Assignment 3b

Controlling the Future with Functions

All Parts Due: 11:59 PM, Monday, February 15, 2016

Ideally, get it done by Sunday night.

Normally, functions compute some answer and just return it to the caller (and thereafter to the rest of the running program). But experience with functional programming suggests an alternative. The purpose of this homework is to explore this alternative, because this pattern of control flow can be useful even in non-functional-programming situations.

Moreover, this assignment is all about treating functions as another kind of value, specifically something you can pass around and manipulate. This viewpoint is an essential aspect of functional programming (an approach programming that Haskell typifies), but something most students in the class have little experience with.

Despite the length of this writeup, the core of this assignment can be completed by writing less than 15 lines of code, and because of symmetry in the problem, significant insight is only necessary for half of that code, the second half is a reprise of the first class. (These figures do not include any new inputs you define purely for testing purposes.)

Preliminaries

You may work on this assignment with a partner (preferred) or by yourself. If you work with a partner, you must follow pair programming rules (see the PairProgramming page on the wiki for full rules). Both members of the pair must fully understand all jointly submitted work. You are not required to work with your partner from lab. You must use the same partner for Assignment 3a and 3b.

Files for this assignment are at https://svn.cs.hmc.edu/cs131/spring16/given/hw3. Copy them in the usual way (see CopyingAssignmentFiles on the wiki for a reminder of how).

Required Code Quality

The same rules apply as in the last homework. Write clear code, and appropriate comments. You code must compile and type check, etc.

Background: Continuation Functions

A continuation function is a function value that represents “what to do with the result of the current sub-computation”. The key idea is to take a function \( f \) and add a continuation function as an extra argument; if we call this argument \( k \), then rather than return the answer \( v \) directly to the caller of \( f \), we provide the answer to the continuation function (i.e., \( f \) finishes by calling \( k(v) \)).
As a fairly trivial example, compare the “direct style” definitions:

```haskell
add :: Int -> Int -> Int
add x y = x+y

mult :: Int -> Int -> Int
mult x y = x*y
```

with the “continuation-passing style” (CPS) definitions:

```haskell
addAndThen :: Int -> Int -> (Int -> a) -> a
addAndThen x y k = k (x+y)

multAndThen :: Int -> Int -> (Int -> a) -> a
multAndThen x y k = k (x*y)
```

The `addAndThen` and `multAndThen` functions have an extra “what-to-do-with-the-result” parameter `k`.

Then instead of writing `myfunc (add (mult 2 3) (mult 6 7))`, we could write

```haskell
multAndThen 2 3 (
    x -> multAndThen 6 7 (
        y -> addAndThen x y myfunc)
)
```

One way to read this line is “multiply 2 and 3 and let `x` be that product; then multiply 6 and 7 and let `y` be that product; then add `x` and `y` and pass the sum to `myfunc`.” It should be clear that the continuation argument to each call to `mult` and `add` really is an explicit representation of “the rest of the computation”. After the first product `2*3` we still need to do a multiply and add and call `myfunc`, and that’s what its `k` argument does; after the second product `6*7` we still need to do an addition and call `myfunc`, and that’s what its `k` argument does; etc.

But so far, it might seem like passing a continuation function is pointless make-work. Why would anyone write code in such a convoluted and confusing way? It turns out that this approach, where “what happens next” is explicitly represented is surprisingly powerful.

In particular, turning “the rest of the computation” from implicit concept to an explicitly represented value lets us control the execution of a program in tricky ways. For example, consider the following multiply-a-list-of-integers function:

```haskell
product :: [Integer] -> Integer
product [] = 1
product (x : xs) = x * product xs
```

One way to optimize this code is observe that we don’t have to look at the rest of the list if we encounter a zero in the list.

```haskell
product :: [Integer] -> Integer
product [] = 1
product (0 : xs) = 0
product (x : xs) = x * product xs
```
We can write equivalent code (without changing the type) by writing code that uses an explicit continuation argument, and then providing it with a continuation that says “the thing to do with the final answer is just to return it.”

\[
 product :: [Integer] -> Integer
 product xs = productAndThen xs (\x -> x)
 where productAndThen :: [Integer] -> (Integer -> Integer) -> Integer
      productAndThen [] k = k 1
      productAndThen (0:xs) k = k 0
      productAndThen (x:xs) k = productAndThen xs (\tailprod -> k (tailprod * x))
\]

Make sure you understand how this code works. The last line can be read “If I am asked to take the product of the list \(x:xs\) and pass the result to \(k\), then I’ll take the tail \(xs\), compute its product \(tailprod\), use that to find the product of the whole list by multiplying by \(x\), and then pass this along to the continuation \(k\) (which tells us what to do with the product of the whole list \(x:xs\)).”

In essence we are using the last argument as way to put together the final computation we want to do. The multiplication isn’t actually done until we hit one of the base cases, then we provide our continuation function with 0 or 1. Thus, if given \([7, 42, 54]\) by the time it gets to the end of the list, \(k\) is the function:

\[
(\text{tailprod3} -> \\
 (\text{tailprod2} -> \\
 (\text{tailprod1} -> \\
 (\text{x} -> x) (\text{tailprod1} * 7)) \\
 (\text{tailprod2} * 42)) \\
 (\text{tailprod3} * 54))
\]

which is simplifies (via beta reduction!) to the function\[
\text{tailprod3} -> ((\text{tailprod3} * 54) * 42) * 7
\]

and when we pass in 1 to this function (i.e., \(k\ 0\)), it calculates \(((1 * 54) * 42) * 7\), which is exactly the product our (simpler!) recursive code would have computed. If instead we had had the list \([7, 42, 54, 0, 3, 12]\), on finding the zero (second case in our pattern match), it would have computed \(((0 * 54) * 42) * 7\), producing zero and skipping the rest of the list, again just like our simpler version.

But the cool part is still to come. Both versions are optimized to take advantage of the fact that anything times zero will be zero. This helps a lot for inputs like \([0, 1, 2, 3, 4, 5]\) but not at all for inputs like \([5, 4, 3, 2, 1, 0]\): even though the product will be zero, we still do all the multiplications as the recursive calls return. In the direct-style code, it’s a bit tricky to do better.¹

¹ Without using exceptions, or doing extra checks-for-zero, which might be slower than just doing the unnecessary multiplications.
But in the CPS version, we can simply write:

\[
\text{product} :: \mathbb{Z} \rightarrow \mathbb{Z} \\
\text{product} \; \text{xs} = \text{productAndThen} \; \text{xs} \; (x \rightarrow x) \\
\quad \text{where} \quad \text{productAndThen} :: \mathbb{Z} \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow \mathbb{Z} \\
\quad \; k = k \; 1 \\
\quad \; \text{productAndThen} \; (0:xs) \; k = 0 \quad \quad \quad \quad \text{-- The only line that changed} \\
\quad \; \text{productAndThen} \; (x:xs) \; k = \text{productAndThen} \; \text{xs} \; (\text{tailprod} \rightarrow k \; (\text{tailprod} \; \times \; x))
\]

We know that \(k\) will be a composition of multiplications-yet-to-be-performed. So since we know the result of all these multiplications will be zero, we can ignore all these pending computations and simply return zero. No multiplications at all occur if there’s a zero in the list.²

Even this example is still a bit contrived; I’m sure you could come up with other, simpler, ways of writing \text{product} to avoid doing multiplications when there’s a zero. But it does demonstrate how continuations can be used to organize control flow. The following section discusses an even more useful example where you want continuations not because you want to avoid doing the rest of the computation, but because you’d like to do it more than once!

**Takeaways**

There are several takeaways to for you from this code and approach:

- We can actually *explicitly* represent “the future”, specifically, the code that cares about what we’ve just calculated. Instead of passing our result back as a return value, we pass it forward to this “consumer” function.
- We can *manipulate* this “the future” function. The last line of \text{multAndThen} does exactly that, extending it to make it do one more multiplication.
- We can *discard* our planned future and make a new, different, future. You cannot easily do this with normal recursion. In normal recursion, we always return to the code that will take our return value and do whatever it planned to do with it.
- Likewise, although we didn’t do it in this example, we can *repeat* our planned future. We can check the value that we’ll eventually calculate and if we don’t like the final result, we can decide to do something different (e.g., feed a different result into the future and see if that makes for a better future for us).

For most students in this class, this whole concept is *weird*. You’ve never written code like this and it takes a bit to wrap your head around it. But it shows how being able to manipulate functions and pass them around allows you to express ideas that would be hard to do otherwise. And now it becomes part of your toolbox. C++ and Python both let you build new functions and treat functions as values so you could write similar code there. It’s not just a Haskell-only thing.

---

² We do however build up a function and in this case that’s probably much more expensive than a simple multiplication, but you can imagine a more expensive operation where we really might save work.
**WARNING:** If you’re sketchy about this example and these properties, it’s probably best to take the time to think about it and get help from your peers, the grutors, or Melissa, because what you actually have to do for this assignment is much much easier if you properly understand how this approach works. Yadda-yadda-ing it and hoping you’ll figure it out as you dive deeper is not a good strategy.

---

**Solving SAT with Backtracking Search**

Finally, we get to the actual task of this assignment, writing a program that solves the **boolean satisfiability problem** (SAT), also known as a SAT solver.

Here is an example SAT problem: Can you come up with values for \( p, q, r, \) and \( s \) (i.e., true or false) to make the following formula true:

\[
(\neg p \lor \neg r \lor \neg s) \land (q \lor r \lor \neg s) \land (p \lor \neg q \lor s) \land (p \lor r \lor s) \land (\neg p \lor q \lor \neg r)
\]

Also (for the converse problem, falsifiability), can you find values for the variables that will make the formula false.³

More formally, a propositional formula is said to be **satisfiable** if there exists a way of setting propositional variables to true or false that makes the whole formula true. It is said to be **falsifiable** if there are values for its variables that makes the formula false. Finally, it is said to be a **tautology** if all possible choices of values for the variables makes the formula true (i.e., if it is not falsifiable).

For example, \((p \lor q) \land \neg p\) is both satisfiable (e.g., make \( p \) false and \( q \) true) and falsifiable (e.g., make \( p \) and \( q \) both false). The formula \( p \lor \neg p\) is satisfiable and not falsifiable (and not being falsifiable makes it a tautology).

One way to determine whether a formula is satisfiable or falsifiable is to look at all possible assignments. This is a bit wasteful, since if the formula is something like \( p \land \neg p \land (q \lor r \lor \neg s \lor t)\) we would like to stop trying to satisfy the formula almost immediately, rather than trying all 32 possibilities. Therefore, we can try to find a satisfying or falsifying assignment by search, short-circuiting whenever we realize that our goal is impossible (e.g., after seeing \( p \land \neg p\)).

If we represent propositional formulas with the type

```haskell
data Prop = Var String
           | Not Prop
           | Or  Prop Prop
           | And Prop Prop
           | Imply Prop Prop
```

then the following code for determining satisfiability seems initially plausible (where an \( \text{Asn} \) is an “assignment” (lookup table) from variable names to booleans.)⁴

---

³ If you’re wondering, \( p \mapsto \text{false}, q \mapsto \text{true}, r \mapsto \text{true}, s \mapsto \text{true} \) will satisfy the formula and \( p \mapsto \text{true}, q \mapsto \text{true}, r \mapsto \text{true}, s \mapsto \text{true} \) falsifies it. You might want to think about whether any other assignments would work.

⁴ You may want to check the definition of Haskell’s Maybe type in Learn You a Haskell.
-- Determines whether there is a way to make the given proposition
-- true, given an assignment specifying truth values for the
-- variables we have seen so far.
satisfy :: Asn -> Prop -> Maybe Asn

satisfy asn (Var v) =
    case Map.lookup v asn of
        Nothing -> -- We can make the formula (consisting of just
                      -- this variable) true by making the variable true.
                      Just (Map.insert v True asn)
        Just True -> -- We already decided this variable should be
                      -- true, so return the assignment unchanged
                      Just asn
        Just False -> -- Oops...we already decided this variable must be
                      -- false, so we cannot make the given formula true.
                      Nothing

satisfy asn (And p1 p2) =
    -- Try to find an assignment satisfying both p1 and p2.
    case satisfy asn p1 of
        Just asn' -> satisfy asn' p2
        Nothing -> Nothing

satisfy asn (Or p1 p2) =
    -- Try to satisfy p1 starting with assignments asn.
    -- If that is impossible, come back and try to satisfy p2.
    case satisfy asn p1 of
        Just asn' -> Just asn'
        Nothing -> satisfy asn p2

satisfy asn (Not p) =
    -- There is a way to make (Not p) true iff there is a way to make
    -- p false (without changing the truth values of any variables
    -- appearing in asn).
    falsify asn p

satisfy asn (Imply p1 p2) =
    -- Use an equivalence of classical propositional logic
    satisfy asn (Or (Not p1) p2)

falsify :: Asn -> Prop -> Maybe Asn
    -- defined analogously (not shown)

Unfortunately, if we try to check whether \((p \lor q) \land \neg p\) is satisfiable by running

    satisfy Map.empty (And (Or (Var "p") (Var "q")) (Not (Var "p")))

we wrongly get the answer Nothing, rather than Just and an assignment that makes p false and q true.

6
The problem is that this code signals success by returning, but we expect a function to return only once. A sub-expression (like \( p \lor q \)) might be satisfiable in several ways; if our first choice can’t be extended to satisfy the rest of the proposition, there’s no way to “un-return” and request a different satisfying assignment.

---

MAKE SURE YOU UNDERSTAND WHAT WENT WRONG BEFORE CONTINUING ON!

There are several ways to fix this problem, but a large number of them boil down to turning “the rest of the proposition” into an extra parameter. One of the simplest methods is to add a \textit{continuation} argument. In essence, we’re saying “this looks good to me, but is ‘future me’ okay with it?”. The continuation function is thus a function \( \text{Asn} \rightarrow \text{Maybe Asn} \). It takes the variable assignments I’ve come up with and then returns the variable assignments that “future me” has figured out (or \textit{Nothing} if future me can’t satisfy the proposition).

---

These functions should return (Just asn’) if there is an extension of the given assignment that

1. satisfies/falsifies the given proposition
2. this extension can be further extended (by the continuation function) to satisfy the original proposition.

If not, they return the constant Nothing.

Intuitively, the continuation function will be checking that

- the "the rest of the proposition" is satisfiable or falsifiable respectively.

\[
\text{satisfyAndThen :: Asn} \rightarrow \text{Prop} \rightarrow (\text{Asn} \rightarrow \text{Maybe Asn}) \rightarrow \text{Maybe Asn}
\]

\[
\text{falsifyAndThen :: Asn} \rightarrow \text{Prop} \rightarrow (\text{Asn} \rightarrow \text{Maybe Asn}) \rightarrow \text{Maybe Asn}
\]

so that we can write

---

Check for satisfiability (returning an assignment of values to variables)

\[
\text{satisfy :: Prop} \rightarrow \text{Maybe Asn}
\]

\[
\text{satisfy p} = \text{satisfyAndThen Map.empty p (\text{\textbackslash asn} \rightarrow \text{Just asn})}
\]

Check for falsifiability (returning an assignment of values to variables)

\[
\text{falsify :: Prop} \rightarrow \text{Maybe Asn}
\]

\[
\text{falsify p} = \text{falsifyAndThen Map.empty p (\text{\textbackslash asn} \rightarrow \text{Just asn})}
\]

\textbf{To do:} That’s the theory anyway. In practice, \texttt{satisfyAndThen} and \texttt{falsifyAndThen} are unfinished. Your task is to fill in the undefined parts of these functions in \texttt{SolveSAT.hs} so that satisfiability and falsifiability and tautology testing work correctly.

\textbf{Warning:} Both \texttt{satisfyAndThen} and \texttt{falsifyAndThen} are analogous and you’ll be tempted to copy and paste code from \texttt{satisfyAndThen} to \texttt{falsifyAndThen}. But you
should check your logic carefully because although these functions are duals of each other, they are not literal copies of the same code. You should certainly try more tests than the few tests provided.

Concluding Thoughts — Backtracking and Continuations

- Languages like Prolog use exactly this same sort of (continuation-based) backtracking search as the main mechanism for executing programs.

- You may have seen other backtracking algorithms that worked without any need for continuations. For example, to place $n$ queens on a chessboard so that none of them attack each other, backtracking search is essentially:

To place queens in columns $i .. n$:
For each "legal" position in the $i$-th column
(given previously chosen positions for queens in columns $< i$):
  Recursively try to place queens in columns $(i+1) .. n$.
  If this succeeded, abort loop by returning success.
If the loop ends without a success, return failure.

Or, to color an $n$-node undirected graph with no two adjacent nodes having the same color:

To color nodes $i .. n$:
For each "legal" color for the $i$-th node
(given the previously chosen colors for nodes $< i$):
  Recursively try to color nodes $(i+1) .. n$.
  If this succeeded, abort loop by returning success.
If the loop ends without a success, return failure.

The difference is that in both examples, when you’re working on step $i$, it’s obvious what the rest of the problem is going to be: steps $i+1$ to $n$. We can therefore make a recursive function call to solve the entire rest of the problem. If this doesn’t work out, we can make a different choice and re-do the recursive call, to see if this helped.

Explicit continuation arguments arise when “the rest of the problem” is not as apparent. In the problem of Part 2, for example, when you’re working on satisfying some subexpression, you know what the subexpression is but not what the original expression looked like. So, there’s no way to make a recursive call to satisfy “the remainder of the original proposition that follows this particular subexpression”.

Instead, we provide a continuation function that does exactly what we need: when called, it will try to satisfy the remainder of the proposition after this subexpression.

- Satisfiability testing is one of the best known NP-complete problems. The backtracking search suggested above is exponential (even trying all possible combinations) in the worst-case.
However, in the past decade there have been quite startling advances in efficient heuristics for solving satisfiability problems. Even though the worst case is still bad, in practice it’s often possible to solve huge satisfiability questions very quickly! Thus, an increasingly popular approach to solving hard problems in other domains (cryptography, program verification, scheduling) is to translate the problem into a huge boolean expression, and then to run this through an efficient heuristic SAT solver.