Principles of Programming Languages VII

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.dgp.toronto.edu/~wael/324.html

Acknowledgement: Some of the material in these lectures are based on material by Diane Horton
Today

- ML: recursive types, signatures & structures

- Introduction to Logic Programming
ML Recursive Types
ML: recursive data types

- Syntax

  ```ml
  datatype <type-name> =
  <type-constructor> of <type-name * type-expression>
  
datatype <type-name> =
  <type-constructor_1> of <type-name * type-expression_1>
  | <type-constructor_2> of <type-expression_2 * type-name>
  .......... 
  | <type-constructor_n> of <type-expression_n>
  ```
**ML: recursive data types**

- **Syntax**
  
  ```
  datatype <type-name> =
    <type-constructor_1> of <type-name * type-expression_1>
  | <type-constructor_2> of <type-expression_2 * type-name>
  .........
  | <type-constructor_n> of <type-expression_n>
  ```

- **Example:**
  
  - Tree representation
    ```
    - datatype tree =  leaf    of int
    | node of tree * tree;
    - node(leaf(2),leaf(5));
    > val it = node (leaf 2,leaf 5) : tree
    - node(leaf(3),node(leaf(2),leaf(5)));
    > val it = node (leaf 3,node (leaf 3,leaf 5)) : tree
    ```
ML: recursive data types

- Define a function to sum the leafs of the tree
  - `datatype tree = leaf of int`  
  `| node of tree * tree;`

  - `sum (node(leaf 3, node(leaf 2, leaf 5))));`
  > val it = 10 : int

  - `fun sum (leaf N) = N`
  `| sum (node(T1, T2)) = sum(T1) + sum(T2);`

```
sum(leaf 3) + sum(node (leaf 2, leaf 5))
3 + sum(leaf 2) + sum (leaf 5)
3 + 2 + sum (leaf 5)
3 + 2 + 5
```
Example:

- Define a binary tree whose leaves are integer functions & whose internal nodes are labeled by integers.
  
  ```ml
  fun square(X) = X * X;
  fun cube (X) = X * X * X;
  fun inc (X) = X + 1;
  node (5,leaf(inc),node(7,leaf(square),leaf(cube)));
  ```

  > val it = node (5,leaf fn,node (7,leaf fn,leaf fn)) : tree

  ```ml
  datatype tree = leaf of int -> int
  | node of int * tree * tree;
  ```

- Define a function that gathers all the functions at the leaves in a list
  
  ```ml
  gather (node (5,leaf(inc),node(7,leaf(square),leaf(cube))));
  ```

  > val it = [fn,fn,fn] : (int -> int) list

  ```ml
  fun gather (leaf F) = [F]
  | gather (node(N,T1,T2)) = (gather(T1) @ gather(T2));
  ```
Example:
- Define a function that returns the sum of applying all leaves on a number
  - `sum`
  
  (node(5, leaf(inc), node(7, leaf(square), leaf(cube))), 2);
  
  > val it = 15 : int

- `fun` `sum (leaf(F), N) = F(N)`
  
  `| sum (node(_, T1, T2), N) = sum(T1, N) + sum(T2, N);`

- Define a function that counts the number of leafs
  
  - `count (node (leaf(3.1), node(leaf 4.7, leaf 2.8)));`

  > val it = 3 : int

  - `fun` `count (leaf (X)) = 1`
    
    `| count (node (T1, T2)) = count(T1) + count(T2);`

  > val count = fn : 'a tree -> int
ML Structures & Signatures
**ML: structures**

- **Syntax**

  ```ml
  structure <structure-name> =
  struct
    (* exceptions, definitions, functions… *)
  end
  ```

- **Example:**

  ```ml
  structure Mapping =
  struct
    fun insert(key, value, []) = [(key, value)]
    | insert(key, value, (key1, value1)::rest) =
      if key = key1 then
        (key, value)::rest
      else
        (key1, value1)::insert(key, value, rest);

    fun lookup(key, (key1, value1)::rest) =
      if key = key1 then
        value1
      else
        lookup(key, rest);
  end:
  ```
ML: structures – cont’d

• Structure access:
  – Using long identifier
    • E.g. - Mapping.insert(538,"languages",[]);
            > val it = [(538,"languages")]: (int * string) list

            - Mapping.lookup(538,[(538,"languages"),(540,"courses")]);
            > val it = "languages" : string

            - Mapping.lookup(538,[(600,"teachers"),(540,"courses")]);
            > ????

  – Using open function
    • E.g. - open Mapping;
            - lookup(538,[(538,"languages"),(540,"courses")]);
            > val it = "languages" : string
**ML: structures – cont’d**

- **Properties**
  - It is legal to define one structure within another
  - If a structure has been defined within another structure, then its components can be accessed by an extension of the long identifier principle (x.y.z…)
  - A structure may be opened within another to achieve greater modularity. However, this may lead to name redefinition problems (similar to those discussed in last slide)
  - There is no equality defined over structures.
**ML: signatures**

- **Syntax**

  ```ml
  signature <signature-name> =
  sig
    (* definitions *)
  end;
  ```

- **Example:**

  ```ml
  - signature OBJ_sig =
    sig
      type OBJECT
      val grow : OBJECT -> OBJECT
      val shrink: OBJECT -> OBJECT
    end;
  ```
ML: signatures

• **Signatures & Structures:**
  - **signature** OBJ_sig =
    ```ml
    sig
      type OBJECT
      val grow : OBJECT -> OBJECT
      val shrink : OBJECT -> OBJECT
    end;
    ```
  - **structure** INT_struct : OBJ_sig =
    ```ml
    struct
      type OBJECT = int
      fun grow n = n + 1
      fun shrink n = n - 1
    end;
    ```

• **Benefits of using signature:**
  – Separation of specification from implementation decisions
  – Ability to provide programmers with different views of source code

• *If a structure implements a signature, then this structure is said to be constrained by this signature.*
ML: signatures

- Rules of signatures
  - Rule 1: name matching
    - Every type name and val name declared within a signature must have a corresponding definition in the constrained structure, else an error arises
  - Rule 2: type matching
    - Any definition that appears within a constrained structure and that has a corresponding declaration in the signature must match the type declared in the latter.
  - Rule 3: privacy
    - Any definition within a constrained structure that is not matched within its signature is private.
      - Such definition cannot be referenced by long identifier nor is it made available if the structure is opened
    - E.g.
      ```
      signature FOO =
      sig
        val talkToMe : unit -> int
      end;
      structure Foo2 : FOO =
      struct
        val bar = 42
        fun talkToMe () = bar
        fun hidden() = (* more code *)
      end;
      ```
ML: signatures

• Properties
  – There is no equality defined for signatures.
  – They are top-level objects, and cannot be defined within another object; furthermore (unlike structures) they cannot be nested.
  – The keyword include can be used to save writing long signatures by incorporating the contents of existing signatures within a new definition:
    • E.g.  
      
      signature NUM_sig =
      
      sig
        include OBJ_sig
        val Int_to_OBJ: int -> OBJECT
        val Real_to_OBJ: real -> OBJECT
      
      end
Introduction to Logic Programming
Logic Programming (LP)

• **Evolution:**
  Problem → Algorithm → Assembly Code → Machine Code
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>---Assembler---------</td>
</tr>
<tr>
<td>----Imperative/functional Compiler/Interpreter--</td>
</tr>
<tr>
<td>---------------------</td>
</tr>
</tbody>
</table>

  Why LP?
  – We can understand the meaning without knowing the “state” of the program
  – A lot easier to say *what*, but not *how*.
  – Direct manipulation of symbolic structures gives us it’s power.

• **Popular LP languages:** Prolog, SQL, Datalog

• **E.g.:**
  – Find X and Y such that 3 X + 2 Y = 1 and X – Y = 4
  – Retrieve the telephone number of the person whose name is Tom Smith
  – The value of X equals the value of Y + 3
LP: introduction – cont’d

• LP Characteristics:
  – Not based on state modifications
  – Not procedural in nature
  – Does not have control flow (as we are used to thinking of it)

  *So, what does it have?*

• A program in a logic programming language consists of a set of *declarations* related together using *predicate calculus*.

• An algorithm in a LP language = logic + control
  – Logic: programmer provides the “logic” which is what the program does.
  – Control: language run-time system provides the control (!)
LP: operators in predicate calculus

- Connectors:

<table>
<thead>
<tr>
<th>Name</th>
<th>Book</th>
<th>Ex</th>
<th>Alt</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>¬</td>
<td>¬a</td>
<td>!</td>
<td>~</td>
</tr>
<tr>
<td>conjunction</td>
<td>∩</td>
<td>a∩b</td>
<td>∧</td>
<td>&amp;</td>
</tr>
<tr>
<td>disjunction</td>
<td>∪</td>
<td>a∪b</td>
<td>∨</td>
<td></td>
</tr>
<tr>
<td>equivalence</td>
<td>≡</td>
<td>a≡b</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>implication</td>
<td>⊃</td>
<td>a⊃b</td>
<td>⇒</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a  b</td>
<td>←</td>
<td>:-</td>
<td>b implies a</td>
</tr>
</tbody>
</table>

Precedence: ¬ then ∩ ∪ ≡ then ⊃

Examples:
- a ∩ b ⊃ c
- a ∪ b ⊃ c
- a ∪ (b ∩ c) ⊃ d
- a ∨ (b ∧ c) ⊃ d

- Quantifiers:

Universal  ∀X.P  For all X, P is true
Existential  ∃X.P  There exists a value of X such that P is true

(often leave out the ".")

Examples:
- ∀X.(teachingFaculty(X) ⊃ facultyMember(X))
- ∀X.(teachingFaculty(X) ⊃ ∃Y.teaches(X,Y))
A proposition is a *logical statement* that may or may not be true.

Consists of *objects* and their *relationships* to each other

Propositions are written in a *mathematical function form*
  - E.g. A is a B written as B(A)

Propositions have no intrinsic semantics.
  - Do not supply meaning, just ids. We are actually interpreting them.
LP: atomic vs. compound propositions

- **Atomic Proposition:**
  - Simplest form of logical statements
  - Made up of two parts: *functor* and *parameters*
  - E.g.
    - Mary is a woman \(\text{woman}(\text{mary})\)
    - Tom and Mary are married \(\text{married}(\text{tom}, \text{mary})\)
    - Scott teaches CSC341 in Summer \(\text{teaches}(\text{scott}, \text{CSC341}, \text{Summer})\)

- **Compound propositions:**
  - Two or more atomic propositions connected with *logical connectors*
  - E.g.
    - Tom is either smart or dumb \(\text{smart}(\text{tom}) \lor \text{dumb}(\text{tom})\)
    - Tom is not dumb \(\neg \text{dumb}(\text{tom})\)
    - Tom is married to someone \(\exists X [\text{married}(\text{tom}, X)]\)
    - Tom loves everything \(\forall X [\text{loves}(\text{tom}, X)]\)
    - Tom is married to a human female \(\exists X [\text{married}(\text{tom}, X) \land \text{female}(X) \land \text{human}(X)]\)
LP: implication in predicate calculus

• Propositions related with each other by an *if-then* semantics, can be expressed using logical implication (denoted by \( \Rightarrow \))

• Examples:
  – If someone breaks the law, then she/he will be sent to jail or given a fine but not both.
    • P is breaks the law, Q is sent to jail, R is given a fine, v is a variable
    • \( P(v) \Rightarrow [(Q(v) \lor R(v)) \land \neg(Q(v) \land R(v))] \)
  – If December is a cold dark month then January is a cold dark month
    • P is dark, Q is cold, R is a month, d December and j January
    • \([P(d) \land Q(d) \land R(d)] \Rightarrow [P(j) \land Q(j) \land R(j)]\)
    • Literally: *if December is cold, and December is dark and December is a month then January is cold, and January is dark and January is a month*
  – There exists at least one x, such that x is a country and x is ruled by a Queen.
    • P is a country, Q ruled by a Queen, x is a variable
    • \((\exists x) (P(x) \land Q(x))\)
    • Literally: *there is an X that is both P and Q*
Examples cont’d:

- Every person who is smart is also rich:
  \[(\forall X) [\text{person}(X) \land \text{smart}(X) \Rightarrow \text{rich}(X)]\]

- John has exactly one mother:
  \[(\exists X) [\text{mother}(\text{John}, X) \land \text{mother}(\text{John}, Y) \Rightarrow Y = X]\]

- All artists, except poor ones, are rich:
  \[(\forall X) [ (\text{artist}(X) \land \neg \text{poor}(X) ) \Rightarrow \text{rich}(X) ]\]