University of Sussex
Informatics

Spring 2019

Limits of Computation
Feedback for Exercises 4
Dr Bernhard Reus

WHILE-programs, WHILE-decidability, WHILE-computability
(Lectures 7–9)

1. By writing a WHILE-program, show that $A = \{ 4, 6, 8 \} \subseteq \mathbb{D}$ is WHILE-decidable. Test your program by running it in hwhile.

   Answer:

```
decisionProc read X {
  switch X {
  case 4: result:= true
  case 6: result:= true
  case 8: result:= true
  default: result := false
  }
}
write result
```

2. Show that *any* finite set $A \subseteq \mathbb{D}$ is WHILE-decidable.
   Hint: assume without loss of generality that the finite set has n elements $A = \{ d_1, d_2, \ldots, d_n \}$ and write the decision procedure for this A.

   Answer:

```
dpF read X {
  switch X {
  case d₁: result:= true
  case d₂: result:= true
  ⋮
  case dₙ: result:= true
  default: result := false
  }
}
write result
```

Note that in hwhile we can directly inject trees d as *tree literals*, an extension of pure WHILE (we have not discussed in detail).

One way is to construct them explicitly with `cons` and `nil` in `WHILE`. (Alternatively we can deal with them in ASTs via `quote` without having to change the self-interpreter.)

Clearly, `dpf` always terminates. It is also rather obvious that it returns true if and only if, the argument `X` is equal to one of the d_i in A (for $1 \leq i \leq n$). And by definition of A this means that $[\![\texttt{dpF}]\!](d) = \mathit{true}$ if, and only if, $d \in A$ for any $d \in \mathbb{D}$.

Another solution (some students always suggest) would be to convert the set A of binary trees into a list and then run through this list as follows:

```
test2 read X {
   A:= [d1,d2,...,dn];
   while A {
       Y := hd A;
       if X=Y {
               result := true;
               A := nil          // loop exit
               }
       else  { A:= tl A
             }
       }
 }
write result
```

Again `di` means the representation of element `di` $\in \mathbb{D}$ in a `WHILE`-program. It is important to understand, however, that the list of elements is *not* an argument of the decision procedure. It is supposed to test membership in this (fixed) set.

In the seminars, students often wanted to make `A` the parameter of the decision procedure, but this is wrong. The decision procedure is for *deciding membership in* ***A***. Its argument is an arbitrary tree for which we want to know whether it is in `A` or not!

3. Show that, if $A \subseteq \mathbb{D}$ is `WHILE`-decidable then so is the complement of A, that is $\mathbb{D} \setminus A$.
   Hint: assume A is `WHILE`-decidable and thus we have a `WHILE`-program `p` that decides A. Now write a `WHILE`-program `q` that decides the complement of A. Of course, you can and should use `p`.

   Answer:

   Assume $A \subseteq \mathbb{D}$ is decidable. Thus, there is a program `p` that decides A. This means in particular that `p` always terminates.

Now let us construct a program `q` that decides $\mathbb{D}\backslash A$, the complement of A, as follows:

```
q read X {
    Y := <p> X;
    if Y  {
           result := false
           }
    else
           {
           result := true
           }
}
write result
```

We obviously get that `q` always terminates, since `p` does. We also see quickly that $[\![\mathtt{q}]\!]\,(d) = \mathit{true}$ if, and only if, $[\![\mathtt{p}]\!]\,(d) \neq \mathit{true}$ and therefore $d \in \mathbb{D}\backslash A$ iff $d \notin A$, which is exactly what we need.

4. Why is any `WHILE`-decidable set automatically `WHILE`-semi-decidable.

   Answer: Because a decision procedure is automatically a semi-decision procedure by definition. It is one that alsways terminates.

5. Write a `WHILE`-program `equal` that does not use the built-in equality (but can use all other extensions). The program `equal` takes a list of two trees `[l,r]` and tests whether the trees are equal, i.e. whether `l` = `r`. The function `equal` can be defined recursively as follows:

$$\mathtt{equal}([\mathrm{nil}, \mathrm{nil}]) = \mathtt{true}$$

$$\mathtt{equal}([\mathrm{nil}, \langle\, \mathtt{l.r}\,\rangle]) = \mathtt{false}$$
$$\mathtt{equal}([\langle\, \mathtt{l.r}\,\rangle, \mathrm{nil}]) = \mathtt{false}$$
$$\mathtt{equal}([\langle\, \mathtt{l.r}\,\rangle, \langle\, \mathtt{s.t}\,\rangle]) = \mathtt{equal}([\mathtt{l}, \mathtt{s}]) \wedge \mathtt{equal}([\mathtt{r}, \mathtt{t}])$$

   Unfortunately `WHILE` does not provide any recursive features. So your implementation has to traverse both input trees using a while-loop. One way to do this is to generalise the equality test to stacks of pairs of trees represented as a list of pairs of trees:

$$\mathtt{equalG}([]) = \mathtt{true}$$
$$\mathtt{equalG}([[\mathrm{nil}, \mathrm{nil}], \mathtt{S}]) = \mathtt{equalG}(\mathtt{S})$$
$$\mathtt{equalG}([[\mathrm{nil}, \langle\, \mathtt{l.r}\,\rangle], \mathtt{S}]) = \mathtt{false}$$

$$\texttt{equalG}([[\langle\,\texttt{l.r}\,\rangle,\mathrm{nil}],\texttt{S}]) = \texttt{false}$$

$$\texttt{equalG}([[\langle\,\texttt{l.r}\,\rangle,\langle\,\texttt{s.t}\,\rangle],\texttt{S}]) = \texttt{equalG}([[\texttt{l},\texttt{s}],[\texttt{r},\texttt{t}],\texttt{S}])$$

(If the input list contains more than two trees, those following the first two shall be simply ignored.) One can now define

$$\texttt{equal}(\texttt{L}) = \texttt{equalG}([\texttt{L}])$$

The definition of `equalG` is a so-called *tail-recursive* definition which means it can relatively straightforwardly be transformed into a while loop like so

```
equalG read L {
 res:= true;
 while L {
  X:= hd L;
  ...
  res:= ...
  if res
   { L := tl L  // tail recursive call
   }
   else
   { res:= false;
     L:= nil   // loop exit
     }
  }
}
write res
```

where some bits (represented by ...) have been left out for you to fill in. Test your program in `hwhile`.

Answer: See Canvas (program section).