Interactive Theorem Proving in HOL4

Course 01: Programming in Standard ML

Dr Chun TIAN
chun.tian@anu.edu.au

Australian
National
University

1 August 2024

Acknowledgement of Country

We acknowledge and celebrate the First Australians on whose traditional lands we meet, and pay our respect to the elders past and present.

More information about Acknowledgement of Country can be found here and here

Introduction

- Standard ML (SML) is a general programming language in the ML (*Meta Language*) family;
- The HOL theorem prover (HOL4) is written in Standard ML;
- By writing *formal proofs* in HOL4, one is actually writing *programs* in SML;
- Programs as formal proofs run, to generate SML structures (modules) containing names and statements of the proved theorems (i.e. no proofs);
- When a proof of a theorem uses another theorems of a different theory, only the statements (without proofs) of the other theorem is used;
- SML implementations: Poly/ML (preferred), Moscow ML, MLton, SML/NJ, ...
- Only limited language features are useful in writing formal proofs (scope of this course).

A basic subset of SML

Topics

- Types and Values
- Declarations
- Functions
- Case analysis
- Recursive Functions
- Programming with Lists
- Side Effects

Reference

See *R. Harper, "Programming in Standard ML," 2012.* for more details.

Types and Values

A typical SML program is a sequence of value assignments:

```
val var : type = value;
```

NOTE: Functions are also values stored in variables (as function names). i.e. *functions and variables in SML have the same namespace.*

Types in SML (1)

Some primitive types in SML

Name	Sample values	Some operations	Structure
`bool`	`true`, `false`	`not`, `andalso`, `orelse`, $=$	`Bool`
`real`	`3.14`, `2.17`, `0.1E6`	$+, -, *, /, =, <$	`Real`
`int`	`2`, `~1`, `0`	$+, -, *, =, <$, `div`, `max`	`Int`
`char`	`#"a"`, `#"b"`	`ord`, `chr`, $=, <$	`Char`
`string`	`"abc"`, `"1234"`	`^`, `size`, $=, <$	`String`
`unit`	`()`	-	-

NOTE: The command `"open Int;"` shows all built-in operations defined on the type `int`.

Parametric Types

`'a list`, `'a option`, etc. Thus "`string list`", "`int option`", etc. are also valid types.

Types in SML (2)

Product Types

If the type of `x` is `A`, the type of `y` is `B`, then the type of `(x, y)` is `A * B`.

```
> (1,2);
val it = (1, 2): int * int
> (1,2.5);
val it = (1, 2.5): int * real
```

Function Types

The type of the function taking an argument of type `A` and returns a value of type `B`, is `A -> B`.

```
> val even = fn i => i mod 2 = 0;
val even = fn: int -> bool
> even 4;
val it = true: bool
```

Declarations

- Type bindings: `type float = real;` `type count = int and average = real;`
- Value bindings: `val m :int = 3+2;` `val pi :real = 3.14 and e :real = 2.17;`
- Limiting scopes:

```
val n = let val m = 2;
in
   m + m;
end
```

- Another way:

```
local val m = 2;
in
   val n = m + m;
end
```


Functions

Primitive functions

```
> (fn x : real => Math.sqrt (Math.sqrt x)) (16.0);
val it = 2.0: real

val fourthroot : real -> real =
    fn x : real => Math.sqrt (Math.sqrt x)
```

Named functions

```
fun fourthroot (x:real):real = Math.sqrt (Math.sqrt x)

> fun add1 (x,y) = x + y + 1;
val add1 = fn: int * int -> int
> fun add2 x y = x + y + 1;
val add2 = fn: int -> int -> int
```

Tuples and Patterns

```
val pair : int * int = (2, 3)
val triple : int * real * string = (2, 2.0, "2")
val quadruple
: int * int * real * real
= (2,3,2.0,3.0)
val pair of pairs
: (int * int) * (real * real)
= ((2,3),(2.0,3.0))

val (m:int, n:int) = (7+1,4 div 2)
val (m:int, r:real, s:string) = (7, 7.0, "7")
val ((m:int,n:int), (r:real, s:real)) = ((4,5),(3.1,2.7))
val (m:int, n:int, r:real, s:real) = (4,5,3.1,2.7)
```


Case Analysis

```
fun recip 0 = 0
| recip (n:int) = 1 div n

fun not true = false
| not false = true
```

if-then-else

The *conditional* expression "`if exp then exp1 else exp2`" is short-hand for the case analysis:

```
case exp
of true => exp1
| false => exp2
```

Recursive Functions

```
fun factorial 0 = 1
| factorial (n:int) = n * factorial (n-1)
```

Iteration (as the "for" loop)

```
local
fun helper (0,r:int) = r
| helper (n:int,r:int) = helper (n-1,n*r)
in
  fun factorial (n:int) = helper (n,1)
end
```

Mutual Recursion

```
fun even 0 = true
| even n = odd (n-1)
and odd 0 = false
| odd n = even (n-1)
```

Programming with Lists

Definition of lists

- `nil` is a value of type "*typ* `list`".
- if `h` is a value of type *typ*, and `t` is a value of type *typ* `list`, then `h::t` is a value of type *typ* `list`.
- Nothing else is a value of type *typ* `list`.

```
fun length nil = 0
| length (_::t) = 1 + length t

fun append (nil, l) = l
| append (h::t, l) = h :: append (t, l)

fun rev nil = nil
| rev (h::t) = rev t @ [h]
```

Datatypes

The option type

```
datatype 'a option = NONE | SOME of 'a
val isSome = fn: 'a option -> bool
val valOf = fn: 'a option -> 'a
> valOf (SOME 1);
val it = 1: int
```

Recursive Datatypes

```
datatype 'a tree =
    Empty | Node of 'a tree * 'a * 'a tree
> Node;
val it = fn: 'a tree * 'a * 'a tree -> 'a tree
> Node (Empty,1,Empty);
val it = Node (Empty, 1, Empty): int tree
```

Side Effects

```
> val r = ref 0;
> val s = ref 0;
> val _ = r := 3;
> r;
val r = ref 3: int ref

> val x = !s + !r
val x = 3: int
```

The `print` function

```
> print;
val it = fn: string -> unit

> val _ = print "Hello, World!\n";
Hello, World!
```