Even More on Data Types and ML

CS152

Chris Pollett

Nov. 17, 2008.

Outline

- Finish up Type Constructors
- As we talk about the above we'll continue to introduce ML.

More Type Constructors

- Last day, we talked about simple types and how to build new types from simple types using: Cartesian product, Records, Union, and Subset.
- Functions provide another way to make types.
- Given two types U and V, the set of all functions from U to V gives rise to a type.
- In ML you could create such a type with a line like:

  ```
  type int_fun = int -> int;
  ```
- We could use this type with a line like:

  ```
  val f:int_fun = fn x => 2*x;
  ```
- In class we talked about currying (also in book).

Arrays (Vectors)

- If U in the type U-> V is an initial segment of the integers: say 0, 1, 2 .. up to some m; then we could view a function arr:U->V as an array where arr(i) is the ith element of the array.
- Languages often differ on whether the end point of the array can be dynamically set. C,C++ versus say Java.
- Some languages like Ada support the ability to set the start and end point of array indexes, say from -15 to 15.
- Other languages like Perl, PHP, Javascript, etc support associative arrays when we can have arbitrary key value pairs stored in arrays.
- In SML, we saw that constant sized arrays could be faked just using Cartesian product. SML also supports a vector type (and an array type):

```
val b = #["first-element", "second"];
```

- Each element of a vector has the same type. In the case above the type for the complete object is: string vector
- The ith element of a Vector.sub(b, i);
- The length of the vector can be found using: Vector.length(b);
- To access functions without Vector prefix use open Vector;

Lists in ML

- Related to arrays, ML also has a built-in facility for lists:

```
[1, 2, 3];  (* same as 1::2::3::nil *)
nil; (* empty list *)
[1,2]@[3,4]; (*concatenates to make [1,2,3,4] *)
hd([1,2,3]); (* returns head of list. I.e., 1 *)
tl([1,2,3]); (*returns tail of list. I.e., [2,3] *)
```

- As with vectors and arrays, each element in the list needs to be of the same type.
- We could create a new list type with a line like:

```
type int_list_type = int list;
```

Lists and Functions

- You can use patterns with lists when you are writing functions to make very succinct code:

```
fun reverse(nil) = nil
| reverse(x::xs) = reverse(xs)@[x];
```

- If we didn't use patterns we might have to type something like:

```
fun reverse(L) = if L = nil then nil
                 else reverse(tl(L)) @[hd(L)];
```

- Notice the type of this function is fn : 'a list -> 'a list.
- Here 'a denotes an arbitrary type. So this function could be applied to an int list, a string list, etc. This is an example of **polymorphism**.
- Converted above to a tail recursive function in class.

Pointers and References

- One type which does not correspond to a set operation is the pointer or reference type.
- This types corresponds to the set of all addresses that refer to a specified type.
- In C we could declare such a type using a syntax like: typedef int* IntPtr;
- To create a reference in ML we can do things like: val x = ref v;
- To create an actual reference type we could do:
  - datatype ref_int = ref of int;
  - We can modify a ref the value of a variable using :=
  - We can get the value of a ref variable with !

Recursive Datatypes.

- ML allows one to build up datatypes recursively:

```
 datatype 'label btree =
    Empty |
    Node of 'label * 'label btree * 'label btree;
```

- One can then define functions on these recursive types.