CS3110 Fall 2013 Lecture 4: Lists, unions, map, fold, specification issues (9/10)

Robert Constable

1 Lecture Plan

1. OCaml lists and operations on them
   - type name
   - values (canonical expressions)
     - constructors (::)
   - operators (non canonical expressions)
     - destructors (match)
   - examples
   - specifications
2. Disjoint union type and variant types
3. The map operation on lists, folding
   - `List.map`
   - `List.fold_right`
4. Specifications appropriate to list implementation
5. Induction on lists as basis for proofs
6. Connections to Problem Set 2

2 Review and Overview

In Lecture 3 we saw that induction on natural numbers is a kind of recursive procedure that is guaranteed to terminate. Termination is a very

significant property of recursive procedures that type checkers cannot decide. We also saw that the core of all the proofs by induction can actually be executed and used as a program. Moreover, if the logical evidence is simple enough, e.g. a Boolean value, then we can execute the entire inductive proof.

In this lecture we will go more deeply into the study of programming with lists in OCaml. We briefly introduced them on the problem set, so you know the syntax. This lecture will follow the approach of Prof. Kozen's Lecture 3 from 2011sp. You have access to these notes from the list of CS courses:
http://www.cs.cornell.edu/courses/CS3110/2011sp/lectures/lec03-scope/scope.htm

The key concepts from his Lecture 3 are: lists, constructing lists, and pattern matching. We give the computation rules for pattern matching.

We will also use Prof. Kozen's treatment of disjoint unions and variant types in his Lecture 4 from 2011sp,
http://www.cs.cornell.edu/courses/CS3110/2011sp/lectures/lec04-types/types.htm,
and his treatment of map and fold from Lecture 5,
http://www.cs.cornell.edu/courses/CS3110/2011sp/lectures/lec05-map-reduce/map-reduce.htm

The key topics from his Lecture 4 are: variant types, and variant type syntax. We show here how to use those concepts to overcome the limitation of OCaml having only homogeneous lists.

From Kozen's Lecture 5 we take the definition of mapping and folding. This section is the most key part of his notes for solving the exercises on Problem Set 2 involving folding.

There is one core new issue about the task of reasoning about programs that we face when we program with lists; it is the issue of **how do we specify programming tasks involving lists?** In examples that involve numbers, natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), real numbers ($\mathbb{R}$), etc., the programming problem can often be expressed in terms of a well known mathematical concept such as the summation operator, $\Sigma_{i=1}^{n} i$ or the continuous analogue, the integral.

In the case of lists and trees and other recursive data structures, there are

no well established mathematical notations that make it clear how to state programming problems. That is why we resort to logic, the $\forall, \exists$ operators and other logical symbols. We will see that **OCaml's type theory and mathematical type theory provide good concepts for specification as well as computation**.

3 Lists

In this summary of the lecture points, we refer to the concepts and notations from Kozen's lecture. The type of an OCaml list is **t list** where **t** is a type. All elements of the list must be of this type, that is the list is **homogeneous**. For example `[0;1;2;3;4]` is a length 5 list of integers. The following are examples of list types:

unit list	int list	bool list	α list
$(\alpha \to \alpha)$ list	$(\alpha * \alpha)$ list		
(int list) list	((int list) list) list		

Elements (values, canonical expressions) of type list

int list `[1;2;3;4]`

also `[]` the **nil list**

Is `[1*1;2;2+1;2*2]` also a list?

yes, but not a canonical value as the first example is

How can we construct new lists?

There is a polymorphic operator, called cons in this offering of the course which takes an element of type t and a list of type t *list* and builds another list.

Typing Rule $x : t,\ l : t\ list \vdash x :: l \in t\ list$

e.g. `3::[]` $\downarrow$ `[3]`

`2::[3]` $\downarrow$ `[2;3]`

`3::[4;5]` $\downarrow$ `[3;4;5]`

How do we take lists apart and access their elements?
There is a **destructor** corresponding to the constructor.

$$\textbf{match } lst \textbf{ with } h :: t \to exp_1(h,t) \mid \texttt{[]} \to exp_2$$

The computation rule or reduction rule is

> Compute lst to a value.
> It will either be $\texttt{[}v_1 \cdots \texttt{]}$ or `[]`. In the case $\texttt{[}v_1 \cdots \texttt{]}$ the reduction evaluates the expression $exp_1(v_1, \cdots)$. In the case `[]` it evaluates exp_2.

We need to look more closely at the $\texttt{[}v_1 \cdots \texttt{]}$ case. What is $\cdots$?

We can reveal this by computing the match on t. What is

> **match** `[1]` **with**
> $\mid h :: t \to$ (**match** t **with** $h_1 :: t_1 \to$ `[0]` $\mid$ `[]` $\to$ `[2]`)
> $\mid$ `[]` $\to$ `[]`

The result is `[2]`. This computation shows that `[1]` is actually `1::[]`.
We will discuss list structure further on Thursday.

In order to talk about lists and specify the operations we want to perform on them, we need some basic operations.

- find the length of a list
- find the i^{th} element of a nonempty ("non nil") list for $1 \leq i \leq length\ l$

Length of a list

All lists have a fixed length and are immutable. We can easily compute the length of any list. The code is

```
# let rec len (lst : 'a list) =
    match lst with
    | h::t -> 1 + len t
    | [] -> 0 ;;
val len : 'a list -> int = <fun>

len [1;2;3;4;5]
- : int = 5

len []
- : int = 0
```

Note that `len` is polymorphic, so `len [‘a’;‘b’;‘c’;‘d’]` $\downarrow 4$.

m^{th} element of a list

One of the key functions on lists finds its m^{th} element starting from 1. For example,

```
# mth [‘a’;‘b’;‘c’;‘d’;‘e’;‘f’;‘g’] 4 ;;
- : char = ’d’
```

Here is how we compute the m^{th} element of a list having the property

`mth [1;2;3] 1 = 1`
`mth [1;2;3] 3 = 3`
`mth [1;2;3] 4` raise an exception

```
# let rec mth lst n =
    match lst with
    | h::t -> if n = 1 then h else mth t (n-1)
    | [] -> raise Not_found ;;
val mth : ’a list -> int -> ’a = <fun>
```

The code is polymorphic, that is, it works for `lst` of type `’a list`.

Note `Not_found` has type `exception`, but we do not return that as a value. We will see another way to deal with the empty list shortly.

Notice that the `mth` operator makes lists look somewhat like arrays, but arrays have a fixed length and lists do not. We see that any specific list `lst = [a1;a2;...;an]` is equal to `[mth lst 1;...;mth lst (len lst)]`.

4 Disjoint union and variant types

OCaml lists are **homogeneous**, e.g. all elements are of the same type. This might seem to be a serious limitation. What if we want lists that represent “banking numbers” such as $27,572,641. We’d like the numbers, commas, and dollar sign as elements – we could skip the $.

What we want is a type like the union type on sets, e.g.
$\{0, 1, 2, 3, \ldots, 9\} \cup \{‘g’\}$.

OCaml has **disjoint unions**, also called variant types. Please read Jason Hickey's account of union types in Chapter 6 of his book. You should also look at his Chapter 9 on Exceptions, 9.1 and 9.2. The simplest case of a union is the binary disjoint union of two types.

Example: Suppose we have two types *type_a*, *type_b*. Their union is
type a_or_b = *A of type_a* | *B of type_b*.

The **canonical values** of this type are
A a where $a \in$ *type_a*
B b where $b \in$ *type_b*

For example, *type int_char* = *Int of int* | *Char of char*.

With disjoint unions, we can have a list whose elements are essentially integers or characters. This would let us write a list of numbers grouped into segments of three by commas to represent standard decimal numbers such as 34,459,720 with the list `[Int 3;Int 4;Char ,;Int 4;Int 5; Int 9; Char ,;Int 7;Int 2;Int 0]`.

To operate on elements of disjoint unions we use the matching operation, in the style used for lists

$$\textbf{match} \quad \textbf{with}\ p_1 \to e_1 \mid \cdots \mid p_n \to e_n$$

For *a_or_b* we could have the match

$$\textbf{match}\ e\ \textbf{with}\ A\ a \to exp_1(a) \mid B\ b \to exp_2(b)$$

Imagine how you would implement the addition and multiplication of "banking numbers"

```
[3,275,826,115] * [478,296]
```

This is quite simple and is a warm up for our PS2 example of operating on "bignums." As Problem Set 2 will show you, we can write the standard arithmetic operations on **unbounded natural numbers** by representing the numbers as lists of natural numbers (non-negative integers) grouped

with commas. This is simply a matter of writing down the usual operations as recursive procedures on these lists, taking care of carrys. With the right education, elementary school students could be taught to do this in a version of an OCaml-like language designed for such students.

Note, we can also use disjoint unions in place of exceptions: we can use a union type to define outputs that report an exception without using the exception mechanism.

In the list example, `mth [1;2;3] 4` could return a value in int_char. As another example, we can report the error of attempting to divide by 0 in an expression and define a function $(float \star float) \to float\_char$, defined by analogy with int_char, that reports the character ? as an exception.

5 Mapping functions over lists

See Kozen Lecture 5 2011 spring for an introduction to map and fold and a discussion of the "Map-Reduce" paradigm.

The OCaml List library has a function `List.map`

Its type is $(\alpha \to \beta) \to \alpha\ list \to \beta\ list$.

It does just this: `List.map` f `[`a;b;c`] = [`$f\,a$;$f\,b$;$f\,c$`]`

Suppose that f is the function on int which finds the least prime factor of the absolute value, call it lpf.

`List.map` lpf `[18;169;343;605;289]` $\downarrow$ `[2;13;7;5;17]`

Suppose we wanted the product of these numbers. We could just multiply the resulting elements of the list, but OCaml provides these functions that are called folding.

`List.fold_right` f `[a;b;c]` $r = f\ a\ (f\ b\ (f\ c\ r))$