CS3110 Spring 2017 Lecture 7
Specifications using types continued

Robert Constable

1 Lecture Plan

1. Repeating schedule of remaining five problem sets and prelim.
2. Fixing OCaml specification and code for primality.
3. Solving the challenge specification from Lecture 6.
4. Specifying and solving basic list tasks, *map* and *reduce*.

2 Schedule of problem sets and in-class prelim

	Date for	Due Date
PS2	Out on Feb 23	March 2
Prelim	Tue. March 14, in class	
PS3	Out on Thur. March 2	March 16
PS4	Out on March 16	March 30
PS5	Out on April 10	April 24
PS6	Out on April 24	May 8 (day of last lecture)

3 Logical specifications compared to type theory specifications

You will notice that in the posted Lecture 6 notes, I changed the specification of the problem a bit from what I wrote on the board. That is because Mr. J. Miller noticed a small mistake in the specification. It is fixed and discussed in these notes. We changed a clause in the primeness test from $0 < x < y$ to $0 < x \leq y$.

Using the quantifiers and connectives of logic we define $prime(p)$ iff

$$p > 1 \ \& \ (\forall x, y : \mathbb{N}.(0 < x \leq y \ \& \ (x \times y) = p)) \Rightarrow x = 1 \ \& \ y = p).$$

What does this say in words? It says that p is a prime number if and only if (iff) its only positive integer divisors are 1 and p itself.

The key clauses are the three Boolean relations $b1 : (0 \ < \ fst(pr))$, $b2 : (fst(pr) \ \leq \ snd(pr))$, $b3 : (fst(pr) \times snd(pr)) = p$. From these inputs, we must conclude that $(fst(pr) = 1)$ and $(snd(pr) = p)$. If we can draw that conclusion, then p is prime. The first specification used the weaker clause $b2 : (fst(pr) \ < \ snd(pr))$. That specification would miss cases such as 3*3 = 9.

At some later point we will discuss writing the code that satisfies this specification. Before that, we will look at other simpler specifications.

We have already seen that some *simple polymorphic OCaml types resemble propositional formulas* from logic. This analogy is worth investigating further because it reveals a very strong connection between types and propositions that has been explored in computer science and logic for the past four decades, and it has been implemented in proof assistants such as Agda, Coq, and Nuprl [2, 1, 3]. It is one of the many deep connections between computer science, mathematics, and logic. It is also having an impact in philosophy, for instance in epistemology.

3.1 Comments on polymorphic type specifications

We have already seen that polymorphic types allow us to express simple programming tasks. We noticed that the specifications look close to logical expressions.

Consider the type

$$(L(\alpha \rightarrow \gamma)|(R(\beta \rightarrow \gamma)) \rightarrow (L\alpha|R\beta) \rightarrow \gamma).$$

Is there a program in this type? When seen as a logical expression, is it true?

First let us try to develop a program from the specification. We see that from the input $(L(\alpha \rightarrow \gamma)|(R(\beta \rightarrow \gamma))$ we need to find code for $(L\alpha|R\beta) \rightarrow \gamma)$. So we start with a function $fun\ df\ \rightarrow \{\quad\}$.

We can try to do a match on df and reduce the problem to filling in the hole $\{\quad\}$ in two separate cases. The first case is when we assume

$L(\alpha \to \gamma)$ the second is when we assume $R(\beta \to \gamma)$. Suppose we are in the first case, then we can also assume $dab : (L\alpha|R\beta)$. We can now proceed by cases on $(L\alpha|R\beta)$. If we are in the $L\alpha$ case, then we can use $(\alpha \to \gamma)$, but if we are not, **then we are stuck**. We see that we'll be stuck in the $R(\beta \to \gamma)$ case as well when we are trying to use $L\alpha$.

This situation leads us to wonder whether the proposition is even true. We can use a truth table analysis to figure this out. We can see by some careful analysis that the formula is not a tautology. We can make it false by using the truth values $T\alpha$ and $F\gamma$ or $T\beta$ and $F\gamma$. This tells us that *the specification is not programmable.*

This is a general fact: if a polymorphic Boolean specification is not a tautology, then it is not programmable.

However, it is not the case that all Boolean satisfiable specification are programmable. A simple example is $L\alpha|R(\alpha \to void)$ where *void* is the empty type. Another example of this phenomenon arises in trying to program *Pierce's Law* which using polymorphic OCaml types is this specification, $((\alpha \to \beta) \to \alpha) \to \alpha$. We will look deeper into this situation once we have developed more tools for thinking about it.

4 List processing and the map-reduce paradigm

OCaml has a built in type of lists created with the template [; ; ;]. On the other hand, we can define our own version of lists as an example of a *recursive type*. We show that approach first, taking material from the 2008 version of the course using the above url.

```
The Map-Reduce Paradigm

Map-reduce is a programming model that has its roots in functional programming.
In addition to often producing short, elegant code for problems involving lists
or collections, this model has proven very useful for large-scale highly parallel
data processing.  Here we will think of map and reduce as operating on lists for
concreteness, but they are appropriate to any collection (sets, etc.).

Map operates on a list of values in order to produce a new list of values,
by applying the same computation to each value.  Reduce operates on a list
of values to collapse or combine those values into a single value (or more
```

generally some number of values), again by applying the same computation to each value.

In the introduction of their 2008 paper, Dean and Ghemawat write "Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately." In the paper they discuss a number of applications that are simplified by this functional programming view of massive parallelism. To give a sense of the scale of the processing done by these programs, they note that over ten thousand programs using map-reduce have been implemented at Google since 2004, and that in September 2007 over 130,000 machine-months of processing time at Google were used by map-reduce, processing over 450PB (450,000 TB) of data.

For our purposes here in a programming course, it is illustrative to see what kinds of problems Google found useful to express in the map-reduce paradigm. Counting the number of occurrences of each word in a large collection of documents is a central computational issue for indexing large document collections. This can be expressed as mapping a function that returns the count of a given word in each document across a document collection. Then the result is reduced by summing all the counts together. So if we have a list of strings, the map returns a list of integers with the count for each string. The reduce then sums up those integers. Many other counting problems can be implemented similarly. For instance, counting the number of occurrences of some pattern in log files, such as the number of occurrences of a given user query or a particular url. Again there are many log files on different hosts, this can be viewed as a large collection of strings, with the same map and reduce operations as for document word counts.

Reversing the links of the web graph is another problem that can be viewed this way. The web is a set of out-links, from a given page to the pages that it links to. A map function can output target-source pairs for each source page, and a reduce function can collapse these into a list of source pages corresponding to each target page (i.e., links in to pages).

An inverted index is a map from a term to all the documents containing that term. It is an important part of a search engine, as the engine must be able to quickly map a term to relevant pages. In this case a map function returns pairs of terms and document identifiers for each document. The reduce collapses the result into the list of document ID's for a given term.

In the Google papers they report that re-implementing the production indexing system resulted in code that was simpler, smaller, easier to understand and modify, and resulted in a service that was easier to operate (ie failure diagnosis, recovery, etc.), yet the approach results in fast enough code to be used for a key part of the service.

Mapping and Folding (Reducing)

First lets look in more detail at the map operation. Map applies a specified function f to each element of a list to produce a resulting list. That is, each element of the result is obtained by applying f to the corresponding element of the input list. We will consider the curried form of map. In OCaml the built-in function List.map produces the following value for a list of three elements:

```
    map f [a; b; c] = [f a; f b; f c]
```

Recall we introduced the list_ type:

```
    type 'a list_ = Nil_ | Cons_ of ('a * 'a list_)
```

The map operation for this type can be written as:

```
    let rec map (f: 'a->'b) (x: 'a list_): 'b list_ =
      match x with
          Nil_ -> Nil_
        | Cons_(h,t) -> Cons_(f(h), map f t)
```

Note the type signature of map which is

```
    ('a -> 'b) -> 'a list_ -> 'b list_
```

The parameter f is a function from the element type of the input list 'a to the element type of the output list 'b.

Using map we can define a function to make a copy of a list_ (using an anonymous function),

```
    let copy l = map (fun x -> x) l
```

Similarly we can create a string list_ from an int list_:

```
    map string_of_int Cons_(1,Cons_(2,Cons_(3,Nil_)))
```

Now consider the reduce operation, which like map applies a function to every element of a list, but in doing so accumulates a result rather than just producing another list. Thus in comparison with map, the reduce operator takes an additional argument of an accumulator. As with map, we will consider the curried form of reduce.

There are two versions of reduce, based on the nesting of the applications of the function f in creating the resulting value. In OCaml there are built-in reduce functions that operate on lists are called List.fold_right and List.fold_left. These functions produce the following values:

```
    fold_right f [a; b; c] r = f a (f b (f c r))
    fold_left f r [a; b; c] = f (f (f r a) b) c
```

From the forms of the two results it can be seen why the functions are called fold_right which uses a right-parenthesization of the applications of f, and fold_left which uses a left-parenthesization of the applications of f. Note that the formal parameters of the two functions are in different orders, in fold_right the accumulator is to the right of the list and in fold_left the accumulator is to the left of the list.

Again using the list_ type we can define these two functions as follows:

```
    let rec fold_right (f:'a -> 'b -> 'b) (lst: 'a list_) (r:'b): 'b =
        match lst with
          Nil_ -> r
        | Cons_(hd,tl) -> f hd (fold_right f tl r)

    let rec fold_left (f: 'a -> 'b -> 'a) (r: 'a) (lst: 'b list_): 'a =
        match lst with
          Nil_ -> r
        | Cons_(hd,tl) -> fold_left f (f r hd) tl
```

Note the type signature of fold_right which is

```
    ('a -> 'b -> 'b) -> 'a list_ -> 'b -> 'b
```

The parameter f is a function from the element type of the input list 'a and the type of the accumulator 'b to the type of the accumulator. The type signature is analogous for fold_left,except the order of the parameters to both f and to fold_left itself are reversed compared with fold_right.

Given these definitions, operations such as summing all of the elements of a list of integers can naturally be defined using either fold_right or fold_left.

```
    fold_right (fun x y -> x+y) il 0
    fold_left (fun x y -> x+y) 0 il
```

The power of fold

Folding is a very powerful operation. We can write many other list functions in terms of fold. In fact map, while it initially sounded quite different from fold can naturally be defined using fold_right, by accumulating a result that is a list. Continuing with our List_ type,

```
    let mapp f l = (fold_right (fun x y -> Cons_((f x),y)) l Nil_)
```

The accumulator function simply applies f to each element and builds up the resulting list, starting from the empty list.

The entire map-reduce paradigm can thus actually be implemented using fold_left and fold_right. However, it is often conceptually useful to think of map as producing a list and of reduce as producing a value.

What about using fold_left instead to define map? In this case we get a function that not only does a map but also produces an output that is in reverse order of the input list. Note that fold_left takes its arguments in a different order than fold_right (the order of the list and accumulator are swapped), it also requires a function f that takes its arguments in the opposite order of the f used in fold_right.

```
    let maprev f l = fold_left (fun x y -> Cons_((f y),x)) Nil_ l
```

```
This resulting function can also be quite useful, particularly as it
is tail recursive.

Another useful variation on mapping is filtering, which selects a
subset of a list according to some Boolean criterion,
```

```
    let filter f l = (fold_right (fun x y -> if (f x) then
    Cons_(x,y) else y) l Nil_)
```

```
The function f takes just one argument, the predicate for
determining membership in the resulting list. Now we can easily
filter a list of integers for the even ones:
```

```
    filter (fun x -> (x / 2)*2 = x) Cons_(1,Cons_(2,Cons_(3,Nil_)))
```

```
Note that if we define a function that filters for even elements of a list:
```

```
    let evens l = filter (fun x -> (x / 2)*2 = x) l;;
```

```
then type of the parameter and result are restricted to be int list_
rather than the more general 'a list of the underlying filter, because
the anonymous function takes an integer parameter and returns an integer
value.

Determining the length of a list is another operation that can
easily be defined in terms of folding.
```

```
    let length l = fold_left (fun x _ -> 1 + x) 0 l
```

4.1 More on types compared to sets

In Lecture 5 we started to compare types and sets. This will be a minor theme that we mention from time to time. Even for the very simple example of natural numbers and integers, we can see both the similarities and differences. In this subsection, we notice some superficial differences. The most obvious difference might be in the treatment of functions. To define a function from $\mathbb{N}$ to $\mathbb{N}$ in set theory, we only need to create a set of ordered pairs such that any two pairs with the same first element have the same second element. In type theory we need to produce a program to compute the function, and type $\mathbb{N} \to \mathbb{N}$ consists not of a collection of ordered pairs but of a collection of *computable functions* given as OCaml programs.

We consider a simple comparison by looking at the natural numbers. In set theory, the number 0 is defined as the empty set, say ϕ. The number 1 is the set containing only the empty set, $\{\phi\}$, 2 is the set $\{\phi, \{\phi\}\}$. In simpler notation, 1 is $\{0\}$, and 2 is $\{0, 1\}$, and 3 is $\{0, 1, 2\}$. This is all very elegant, but not good for calculating. To define functions we need the idea of an ordered pair in set theory. The ordered pair of two sets x and y, say $< x, y >$ is defined to be the set $\{\{x\}, \{x, y\}\}$.

To summarize as in Lecture 4, types are a collection of *canonical values* from the computation system. There is a notion of equality defined on them. As canonical values, the expressions stand on their own. The meaning of the expressions arises from relating the canonical and non-canonical values. For example, consider the list `[1;2;3]`. The relationship between the constructors and the destructors starts to reveal their "meaning." When we say `hd [1;2;3] = 1` then *we experience the meaning* of the *hd* operation. We start to "understand" what a list is by applying operations to build them and others to take them apart.

We are starting to see how we capture this meaning in the language of the OCaml type theory. We use the *operational semantics* from the second lecture. From this definition, we already see that before we can understand an OCaml type, we need to know the *computation rules* relevant to it. That is why we started by discussing evaluation and canonical values. Without those ideas, we cannot understand types in computer science. In contrast, to understand sets in mathematics, we never mention computation. We start with much more abstract concepts. It is an interesting challenge for type theory to achieve more abstraction while preserving computational meaning.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[2] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *LNCS 5674, Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[3] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, NJ, 1986.