

ICS/CSE 141: Programming Languages

More ML & Types

School of Information and Computer Science
University of California, Irvine

Chris Stork
`cstork@ics.uci.edu`

Overview

1. Algebraic Datatypes
2. Imperative Features
3. Types

Algebraic Datatypes

Enumerations

Yet another way to define **new** types in ML

- the `datatype` declaration introduces **two** things
- a new **type** that ML will use for type inference
- **constructors** that “build” values of that type

The simplest possible thing: **Enumerations**

- `datatype color = Red | Blue | Green; ⇒`
`datatype color = (color, con Blue : color,`
`con Green : color, con Red : color) ...`
- details of this are not important right now...
- `Red; ⇒ val it = Red : color`

Simple Datatypes

The **constructors** can also take “arguments”

- We want to keep track of **degrees**, subjects, schools, and advisors...
- ```
datatype degree = BS of string
 | MS of string*string
 | PhD of string*string*string;
```
- ```
BS("Biology"); ⇒ val it = BS "Biology" : degree
```
- ```
PhD("Computer Science","UC Irvine","Franz"); ⇒
val it = PhD("Computer Science", "UC Irvine",
"Franz") : degree
```

Remember to explicitly use the constructors...

- ```
datatype X = Y of int list;
```
- ```
[1,2]; ⇒ val it = [1, 2] : int list
```
- ```
Y [1,2]; ⇒ val it = Y [1, 2] : X
```

Inductively Defined Functions

Use **patterns** to define functions for algebraic datatypes

```
fun strdeg(BS(sub)) =  
  "Bachelor in " ^ sub  
| strdeg(MS(sub, sch)) =  
  "Master in " ^ sub ^ " from " ^ sch  
| strdeg(PhD(sub, sch, ad)) =  
  "PhD in " ^ sub ^ " from " ^ sch ^ " under " ^ ad;
```

Now we can use the function to “print” degrees...

- `BS("Bla");` \Rightarrow `val it = BS "Bla" : degree`
- `strdeg it;` \Rightarrow `val it = "Bachelor in Bla" : string`

Recursive Types

datatype declarations can be recursive:

```
datatype tree = Leaf of int
              | Node of (tree * tree);
fun sumLeaves (Leaf n) = n
  | sumLeaves (Node (t1,t2)) = sumLeaves t1 + sumLeaves t2;

val t = Node(Leaf 1, Node(Leaf 2, Leaf 3));
sumLeaves t;
```

Imperative Features

No Assignment so far!

In ML, the value of an identifier cannot be changed!

In other words, declarations only introduce **constants**, not **variables**!

Is that weird?

No Assignment so far!

In ML, the value of an identifier cannot be changed!

In other words, declarations only introduce **constants**, not **variables**!

Is that weird? Not really. Bare with me...

Consider C, Pascal, Java, or C++:

- Identifiers declared as regular variables of type integer or string are assignable.
- Identifiers declared as functions are constants, though!
- So, depending on what the identifier is referring to, it is either a constant or a variable.

In ML, a `val` declaration works the same for *all* types of values!

But how do we deal with mutable data?

Review of Variables

Consider the instruction $x := x + 1$ in imperative languages...

- we “know” that x must be a **variable** for this to make sense
 - we “know” that a declaration (and initialization) like `int x := 0` was used
 - to introduce a “box” with the name x that can “hold” an integer
- Okay, but now what does x **mean** after its declaration?
 - does it mean the memory location (“box”) itself? L-value
 - does it mean the value (“content”) it holds? R-value
- **Argh!** It means **both!** How does that make any sense?
 - What is **really** going on? `put(x, get(x) + 1)`
 - So **put** is `:=` while **get** is “invisible”!

Reference Types I

ML distinguishes between the “box”^a and the “content”! Picture!

- the “box” and the “content” it holds have **different types**
- `val x = 0; ⇒ val x = 0 : int` value x
- `val y = ref 0; ⇒ val y = ref 0 : int ref` variable y
- `x = 0; ⇒ val it = true : bool`
- `y = 0; ⇒ error` `x = y; ⇒ error`

Operations for working with variables

(get & put)

- `op := i ⇒ val 'a it = fn : 'a ref * 'a -> unit`
- `op ! i ⇒ val 'a it = fn : 'a ref -> 'a`

^aA box is like a cons cell, except it can store only one value.

Reference Types II

Translating `x := x + 1` from imperative languages

- `val x = ref 0; ⇒ val x = ref 0 : int ref`
(declaration `int x := 0`)
- `!x ⇒ val it = 0 : int`
- `x := !x + 1 ⇒ val it = () : unit`
(instruction `x := x + 1`)
- `!x ⇒ val it = 1 : int`

Identifiers vs Variables

Cleaner handling of **variables** than you're used to...

- seems “strange” at first, but you'll get used to it
- in fact, you'll start thinking like this in C as well

`ref` is a **type constructor** for mutable variables

Summary

- `x : int` not assignable (like a constant)
- `x : int ref` assignable reference cell

Imperative Programming

ML also has other imperative constructs, loops for example

- `val i = ref 0; ⇒ val i = ref 0 : int ref`
- `val j = ref 0; ⇒ val j = ref 0 : int ref`
- `while !i < 10 do (i := !i + 1; j := !j + !i)a;`
`⇒ val it = () : unit`
- `!j; ⇒ val it = 55 : int`

For **certain** applications, imperative programming makes sense

- however, it is “ML style” to be “**as functional as possible**”
- functional programs are usually more **concise**, for example

```
fun sum 0 = 0 | sum n = n + sum (n-1);
```

^aNote the parenthesis!

Sequencing

`e1; e2`

equivalent to

`(fn x => e2) e1`

What does this remind you of?

This explains the scoping!

Types

Type

A type is a collection of computable values that share some structural property.

Examples: Integers, Strings, $\text{int} \rightarrow \text{bool}$,
 $(\text{int} \rightarrow \text{int}) \rightarrow \text{bool}$

“Non-examples”: $\{3, \text{true}, \lambda x.x\}$, even integers, $\{f:\text{int} \rightarrow \text{int} \mid \text{if } x>3$
then $f(x) > x*(x+1)\}$

Distinction between types and non-types is language dependent.

Uses for Types

- Program organization and documentation
 - Separate types for separate concepts
 - * Represent concepts from problem domain
 - Indicate intended use of declared identifiers
 - * Types can be checked, unlike program comments
- Identify and prevent errors Compile-time or run-time checking can prevent meaningless computations such as `3 + true` - ÒBillÓ
- Support optimization
 - Example: short integers require fewer bits
 - Access record component by known offset

Type Errors

- Hardware error
 - function call `x()` where `x` is not a function
 - may cause jump to instruction that does not contain a legal op code
- Unintended semantics
 - `int_add(3, 4.5)`
 - not a hardware error, since bit pattern of float 4.5 can be interpreted as an integer
 - just as much an error as `x()` above

General definition of type error

A **type error** occurs when execution of program is not faithful to the intended semantics.

Do you like this definition?

- Store 4.5 in memory as a floating-point number
 - Location contains a particular bit pattern
- To interpret bit pattern, we need to know the type
- If we pass bit pattern to integer addition function, the pattern will be interpreted as an integer pattern
 - Type error if the pattern was intended to represent 4.5

Compile-time vs Run-time Checking

Lisp uses **run-time type checking**

- `(car x)` check first to make sure x is list

ML uses **compile-time type checking**

- $f(x)$ must have $f : A \rightarrow B$ and $x : A$

Basic tradeoff

- Both prevent type errors
- Run-time checking slows down execution
- Compile-time checking restricts program flexibility
 - Lisp list: elements can have different types
 - ML list: all elements must have same type

Expressiveness

- In Lisp, we can write function like

```
(lambda (x)
  (if (< x 10) x
      (car x)))
```

Some uses will produce type error, some will not!

- Static typing always **conservative**

```
if (big-hairy-boolean-expression)
  then ((lambda (x) ... ) 5)
  else ((lambda (x) ... ) 10)
```

Cannot decide at compile time if run-time error will occur!

Relative type-safety of languages

Not safe: BCPL family, including C and C++

- Casts, pointer arithmetic

Almost safe: Algol family, Pascal, Ada.

- Dangling pointers
 - Allocate a pointer p to an integer, deallocate the memory referenced by p , then later use the value pointed to by p
 - No language with explicit deallocation of memory is fully type-safe

Safe: Lisp, ML, Smalltalk, and Java

- Lisp, Smalltalk: dynamically typed
- ML, Java: statically typed