Programming Languages and Techniques (CIS120)

Lecture 8

Jan 30, 2012

Abstraction

Announcements

- Homework 2 due tonight at midnight
- Homework 3 will be available tomorrow morning
  - Due next Monday, Feb 6 at 11:59:59PM
  - Practice with BSTs, generic functions, abstract types
- Weirich office hours today 3:30-5PM in Levine 510

Generic Functions and Data

Structurally Identical Functions

- Observe: many functions on lists, trees, and other datatypes don't depend on the contents, only on the structure.
- Compare: length for "`int list`" vs. "`string list`"

```
let rec length1 (l:int list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + (length1 tl)
  end
```

```
let rec length2 (l:string list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + (length2 tl)
  end
```

The functions are *identical*, except for the type annotation for `l`.

Notation for Generic Types

- OCaml provides syntax for *generic function* types

```
let rec length (l:'a list) : int =
  begin match l with
  | [] -> 0
  | _::tl -> 1 + (length tl)
  end
```

- Notation: `'a` is a *type variable*; the function `length` can be used on a `t list` for *any* type `t`.
- Examples:
  - `length [1;2;3]` use length on an int list
  - `length [“a”;”b”;”c”]` use length on a string list

Generic List Append

Note that the two input lists must have the *same* type of elements.

The return type can also be generic – in this case the result is of the same type as the inputs.

```
let rec append (l1:'a list) (l2:'a list) : 'a list =
  begin match l1 with
  | [] -> l2
  | h::tl -> h::(append tl l2)
  end
```

Pattern matching works over generic types.
In the body of the branch:
`h` has type `'a`
`tl` has type `'a list`

Generic Zip

Functions can operate over *multiple* generic types.

```
let rec zip (l1:'a list) (l2:'b list) : ('a*'b) list =
  begin match (l1,l2) with
  | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
  | _ -> []
  end
```

- Distinct type variables can be instantiated differently:

  `zip [1;2;3] [“a”;”b”;”c”]`

- Here, `‘a` instantiated to `int`, `‘b` to `string`
- Result is the `(int * string) list`:

  `[(1,”a”);(2,”b”);(3,”c”)]`

User-defined Generic Datatypes

- Recall our integer tree type:

```
type tree =
| Empty
| Node of tree * int * tree
```

- We can define a generic version by adding a type parameter, like this:

```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree
```

Parameter ‘a used here

Note that the recursive uses also mention ‘a

Abstract Collections

Design Process

1. Understand the problem
2. Formalize the interface
3. Write test cases
4. Implement the required behavior

- How to formalize the interface?
  - Determine data representation + relevant operations
  - So far, we have used built-in structures for representation (int, list, tree)
  - What if the built-in structures don't fit exactly?

A design problem

As a high-school student, Stephanie had the job of reading books and finding which words, out of a list of the 1000-most common SAT vocabulary words, appeared in a particular book. She enjoyed being paid to read, but she would have enjoyed being paid to program more. How could she have automated this task?

1. What are the important concepts or *abstractions* for this problem?
   - The collection of words that appear in a book
   - The collection of 1000-most common SAT words
   - The process of determining whether a particular word from the first collection is contained in the second collection

A set is an abstraction

- Use a *set* for this collection
  - In math, we typically write sets like this: {1,2,3} {true,false}
    with operations like: $S \cup T$ or $S \cap T$ for union and intersection
    we write $x \in S$ to mean that "x is a member of the set S"
- A set is a lot like a list, except:
  - Order doesn't matter
  - Duplicates don't matter
  - *It isn't built into OCaml*
- Sets show up frequently in applications
  - Examples: set of students in a class, set of coordinates in a graph, set of answers to a survey, set of data samples from an experiment, …

2. Formalize the interface: myset.ml file

```
type 'a set = …

(* Need a way to create sets… *)
let empty : 'a set = …
let add (x:'a) (s:'a set) : 'a set = …
let union (s1:'a set)(s2:'a set) : 'a set = …
let remove (x:'a) (s:'a set) : 'a set = …
let list_to_set (l:'a list) : 'a set = …

(* …and a way to work with them *)
let is_empty (x:'a set) : bool = …
let member (x:'a) (s:'a set) : bool = …
let equal (s1:‘a set) (s2:‘a set) : bool = …
let elements (s:'a set) : 'a list = …
```

2a. Look at the interface: myset.mli file

```
type 'a set = …
val empty   : 'a set
val add     : 'a -> 'a set -> 'a set
val union   : 'a set -> 'a set -> 'a set
val remove : 'a -> 'a set -> 'a set
val list_to_set : 'a list -> 'a set
val is_empty : 'a set -> bool
val member    : 'a -> 'a set -> bool
val equal     : 'a set -> 'a set -> bool
val elements : 'a set -> 'a list
```

Keyword 'val' names values that must be defined and their types.

- OCaml puts *interfaces* (as above) in a .mli file
- The corresponding *implementation* goes in the .ml file

Function Types

- In OCaml, the type of functions from input `t` to output `u` is written:

```
t -> u
```

- Functions with multiple arguments use multiple arrows
- Here are some examples we have already seen:

```
size : tree -> int
hamming_distance : helix -> helix -> int
acids_of_helix   : helix -> acids list
length : ‘a list -> int
zip    : ‘a list -> ‘b list -> (‘a*’b) list
lookup : tree -> int -> bool
add    : ‘a -> ‘a set -> ‘a set
```

3: Write tests (in another file)

```
let s1 = Myset.add 3 Myset.empty
let s2 = Myset.add 4 Myset.empty
let s3 = Myset.union s1 s2

let test () : bool = (Myset.member 3 s1) = true
;; run_test "Myset.member 3 s1" test
let test () : bool = (Myset.member 4 s3) = true
;; run_test "Myset.member 4 s3" test
```

- To use the values defined in the set module use the "dot" syntax:

  `Myset.`*<member>*

- Alternatively, use "`;; open Myset`" at the top of a file to bring all of the names defined in the interface into scope.
- Note: Module names are always capitalized in OCaml

4. Implement the behavior

- There are many ways to implement sets.
  - lists, trees, arrays, etc.
- **How do we choose which implementation?**

- Many such implementations are of the flavor "a set is a … with some invariants"
  - A set is a *list* with no repeated elements.
  - A set is a *tree* with no repeated elements
  - A set is a *binary* search tree
  - A set is an *array of bits*, where 0 = absent, 1 = present
- **How do we preserve the invariants?**

Abstract types

BIG IDEA: Hide the *concrete representation* of a type behind an *abstract interface.*

- Example:
  - concrete 'set' representation – the *implementation* – is a list or a tree
  - abstract interface defines the operations in terms of a 'set' type

- The interface restricts how other parts of the program can interact with the data.
- Benefits:
  - **Safety**: The other parts of the program can't break any invariants that are being maintained behind the interface.
  - **Modularity**: It is possible to change the implementation of the abstract datatype without changing the rest of the program.

Defining Abstract Types

- Different programming languages* have different ways of letting you define abstract types.

- At a minimum, this means providing:
  - A way to specify (write down) an interface
  - A means of hiding implementation details (*encapsulation*)

- In OCaml:
  - Interfaces are specified using a *signature* or ML *interface file* (.mli)
  - Encapsulation is achieved because the interface can *omit* definitions.
  - Clients can't mention values not named in the interface.

*In Java, interfaces can also be written down explicitly and encapsulation is achieved by public/private modifiers on object fields. (We'll cover this in detail later.)

Example module interface: .mli file

```
type 'a set

val empty : 'a set
val add       : 'a -> 'a set -> 'a set
val union     : 'a set -> 'a set -> 'a set
val remove : 'a -> 'a set -> 'a set
val list_to_set : 'a list -> 'a set
val is_empty : 'a set -> bool
val member     : 'a -> 'a set -> bool
val equal      : 'a set -> 'a set -> bool
val elements : 'a set -> 'a list
```

Type declaration has no definition – its representation is *abstract*.

- Create a .mli file that omits information *on purpose*
  - The definition of the set type is hidden
  - Auxiliary functions used in the implementation are hidden

Naming the interface (a signature)

```
module type Set = sig
  type 'a set

  val empty : 'a set
  val is_empty : 'a set -> bool
  val member : 'a -> 'a set -> bool
  val add    : 'a -> 'a set -> 'a set
  val union  : 'a set -> 'a set -> 'a set
  val remove : 'a -> 'a set -> 'a set
  val list_to_set : 'a list -> 'a set
  val equal  : ‘a set -> ‘a set -> bool
  val elements : 'a set -> 'a list
end
```

Name the interface so that it can be reused by multiple implementations.

Module Implementation: trees

```
module BSet : Set =
struct
  type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree

  type 'a set = 'a tree (* definition hidden by .mli *)

  let empty : 'a set = Empty
  let is_empty (s:'a set) : bool = …
  …
end
```

Constrains the module to use the named interface.

- The implementation has to include *all* of the interface values
  - It can contain *more* functions and type definitions (e.g. auxiliary functions) but those cannot be used outside the module
  - The types of the provided implementations must match the interface

Module Implementation: lists

```
module LSet : Set =
struct

  type 'a set = 'a list

  let empty : 'a set = []
  let is_empty (s:'a set) : bool = …
  …
end
```

Constrains the module to use the given interface.

- To use the values defined in this module later on in the file use the "dot" syntax: `LSet.`*<member>*
- In another file, use "`;; open Hwset.LSet`" at the top of the file to bring all of the names defined in the interface into scope.