956796804005040 Published online by Cambridge University Press

```
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
```

Fig. 4. Generalized `SearchProblem` type class.

We have already mentioned that the `SearchProblem` type class from Figure 2 implements a simple breadth-first search. A generalization can be obtained by abstracting from the append operation that is used to add newly generated states to the list of states, that is, we introduce a function parameter into the definition of `space` and `solutions` that controls the addition of new states to the space. A possible implementation is shown in Figure 4.

To use this generalized type class for the example problem, we only have to pass a corresponding search strategy to the `solutions` function, for instance:

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

For the example problem, the search strategy does not affect the solution, but for other search problems, termination is generally more likely under `bfs` than under `dfs`.

6 Conclusions

Let us first summarize our experience with the shown programming exercise. Most students seemed to like the puzzle style of the assignment, although quite a few students had problems during the development of their solution and had to spend a considerable amount of time debugging their programs. A particular problem was to spot illegal uses of Prolog terms that showed up in the interpreter just through the answer `No`. Another mistake was to confuse terms and predicates. To some degree, the completely different lexical conventions in Haskell and Prolog are probably responsible for this confusion: Variables start with an uppercase letter in Prolog, and with a lowercase letter in Haskell, whereas term constructors start with a with a lowercase letter in Prolog, and with an uppercase letter in Haskell.

https://doi.org/10.1017/S0956796804005040 Published online by Cambridge University Press

Some students tried to work around their problems by coding knowledge about the solution into their programs, for example, fixing the number of forward and backward moves in the problem representation. Some of the solutions handed in by students were similar to the one shown in Figure 1, differing mainly in the chosen term representation and in how the transition predicate was defined. An incorrect term representation was the main problem for those program that did not run at all or that were computing incorrect results.

To obtain feedback about the Haskell approach, a couple of graduate students were asked to solve the problem also in Haskell. All they were given was the definition of the `SearchProblem` type class. Those students who already got the Prolog solution correct reported that it was as easy in Haskell to come up with a solution as in Prolog. Others who had non-perfect Prolog solutions felt it was easier to write the Haskell program than the Prolog program. They also reported that the type system was helpful in designing the solution and in debugging the program.

From our experience with solving the example problem with both languages, we believe that Haskell's type system makes it eventually easier to implement search problems in Haskell than in Prolog. The most important feature of Haskell that supports this impression is the availability of multi-parameter type classes, because we can abstract the general solution schema in a type class and reuse it for other problems.

Acknowledgments

The author thanks Matthias Felleisen for his valuable hints and remarks that helped to improve this paper. Many thanks also go to the students of the programming languages class who provided a lot of feedback about their experience with Haskell, Prolog, types, etc.

References

Claessen, K. and Ljunglö, P. (2000) Typed logical variables in Haskell. *Haskell Workshop. Electrical Notes in Theoretical Computer Science*, **41**(1).

Erwig, M. (2001) *CS 581: Programming Languages*. Graduate Course, Department of Computer Science, Oregon State University. `http://www.cs.orst.edu/~erwig/old/cs581.f01`.

Felleisen, M. Findler, R. B., Flatt, M. and Krishnamurthi, S. (2001) *How to Design Programs – An Introduction to Programming and Computing*. MIT Press.

Haynes, C. T. (1987) Logic continuations. *J. Logic Program.* **4**, 157–176.

Paulson, L. C. (1996) *ML for the Working Programmer (2nd ed.)*. Cambridge University Press.

Peyton-Jones, S. (2003) *Haskell 98 Language and Libraries*. Cambridge University Press.

Rabhi, F. and Lapalme, G. (1999) *Algorithms: A Functional Programming Approach*. Addison-Wesley.

Seres, S. and Spivey, M. (1999) Embedding Prolog in Haskell. *Haskell Workshop*. Technical Report UU-CS-1999-28, Universiteit Utrecht.

Wadler, P. (1985) How to replace failure by a list of successes. *Proc. Conf. on Functional Programming and Computer Architecture. Lecture Notes in Computer Science 201*, pp. 113–128. Springer-Verlag.