Department of Computer Science and Engineering
University of California at San Diego

CSE 230
Winter 1998

Problem Set 2 Solutions

2. In mathematical logic, types were invented by Bertrand Russell to overcome a paradox of naive set theory that he described in 1902 in a letter to Gottlob Frege. The paradox is

   Let $W = \{X : X \notin X\}$. Is it the case that $W \in W$?

   In naive set theory, all entities are sets, and there is one basic binary predicate, namely membership $\in$. There are a number of ways of creating sets. Notably, a set can be empty, the complement of another set, or the union of two other sets. A set can also be defined intensionally, as containing all sets satisfying some logical formula. This method of definition is used for W above.

   Define an ML datatype to represent the sets of naive set theory, and define an appropriate member function. Create an ML value `w` to represent W, and a value `v` to represent the set of all sets that are members of themselves. Does the ML type system allow you to evaluate `member v v`? And `member w w`? If so, what happens? If not, why not?

   [Solution due to Bill Grundy.] The type definition for naive sets contains four constructor functions, each one corresponding to one of the ways of creating sets:

```
datatype set = Empty
             | Union of set * set
             | Complement of set
             | Intension of set -> bool;
```

   A new set can be created in any of four ways: from nothing (`Empty`), from one other set (`Complement`), from two sets (`Union`), or by specifying a function which maps a set to a boolean (`Intension`).

   An example set we can create is the universal set, which is most easily obtained using the `Complement` constructor:

```
val universe = Complement Empty;
- val universe = Complement Empty : set;
```

   Note that we can have several representations of the empty set:

```
val empty' = Union(Empty, Empty);
val empty'' = Complement universe;
```

   Both the empty set and the universal set can be generated using intensional definitions:

```
val empty''' = Intension (fn x => false);
val universe' = Intension (fn x => true);
```

Intensional definitions also allow us to define more complicated sets. For instance, here is the set which contains the empty set:

```
val metaempty = Intension (fn Empty => true
                            | _ => false);
```

Having created a number of sample sets, we now define the `member` function to test whether a given set is a member of another set.

```
fun member _ Empty = false
  | member x (Union(s1,s2)) = (member x s1) orelse (member x s2)
  | member x (Complement s1) = not (member x s1)
  | member x (Intension test) = test x;
```

This definition reflects the simple logic of naive set theory: (1) nothing is an element of the empty set; (2) an element which is a member of the union of two sets must be a member of at least one of those sets; (3) if an element is a member of a set then that element is not a member of the set's complement. The fourth clause in the function implements the idea of intensional definition by applying the given test predicate (i.e. boolean-valued function) to the element.

```
- member Empty Empty;
val it = false : bool
- member empty' empty'';
val it = false : bool
- member Empty empty''';
val it = false : bool
- member Empty universe;
val it = true : bool
- member Empty metaempty;
val it = true : bool
```

An interesting problem arises when we try the following:

```
- member empty' metaempty;
val it = false : bool
```

This result is incorrect. It illustrates that the `member` function does not equate the two representations of the empty set, `Empty` and `empty'`. This fault is certainly a drawback of the `member` function, but it is an unavoidable one. In order for the `member` function to correctly ascertain that `empty'` is in `metaempty`, the function would require an equality function which operates on sets. Such a function is clearly impossible, since it would require either an element-by-element comparison of possibly infinite sets or an equality function defined for pairs of functions.

Another problem with these sets as we have defined them is more fundamental. It is the paradox pointed out by Russell in 1902. We define two sets, one of which contains all and only the sets which contain themselves and the other which contains all and only the sets which do not contain themselves.

```
val v = Intension (fn x => member x x);
val w = Intension (fn x => not (member x x));
```

As defined, these sets cause no problem for ML. The problem arises when we ask whether `v` and `w` are members of themselves. The call `member v v` hangs indefinitely; the call `member w w` generates these messages:

```
- member w w;

[Increasing heap to 12291k]

[Major collection... 78% used (5223356/6627172), 1780 msec]

[Increasing heap to 19191k]

[Major collection...
[Increasing heap to 31423k]
 82% used (8927484/10876048), 3370 msec]

[Increasing heap to 31875k]

Interrupt
```

The increase in heap space is required because the `member` function is calling itself recursively *ad infinitum*. That is, `member w w` reduces to `not (member w w)` and then to `not (not (member w w))` and so on. The heap space used increases as each call is placed on the heap. (ML uses heap instead of stack space for function calls because of its use of higher-order functions.) In contrast, `member v v` reduces to itself, not to a more complex expression. An optimization in the ML implementation allows this variety of recursion (called tail-recursion) to be executed without using any additional heap space.

2. Explain informally what the class of "primitive recursive" functions is, and what the class of "general recursive" functions is. How is the distinction between these respected in Pascal, but not in C? Is the distinction respected in ML?

   Inspired by your answers to the questions above, show that ML is Turing-complete.

   [Solution due to George Xylomenos.] We will start by adapting some definitions from *Elements of the theory of computation* by Harry R. Lewis and Christos H. Papadimitriou (Prentice-Hall, 1981). Let N be the set of natural numbers.

   – The constant zero function ζ is the function from N^0 to N such that $\zeta() = 0$.

- Let $k \geq 1$ and let $1 \leq i \leq k$. Then the i-th k-place projection function π_i^k is the function from N^k to N such that

$$\pi_i^k(n_1, \ldots, n_k) = n_i \text{ for any } n_1, \ldots, n_k \in N.$$

- The successor function σ is the function from N to N such that

$$\sigma(n) = n + 1 \text{ for each } n \in N.$$

An *initial function* is any one of the functions defined above. To make more complex function, we extend the initial functions using the following constructs.

- Let $l > 0$ and $k \geq 0$, let g be an l-place function, and let $h_1, \ldots, h_l$ be k-place functions. Define f to be the k-place function such that for every $n_1, \ldots, n_k \in N$

$$f(n_1, \ldots, n_k) = g(h_1(n_1, \ldots, n_k), \ldots, h_l(n_1, \ldots, n_k)).$$

  Then f is said to be obtained from g and $h_1, \ldots, h_l$ by composition.
- Let $k \geq 0$, let g be a k-place function, and let h be a $(k+2)$-place function. Let f be the $(k+1)$-place function such that for every $n_1, \ldots, n_k \in N$,

$$f(n_1, \ldots, n_k, 0) = g(n_1, \ldots, n_k)$$

  and for every $n_1, \ldots, n_k \in N$ and $m \in N$

$$f(n_1, \ldots, n_k, m+1) = h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m)).$$

  Then f is said to be obtained from g and h by primitive recursion.

Now, a function is said to be *primitive recursive* if it is either an initial function or it can be generated from the initial functions by some finite sequence of compositions and primitive recursions. For general recursion, we must enlarge the class that is defined by the previous operators, using a new one.

- Let $k \geq 0$ and let g be a $(k+1)$-place function. Let g be the k-place function f such that for any $n_1, \ldots, n_k \in N$

$$f(n_1, \ldots, n_k) = \begin{cases} \text{the least } m \text{ such that } g(n_1, \ldots, n_k, m) = 0, \text{ if such an } m \text{ exists} \\ \text{undefined otherwise} \end{cases}$$

  We say that f is obtained from g by unbounded minimalization.

Relaxing a bit the original definition from Lewis and Papadimitriou, we can say that a function is *general recursive* if it can be obtained from the initial functions using composition, primitive recursion or unbounded minimalization.

The main difference between these two classes of recursive functions is the "unbounded" nature of minimalization. In general, we do not know whether the application of this operator will produce a result, since the function it is applied to may not map any of the elements in its domain to zero. It has been proved that the Turing machine is a model of computation that computes exactly the general recursive functions, so we can show that a language is Turing-complete by showing that it can compute all general recursive functions.

In Pascal, there are two varieties of loop: the `for` and the `while/repeat` loops. The first case gives bounded loops, where the number of iterations is set in the beginning, before the loop starts execution, and cannot change during the iterations. This corresponds exactly to the *bounded minimalization* operator, which is shown by Lewis and Papadimitriou to produce primitive recursive functions. It follows that a program that contains only `for` loops in Pascal (we disregard the `goto` construct since we can, and should, avoid it) will compute a primitive recursive function. To compute general recursive functions, we have to use either the `while` or the `repeat` construct, which evaluate general expressions that can (and should) be modified by the iterations, so there is the possibility of having infinite, non-terminating computations.

In C the `for` construct is basically equivalent to the `do` and `while` constructs, since it evaluates a control expression before every iteration, and the value of this expression is allowed to be changed by the actions inside the loop. In this sense, it is just a shorthand for the other control structures, so there is no clear distinction in C between the programs that compute primitive and general recursive functions with regard to the built-in language constructs.

In pure ML repetition is achieved through recursion rather than by loops. We can implement unbounded minimalization using a conditional expression and recursive function calls that carry over the index of the current computation, i.e. m in the definition above. The same syntax is used in ML (and all other major programming languages) for primitive and for general recursion. There are no syntactic constructs in ML that inherently keep the functions written using them within the class of primitive recursive functions. ML is thus similar to C in not respecting the distinction between primitive and general recursion.

To show that ML is Turing complete, we will show that there are ML functions that compute the initial functions defined above, and that given the definitions of any general recursive functions, we can combine them following the composition, primitive recursion and unbounded minimalization schemes.

For the constant function, the definition is simply

```
fun z()=0;
```

Note that ML regards this function as having the domain `unit`, since we cannot define N^0.

For the rest of this discussion, we will have to deal with functions of multiple arguments, so we will give *templates* instead of actual definitions. That means that we will write $n_1, ..., n_k$ instead of a specific number of integer arguments. For clarification on the number of parameters in each function, refer back to the definitions. We will assume also that the domain of the functions is the mathematical natural numbers, and not the the ML integers, which can be negative and are bounded. A more complete proof would involve defining a data type for the unbounded non-negative naturals.

The i-th projection function is simply

```
fun pi(n1,...,nk) = #i(n1,...,nk);
```

Note that i is fixed for any one of these functions; it is not an argument. Also, note that in this, and in some of the later templates, we will have to define the data type of some of the arguments to make ML avoid regarding the function as polymorphic. The last initial function is the successor function, which can be defined as:

```
fun s(n)=n+1;
```

Given `g` and `h1` through `hl`, we can define the function `f` using composition as follows:

```
fun f(n1,...,nk) = g(h(n1,...,nk),...,hl(n1,...,nk));
```

For primitive recursion, we need functions `g` and `h`, and a two part definition:

```
fun f(n1,...,nk,0) = g(n1,...,nk)
  | f(n1,...,nk,m) = h(n1,...,nk,m-1,f(n1,...,nk,m-1));
```

Note the use of m in the pattern instead of the $m + 1$ of the definition, to avoid pattern matching problems in ML. The last case is the unbounded minimalization construct. As we said, we will use a conditional construct and recursive calls instead of the more awkward `while` construct of ML. Given function `g`, we can construct its unbounded minimalization `f` as follows:

```
local
   fun f0(n1,...nk,m) = if g(n1,...,nk,m) = 0 then m
                        else f0(n1,...nk,m+1)
in
   fun f(n1,...,nk) = f0(n1,...,nk,0)
end;
```

The local function `f0` augments the arguments with the counter that shows how far we have progressed during the recursion and serves (if the function terminates) as its result.

We have now shown how to define in ML any general recursive function, either initial or derived, so we have completed the proof that ML is Turing-complete.