CSci 2041: Advanced Programming Principles
Programming Techniques for Treating Search

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

Why Special Techniques for Programming Search?

The solution to many computational problems can be seen as picking a set of actions to get to a "goal state"

- solving puzzles or playing games
- handling reasoning tasks
- developing AI-style plans

Indeed, this is the first cut at solving pretty much any computational problem

After understanding the problem structure, we may come up with simple, efficient algorithms, but there is no guarantee

Thus, we often have no option but to search the solution space, possibly using heuristics to get good practical performance

The question: how can we encode such search in programs?

Approaches to Treating Search

A general strategy for programming search is the following

- Use some method to pick an alternative when you have to make a choice amongst different possibilities

  Typically, heuristics that embody domain knowledge are used to order options

- Retain enough information so as to be able to "backtrack" on the choice if it does not work out

- Use the information retained to try a different path if the choice does not actually work out

In principle, we can explicitly maintain information realize this style of computation but it can get tedious

We will look at two programming techniques that enable us to do this more easily

Using Exceptions to Realize Search

The main questions in realizing search is how to proceed in the case of success and in the case of failure

The exceptions feature gives us a means for treating these questions

- We make a choice assuming things are going to work out, i.e. the normal return takes care of the success path
- However, we also prepare for failure by setting up an exception handler that will try the alternatives
- When we encounter a deadend, we “give up” by raising an exception and let the handler take care of the rest

Note that the maintenance of state needed for backtracking is taken care of by the language support for exceptions

Using Exceptions to Realize Search (Example)

Consider the following *subset sum* problem

> *Given a list of numbers, find a sublist that sums up to another given number*

This is known to be an *NP-complete* problem, i.e., we currently know of no better method than search to solve it

Here is a way to structure the search in this case

- As we recurse down the list, we choose whether or not to include the head, transforming the problem accordingly
- At the point we make the choice, we set up an exception handler for the other case
- If we cannot solve the subproblem eventually, we raise an exception and let the relevant handler take care of it

Solving Subset Sum Using Exceptions

The OCaml code that realizes the described structure

```
exception Search_Failure
exception Illegal_Sum

let rec find_subset l n =
  match (l,n) with
  | (_,0) -> []
  | ([],n) -> raise Search_Failure
  | ((m::l'),n) ->
      if (n < m) then find_subset l' n
      else try m :: (find_subset l' (n-m)) with
                Search_Failure -> (find_subset l' n)

let subset_sum l n =
  if n < 1 then raise Illegal_Sum
  else try (find_subset l n) with
          Search_Failure ->
               (Printf.printf "No solutions\n"; [])
```

Realizing Search Using Two Continuations

To transform functions into a tail-recursive form earlier, we used continuations that indicated what to do with the result

For search problems, we also need to indicate what to do if there *is not* going to be a result

To capture this situation, we use *two* continuations:

- a *success continuation* that determines what to do when there is success
- a *failure continuation* that determines how to proceed in the case of a failure

Solving Subset Sum Using Continuations

An alternative solution to the subset sum problem

```
let rec find_subset' l n succ fail =
  match (l,n) with
  | (_,0) -> (succ [])
  | ([],_) ->  fail ()
  | ((m::l'),n) ->
     if (n < m) then find_subset' l' n succ fail
     else
       find_subset' l' (n-m) (fun l -> succ (m::l))
                (fun () -> find_subset' l' n succ fail)

let subset_sum' l n =
   if (n < 1)
   then raise Illegal_Sum
   else
     (find_subset' l n (fun l -> l)
       (fun () -> Printf.printf "No solutions\n"; []))
```

General Comments about Search-Based Computation

We have only scratched the surface of this style of computation

- We have only talked about *depth-first search with backtracking*

  Other strategies like *breadth-first* and *best-first* exist; may be explored in an AI course for example

  Our programming techniques can be useful there too

- We have not considered the use of heuristics in search

  E.g. simplifying the problem first, ordering alternatives

- Our programming framework simplifies but does not obviate the treatment of search

  Frameworks exist that make the treatment of search completely implicit, e.g., see logic programming languages