Functional programming Primer I

COS 320

Compiling Techniques

Princeton University
Spring 2016

Lennart Beringer

Characteristics of functional programming

- Primary notions:
  functions and expressions (not commands);
- Primary operations (not sequencing):
  - expression evaluation
  - function formation and application

- expression evaluation in (top-level) interpreter: - 3+4;
- binding value to an identifier: - val x = 3+4;

Basic function formation:

fun SimpleCompiler (input) = backend (frontend (input))

(types often inferred from types of operands and arguments)

Or: val SimpleCompiler = fn input => (backend **o** frontend) (input)

Characteristics of functional programming cont'd

Guiding principles:

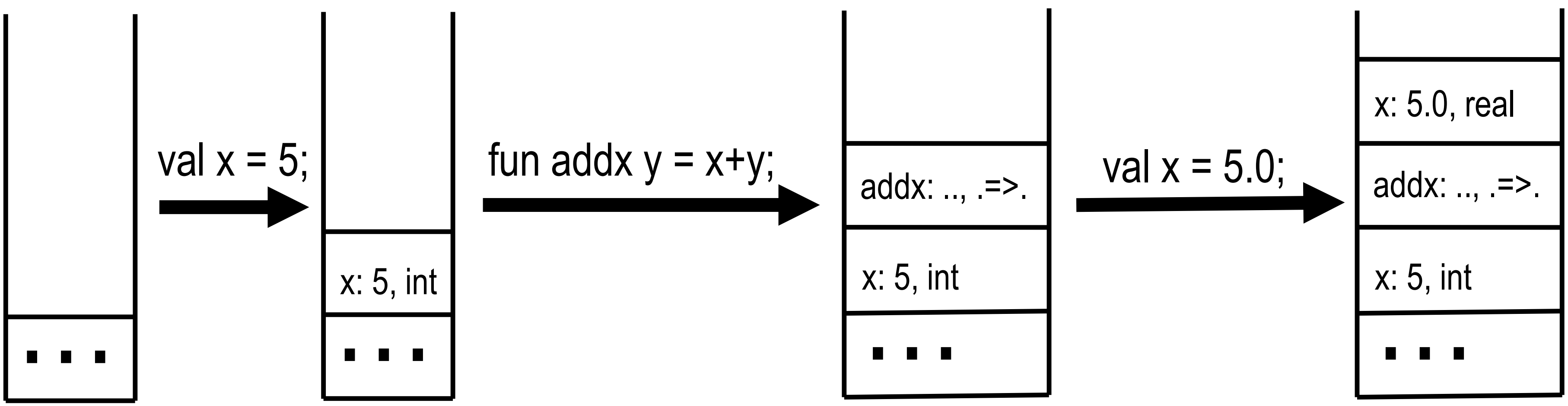
- generally avoid side-effects whenever possible (exception: IO, …)
- referential transparency: can substitute expression that yield equal results

- Variables act as names, not storage cells (registers)
- Statically typed: eliminates many programming mistakes
- Higher-order functions (functions as args & return values)
- Inductive data types, pattern matching
- Parametric polymorphism, with type inference
- (possibly mutually) recursive functions instead of loops
- disciplined model of memory and exceptions
- expressive module system
- Libraries:
  - SML Base library: http://sml-family.org/Basis
  - SMLNJ library: http://www.smlnj.org/doc/smlnj-lib

ML interpreter's evaluation environment

Declarations are interpreted in context of previous declarations

- **top-level** declarations push item onto "stack" but no pop operation
- later declarations use most recent declaration



- a **local** declaration like **let val x = e1 in e2 end** temporarily extends the current environment with a binding for **x** for the duration of **e2**, so pops the binding for **x** from stack after **e2** has finished

Composite expression formation: let-binding

- Naming of intermediate values, with explicit scope

expression involving previously introduced names

expression that may also mention n

let val n = e_1 in e_2 end

- **Scope** of n is e2: if there's another surrounding introduction of n, the "local" n hides the outer one only for the duration of e2 ("outer" n is reestablished after e2).
- n is **bound** (to the value resulting from evaluating e1) **in** e2

Bound and free occurrences of variables

- Let-bindings, function parameters, and pattern matches (below) **bind** variables/names in their respective scope.
- Occurrences of variables that are not bound are **free**.
- Note: an expression may contains bound and free occurrences of the same name.

Example:

```
let x = let x = x*x in x+y end
in let x = x+1 in x end
end
```

Informal disambiguation:

```
let x = let x = x*x in x+y end
in let x = x+1 in x end
end
```

α-renaming

- Renaming a **bound** variable does not change the meaning of an expression

```
Example:
let x = let x = x*x in x+y end
in let x = x+1 in x end
end
```

```
Informal disambiguation:
let x = let x = x*x in x+y end
in let x = x+1 in x end
end
```

```
let a = let z = x*x in z+y end
in let b = a+1 in b end
end
```

(one aspect of referential transparency)

ML types

- Base types:
  - int: (example values: 1, 4, **~**3, 0)
  - reals (example values: 0.0, 3.5, 1.23E**~**10)
  - Strings (“abc\n”)
- Tuples/products: **A * B** (in general: **A_1 * … * A_n**)
  - formation **(**1**,** 3.5**)** : int * real
  - elimination: **fst** p, **snd** p, **#i** p
- empty product: **unit**, with value **()**
- Function space **A -> B**
  - formation: **fn (x:A):B => e**
  - elimination: application **f e**
- Records: **{ lab1:A1, …, labn:An }** (order “irrelevant”)
- Polymorphic types (types containing type variables ‘a, ‘b…)
  - occur typically in combination with (higher-order) functions, and inductive datatypes

Inductive data types

Example: (polymorphic) binary trees

- **Definition** of recursive, polymorphic type:
  datatype ’a bintr = LEAF of ’a | NODE of ’a bintr * ’a bintr;

Inductive data types

Example: (polymorphic) binary trees

- **Definition** of recursive, polymorphic type:
  datatype ’a bintr = LEAF of ’a | NODE of ’a bintr * ’a bintr;

- Constructing values using **constructors**
  LEAF: ’a => ‘a bintree and NODE: ’a bintr * ’a bintr => ‘a bintr
  Example: val mytree1 = LEAF 1; (*yields mytree1: **int** bintr*)

Inductive data types

Example: (polymorphic) binary trees

- **Definition** of recursive, polymorphic type:
  datatype ’a bintr = LEAF of ’a | NODE of ’a bintr * ’a bintr;

- Constructing values using **constructors**
  LEAF: ’a => ‘a bintree and NODE: ’a bintr * ’a bintr => ‘a bintr
  Example: val mytree1 = LEAF 1; (*yields mytree1: **int** bintr*)

- Destructing/inspecting values by **pattern matching**
  ```
  fun height t = case t of LEAF l => 1
                         | NODE (left, right) => 1 + max (height l, height r);
  ```
  (*yields val height = fn : ‘a bintree -> int*)

Inductive data types

Example: (polymorphic) binary trees

- **Definition** of recursive, polymorphic type:
  datatype 'a bintr = LEAF of 'a | NODE of 'a bintr * 'a bintr;

- Constructing values using **constructors**
  LEAF: 'a => 'a bintree and NODE: 'a bintr * 'a bintr => 'a bintr
  Example: val mytree1 = LEAF 1; (*yields mytree1: **int** bintr*)

- Destructing/inspecting values by **pattern matching**

```
fun height t = (case t of LEAF l => 1
                        | NODE (left, right) => 1 + max (height l, height r));
(*yields val height = fn : 'a bintree -> int*)
```

recommendation: add parens!

Inductive data types

Example: (polymorphic) binary trees

- **Definition** of recursive, polymorphic type:
  datatype 'a bintr = LEAF of 'a | NODE of 'a bintr * 'a bintr;

- Constructing values using **constructors**
  LEAF: 'a => 'a bintree and NODE: 'a bintr * 'a bintr => 'a bintr
  Example: val mytree1 = LEAF 1; (*yields mytree1: **int** bintr*)

- Destructing/inspecting values by **pattern matching**

```
fun height t = (case t of LEAF l => 1
                        | NODE (left, right) => height l + height r);
(*yields val height = fn : 'a bintree -> int*)
```

recommendation: add parens!

- Datatypes don't need to be recursive, 0-ary constructors ok:
  datatype colors = RED | GREEN | BLUE;

Higher-order and mutually recursive functions

- Can use functions as parameters/arguments and return values of functions

```
fun twice f x = f (f x);  (*yields val twice = fn : ('a -> 'a) -> 'a -> 'a*)

fun add x = fn y -> x+y;  (*yields val add = fn: int -> int -> int *)

val h = twice (add 3); (*yields val h = fn: int -> int *)

val z = h 7; (*yields val z = 13*)
```

Higher-order and mutually recursive functions

- Can use functions as parameters/arguments and return values of functions

```
fun twice f x = f (f x);  (*yields val twice = fn : ('a -> 'a) -> 'a -> 'a*)
fun add x = fn y -> x+y;  (*yields val add = fn: int -> int -> int *)
val h = twice (add 3); (*yields val h = fn: int -> int *)
val z = h 7; (*yields val z = 13*)
```

- Definition of mutually recursive functions (used in parser)

```
datatype nat = Succ of nat | Zero;
fun even n = (case n of Zero => true | Succ m => odd m)
and odd n = (case n of Zero => false | Succ m => even m);
val SEVEN = Succ (… (Succ Zero)…);
even SEVEN;
```

Boolean conjunction is called andalso, not and

References in ML

Type of "mutable references of type T": ref T

- can hold values of type T

References in ML

Type of "mutable references of type T": ref T

- can hold values of type T
- **creation**-"allocation+initialization": ref e, where e:T holds, i.e. e is an expression of type T
  - evaluate e to a value v; then put v into a fresh ref cell
  - typical use: let val x = ref e in … end
  - type T has certain restrictions regarding polymorphism…

References in ML

Type of "mutable references of type T": ref T

- can hold values of type T
- **creation**-"allocation+initialization": ref e, where e:T holds, i.e. e is an expression of type T
  - evaluate e to a value v; then put v into a fresh ref cell
  - typical use: let val x = ref e in … end
  - type T has certain restrictions regarding polymorphism…
- **read access**: !e, where e:ref T
  - evaluate e to a value v (v: ref T will be satisfied!)
  - return the content **u** of cell v. **u**:T will hold!

References in ML

Type of "mutable references of type T": ref T

- can hold values of type T
- **creation**-"allocation+initialization": ref e, where e:T holds, i.e. e is an expression of type T
  - evaluate e to a value v; then put v into a fresh ref cell
  - typical use: let val x = ref e in … end
  - type T has certain restrictions regarding polymorphism…
- **read access**: !e, where e:ref T
  - evaluate e to a value v (v: ref T will be satisfied!)
  - return the content **u** of cell v. **u**:T will hold!
- **write access**: e:=e' where e:ref T and e':T
  - evaluate e and e', yielding values **v**: ref T and **v'**:T
  - store **v'** in **v**. Type of ref-cell remains unmodified!

References in ML

Type of "mutable references of type T": ref T

- can hold values of type T
- **creation**-"allocation+initialization": ref e, where e:T holds, i.e. e is an expression of type T
  - evaluate e to a value v; then put v into a fresh ref cell
  - typical use: let val x = ref e in … end
  - type T has certain restrictions regarding polymorphism…
- **read access**: !e, where e:ref T
  - evaluate e to a value v (v: ref T will be satisfied!)
  - return the content **u** of cell v. **u**:T will hold!
- **write access**: e:=e' where e:ref T and e':T
  - evaluate e and e', yielding values **v**: ref T and **v'**:T
  - store **v'** in **v**. Type of ref-cell remains unmodified!

No uninitialized memory cells!

No nil pointers – no nil pointer exceptions!

Content guaranteed to be type-correct: no casting

Practicalities

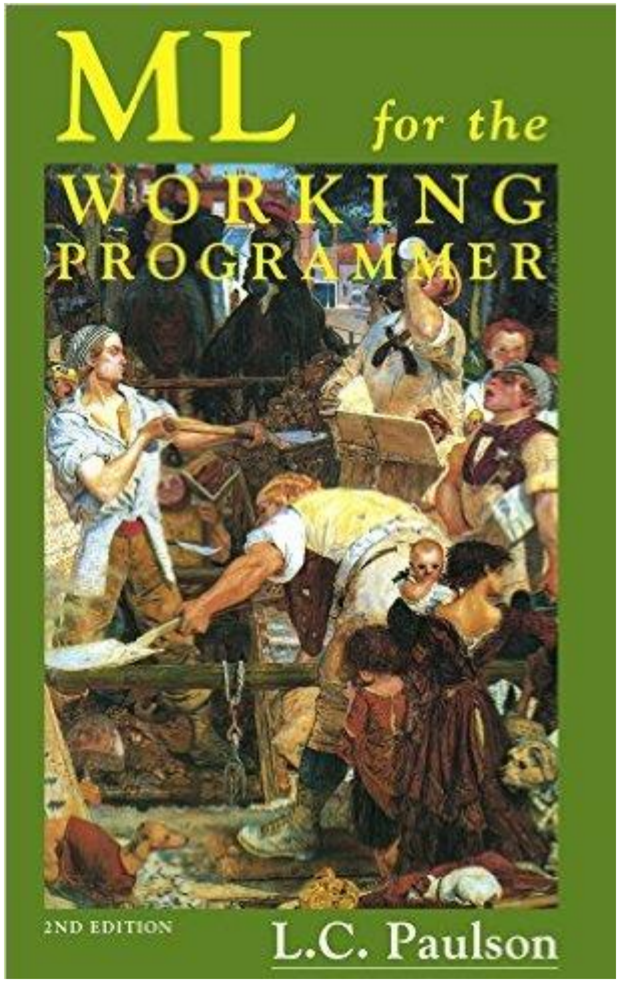
- Loading files:
  - - use myfile.sml;
  - myfile.sml may include subordinate use statements
- Opening (library) structures: - open Math;
- Quitting the interpreter:

  Unix: ctrl-D   Windows: ctrl-Z

  Or call OS.Process.exit(OS.Process.success);
- Emacs mode: see info pages of SMLNJ

Compilation manager CM: see assignment1

Comprehensive details



Cambridge University Press
PU library

Programming in Standard ML

(DRAFT: VERSION 1.2 OF 11.02.11.)

Robert Harper
Carnegie Mellon University

Spring Semester, 2011

http://www.cs.cmu.edu/~rwh/smlbook

More links to books and doc at http://www.smlnj.org