Programming in Standard ML: Continued

Specification and Verification with Higher-Order Logic

Arnd Poetzsch-Heffter
(Slides by Jens Brandt)

Software Technology Group
Fachbereich Informatik
Technische Universität Kaiserslautern

Sommersemester 2008

Outline

Outline

Previous Lecture: Fundamentals

- built-in data types
- functions
- type inference
- exceptions

This Lecture: Deeper Insight

- pattern matching
- case analysis
- data type definitions
- modules

Outline

Tuples

Example (Tuples)

```
- val pair = (2,3);
> val pair = (2, 3) : int * int

- val triple = (2,2.0,"2");
> val triple = (2, 2.0, "2") : int * real * string

- val pairs_of_pairs = ((2,3),(2.0,3.0));
> val pairs_of_pairs = ((2, 3), (2.0, 3.0)) : (int * int) * (real * real)
```

Example (Unit Type)

```
- val null_tuple = ();
> val null_tuple = () : unit
```

Accessing Components

Example (Navigating to the Position)

```
- val xy1 = #1 pairs_of_pairs;
> val xy1 = (2, 3) : int * int

- val y1 = #2 (#1 pairs_of_pairs);
> val y1 = 3 : int
```

Example (Using Pattern Matching)

```
- val ((x1,y1),(x2,y2)) = pairs_of_pairs;
> val x1 = 2 : int
  val y1 = 3 : int
  val x2 = 2.0 : real
  val y2 = 3.0 : real
```

Pattern Matching

Example (Granularity)

```
- val ((x1,y1),xy2) = pairs_of_pairs;
> val x1 = 2 : int
  val y1 = 3 : int
  val xy2 = (2.0, 3.0) : real * real
```

Example (Wildcard Pattern)

```
- val ((_,y1),(_,_)) = pairs_of_pairs;
> val y1 = 3 : int

- val ((_,y1),_) = pairs_of_pairs;
> val y1 = 3 : int
```

Pattern Matching

Example (Value Patterns)

```
- val 0 = 1-1;

- val (0,x) = (1-1,34);
> val x = 34 : int

- val (0,x) = (2-1,34);
! Uncaught exception:
! Bind
```

Binding Values

General Rules

- The variable binding **val** var = **val** is irreducible.
- The wildcard binding **val** _ = **val** is discarded.
- The tuple binding **val**(pat1, ... , patN) = (val1, ... , valN) is reduced to

  val pat1 = valN

  ...

  val patN = valN

Clausal Function Expressions

Example (Clausal Function Expressions)

```
- fun not true = false
    | not false = true;
> val not = fn : bool -> bool
```

Redundant Cases

Example (Redundant Cases)

```
- fun not True = false
    | not False = true;
! Warning: some cases are unused in this match.
> val 'a not = fn : 'a -> bool

- not false;
> val it = false : bool
- not 3;
> val it = false : bool
```

Fact (Redundant Cases)

Redundant cases are always a mistake!

Inexhaustive Matches

Example (Inexhaustive Matches)

```
fun first_ten 0 = true | first_ten 1 = true | first_ten 2 = true
  | first_ten 3 = true | first_ten 4 = true | first_ten 5 = true
  | first_ten 6 = true | first_ten 7 = true | first_ten 8 = true
  | first_ten 9 = true;
! Warning: pattern matching is not exhaustive
> val first_ten = fn : int -> bool
- first_ten 5;
> val it = true : bool
 first_ten ~1;
! Uncaught exception: Match
```

Fact (Inexhaustive Matches)

Inexhaustive matches may be a problem.

Catch-All Clauses

Example (Catch-All Clauses)

```
fun first_ten 0 = true | first_ten 1 = true | first_ten 2 = true
  | first_ten 3 = true | first_ten 4 = true | first_ten 5 = true
  | first_ten 6 = true | first_ten 7 = true | first_ten 8 = true
  | first_ten 9 = true | first_ten _ = false;
> val first_ten = fn : int -> bool
```

Overlapping Cases

Example (Overlapping Cases)

```
- fun foo1 1 _ = 1
    | foo1 _ 1 = 2
    | foo1 _ _ = 0;
> val foo1 = fn : int -> int -> int
- fun foo2 _ 1 = 1
    | foo2 1 _ = 2
    | foo2 _ _ = 0;
> val foo2 = fn : int -> int -> int

- foo1 1 1;
> val it = 1 : int
- foo2 1 1;
> val it = 1 : int
```

Recursively Defined Functions

Example (Recursively Defined Function)

```
- fun factorial 0 = 1
    | factorial n = n * factorial (n-1);
> val factorial = fn : int -> int

- val rec factorial = fn
```

Example (Recursively Defined Lambda Abstraction)

```
- val rec factorial =
fn 0 => 1
         | n => n * factorial (n-1);
```

Mutual Recursion

Example (Mutual Recursion)

```
- fun even 0 = true
    | even n = odd (n-1)
  and odd 0 = false
    | odd n = even (n-1);
> val even = fn : int -> bool
  val odd = fn : int -> bool

- (even 5,odd 5);
> val it = (false, true) : bool * bool
```

Outline

Type Abbreviations

type Keyword

- type abbreviations
- record definitions

Example (Type Abbreviation)

```
- type boolPair = bool * bool;
> type boolPair = bool * bool

- (true,true):boolPair;
> val it = (true, true) : bool * bool
```

Defining a Record Type

Example (Record)

```
- type hyperlink =
    { protocol : string, address : string, display : string };
> type hyperlink = {address : string, display : string, protocol : string}

- val hol_webpage = {
    protocol="http",
    address="rsg.informatik.uni-kl.de/teaching/hol",
    display="HOL-Course" };
> val hol_webpage = {
    address = "rsg.informatik.uni-kl.de/teaching/hol",
    display = "HOL-Course",
    protocol = "http"}
      :{address : string, display : string, protocol : string}
```

Accessing Record Components

Example (Type Abbreviation)

```
- val {protocol=p, display=d, address=a } = hol_webpage;
> val p = "http" : string
  val d = "HOL-Course" : string
  val a = "rsg.informatik.uni-kl.de/teaching/hol" : string

- val {protocol=_, display=_, address=a } = hol_webpage;
> val a = "rsg.informatik.uni-kl.de/teaching/hol" : string

- val {address=a, ...} = hol_webpage;
> val a = "rsg.informatik.uni-kl.de/teaching/hol" : string

- val {address, ...} = hol_webpage;
> val address = "rsg.informatik.uni-kl.de/teaching/hol" : string
```

Defining *Really* New Data Types

datatype Keyword

programmer-defined (recursive) data types, introduces

- one or more new type constructors
- one or more new value constructors

Non-Recursive Data Type

Example (Non-Recursive Datatype)

```
- datatype threeval = TT | UU | FF;
> New type names: =threeval
  datatype threeval =
  (threeval ,{con FF : threeval , con TT : threeval , con UU : threeval})
  con FF = FF : threeval
  con TT = TT : threeval
  con UU = UU : threeval

- fun not3 TT = FF
    | not3 UU = UU
    | not3 FF = TT;
> val not3 = fn : threeval -> threeval

- not3 TT;
> val it = FF : threeval
```

Parameterised Non-Recursive Data Types

Example (Option Type)

```
- datatype 'a option = NONE | SOME of 'a;
> New type names: =option
  datatype 'a option =
  ('a option,{con 'a NONE : 'a option, con 'a SOME : 'a -> 'a option})
  con 'a NONE = NONE : 'a option
  con 'a SOME = fn : 'a -> 'a option
```

- constant NONE
- values of the form SOME *v* (where *v* has the type 'a)

Option Types

Example (Option Type)

```
- fun reciprocal 0.0 = NONE
    | reciprocal x = SOME (1.0/x)
> val reciprocal = fn : real -> real option

- fun inv_reciprocal NONE = 0.0
    | inv_reciprocal (SOME x) = 1.0/x;
> val inv_reciprocal = fn : real option -> real
- fun identity x = inv_reciprocal (reciprocal x);
> val identity = fn : real -> real

- identity 42.0;
> val it = 42.0 : real
- identity 0.0;
> val it = 0.0 : real
```

Recursive Data Types

Example (Binary Tree)

```
- datatype 'a tree =
Empty |
Node of 'a tree * 'a * 'a tree;
> New type names: =tree
  datatype 'a tree =
  ('a tree,
   {con 'a Empty : 'a tree,
    con 'a Node : 'a tree * 'a * 'a tree -> 'a tree})
  con 'a Empty = Empty : 'a tree
  con 'a Node = fn : 'a tree * 'a * 'a tree -> 'a tree
```

- Empty is an empty binary tree
- (Node (t_1, v, t_2) is a tree if t_1 and t_2 are trees and v has the type 'a
- nothing else is a binary tree

Functions and Recursive Data Types

Example (Binary Tree)

```
- fun treeHeight Empty = 0
    | treeHeight (Node (leftSubtree, _, rightSubtree)) =
        1 + max(treeHeight leftSubtree, treeHeight rightSubtree);
> val 'a treeHeight = fn : 'a tree -> int
```

Mutually Recursive Datatypes

Example (Binary Tree)

```
- datatype 'a tree =
    Empty |
    Node of 'a * 'a forest
  and 'a forest =
   None |
   Tree of 'a tree * 'a forest;
> New type names: =forest, =tree
  ...
```

Abstract Syntax

Example (Defining Expressions)

```
- datatype expr =
      Numeral of int |
      Plus of expr * expr |
      Times of expr * expr;
> New type names: =expr
  datatype expr =
  (expr,
   {con Numeral : int -> expr,
    con Plus : expr * expr -> expr,
    con Times : expr * expr -> expr})
  con Numeral = fn : int -> expr
  con Plus = fn : expr * expr -> expr
  con Times = fn : expr * expr -> expr
```

Abstract Syntax

Example (Evaluating Expressions)

```
- fun eval (Numeral n) = Numeral n
    | eval (Plus(e1,e2)) =
              let val Numeral n1 = eval e1
                  val Numeral n2 = eval e2 in
                Numeral(n1+n2) end
    | eval (Times (e1,e2)) =
          let val Numeral n1 = eval e1
                  val Numeral n2 = eval e2 in
                Numeral(n1*n2) end;
> val eval = fn : expr -> expr

- eval( Plus( Numeral 2, Times( Numeral 5, Numeral 8 ) ) );
> val it = Numeral 42 : expr
```

Outline

Structuring ML Programs

Modules

- structuring programs into separate units
- program units in ML: *structures*
- contain a collection of types, exceptions and values (incl. functions)
- parameterised units possible
- composition of structures mediated by *signatures*

Structures

Purpose

- structures = implementation

Example (Structure)

```
structure Queue =
struct
  type 'a queue = 'a list * 'a list
  val empty = (nil, nil)
  fun insert(x, (bs,fs)) = (x::bs, fs)
  exception Empty
  fun remove (nil, nil) = raise Empty
  | remove (bs, f::fs) = (f, (bs,fs))
  | remove (bs, nil)= remove (nil, rev bs)
end
```

Accessing Structure Components

Identifier Scope

- components of a structure: local scope
- must be accessed by qualified names

Example (Accessing Structure Components)

```
- Queue.empty;
> val ('a, 'b) it = ([], []) : 'a list * 'b list

- open Queue;
> ...
- empty;
> val ('a, 'b) it = ([], []) : 'a list * 'b list
```

Accessing Structure Components

Usage of **open**

- open a structure to incorporate its bindings directly
- cannot open two structures with components that share a common names
- prefer to use open in **let** and **local** blocks

Signatures

Purpose

- signatures = interface

Example (Signature)

```
signature QUEUE =
sig
  type 'a queue
  exception Empty
  val empty : 'a queue
  val insert : 'a * 'a queue -> 'a queue
  val remove: 'a queue -> 'q * 'a queue
end
```

Signature Ascription

Transparent Ascription

- descriptive ascription
- extract principal signature
  - always existing for well-formed structures
  - most specific description
  - everything needed for type checking
- source code needed

Opaque Ascription

- restrictive ascription
- enforce data abstraction

Opaque Ascription

Example (Opaque Ascription)

```
structure Queue :> QUEUE
struct
  type 'a queue = 'a list * 'a list
  val empty = (nil, nil)
  fun insert(x, (bs,fs)) = (x::bs, fs)
  exception Empty
  fun remove (nil, nil) = raise Empty
  | remove (bs, f::fs) = (f, (bs,fs))
  | remove (bs, nil)= remove (nil, rev bs)
  end
```

Signature Matching

Conditions

- structure may provide more components
- structure may provide more general types than required
- structure may provide a concrete datatype instead of a type
- declarations in any order

Modular Compilation in Moscow ML

Compiler **mosmlc**

- save structure Foo to file `Foo.sml`
- compile module: mosmlc Foo.sml
- compiled interface in Foo.ui and compiled bytecode Foo.uo
- load module load "Foo.ui"

```
- load "Queue";
> val it = () : unit
- open Queue;
> type 'a queue = 'a list * 'a list
  val ('a, 'b) insert = fn : 'a * ('a list * 'b) -> 'a list * 'b
  exn Empty = Empty : exn
  val ('a, 'b) empty = ([], []) : 'a list * 'b list
  val 'a remove = fn : 'a list * 'a list -> 'a * ('a list * 'a list )
```

Outline

Summary

Summary

- programming in Standard ML
  - case analysis and pattern matching
  - data type definitions
  - modules
- This is the end of the ML programming course. :-)

Outlook: Next Week

- theorem-proving fundamentals
- theorem-prover principles and architecture