Principles of programming languages

Lecture 4

http://few.vu.nl/~nsilvis/PPL/2007

Natalia Silvis-Cividjian
e-mail: nsilvis@few.vu.nl

vrije Universiteit *amsterdam*

From last week

Q: Short circuit operators: not all languages have them?

A: No.

Java && yes, but not &
Pascal not
Ada can do both

From last week

- What is –lm?
- In general, the compiler option -l*NAME* will attempt to link object files with a library file 'lib*NAME*.a'
- -lm for mathematic library

Outline

- Part I. What are types and what are they used for?
- Part II. Functional programming. Second look at ML.

```
int n;
```

What is a type?

A type is

a ***set of values***

$$n \in \{0, 1, -1, 2, -2.....\}$$

All elements of this set have:

The same low-level **representation**

\+

a collection of **operations** that can be applied to those values

---

Primitive/Constructed Types

- Any type that a program can use but cannot define for itself is a *primitive type* in the language
- Any type that a program can define for itself (using the primitive types) is a *constructed type*

Primitive Types

- The definition of a language says what the primitive types are
- Some languages define the primitive types more strictly than others.
- Example: Java vs. C. **WHY?**

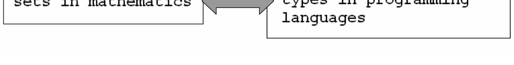
Comparing Integer Types

C:
`char`
`unsigned char`
`short int`
`unsigned short int`
`int`
`unsigned int`
`long int`
`unsigned long int`
No standard implementation,

Java:
`byte` (1-byte signed)
`char` (2-byte unsigned)
`short` (2-byte signed)
`int` (4-byte signed)
`long` (8-byte signed)

Scheme:
`integer`
Integers of unbounded range

Constructed Types


`sets in mathematics` ⟺ `types in programming languages`

Example: ML

- Primitive ML types **`int`**, **`real`**, **`bool`**, **`char`**, and **`string`**
- Type constructors:
  - Tuple types using **`*`**
  - List types using **`list`**
  - Function types using **`->`**

Examples of constructed types

- Enumeration
- Tuples
- Vectors
- Functions

Example 1: Enumeration

Mathematics: $S = \{a, b, c\}$

Programming languages:

C: `enum coin {penny, nickel, dime, quarter};`

Pascal: `type primaryColors = (red, green, blue);`

ML: `datatype day = M | Tu | W | Th | F | Sa | Su;`

Java: `Only from v.5.0!`

```
public enum Season { SPRING, SUMMER, FALL, WINTER };
```

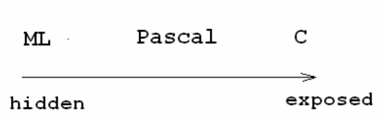
How are enumeration elements represented?

Representation

- A common representation: as integers:

penny = 0; nickel = 1; dime = 2

- Representation can be exposed or hidden to the programmer



Operations

- Depend on how much of the representation is exposed to the programmer
- **ML:** only equality test possible

```
fun isWeekend x = (x = Sa orelse x = Su);
```

- **Pascal:**

```
for c:=red to blue do fred(c);
```

- **C:**
  - `enum coin { penny = 1, nickel = 5, dime = 10, quarter = 25 };`

Example 2: Tuples

- Mathematics: $S = X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$
- ML: type irpair=int*real ;
- **C records**
- **struct complex {**

```
struct complex {
  double rp;
  double ip;
};
```

Tuples

- Some languages support pure tuples:

```
fun get1 (x : real * real) = #1 x;
```

- Many others support **record** types, which are just tuples with named fields:

ML:

```
type complex = {
  rp:real,
  ip:real
};
fun getip (x : complex) = #ip x;
```

Representation

- A common representation : place the elements side-by-side in memory
- But there are lots of details:
  - in what order?
  - with “holes” to align elements (e.g. on word boundaries) in memory?
  - is any or all of this visible to the programmer?

Operations

- Selection:

C: `x.ip`

ML: `#ip x`

- Other operations depending on how much of the representation is exposed:

C:

```
double y = *((double *) &x);
struct person {
  char *firstname;
  char *lastname;
} p1 = {"marcia","brady"};
```

Example 3: Arrays, strings and lists

- Fixed-size vectors:

$$S = X^n = \{(x_1, \ldots, x_n) \mid \forall i \,.\, x_i \in X\}$$

- Arbitrary-size vectors:

$$S = X^* = \bigcup_i X^i$$

Typical issues

- What are the index values?
- Is array size fixed at compile time (part of static type)?
- What operations are supported?
- Is redimensioning possible at runtime?
- Are multiple dimensions allowed?
- Is a higher-dimensional array the same as an array of arrays?
- What is the order of elements in memory?
- Is there a separate type for strings (not just array of characters)?
- Is there a separate type for lists?

Index issue

- Java, C, C++:
  - First element of an array **`a`** is **`a[0]`**
  - Indexes are always integers starting from 0
- Pascal is more flexible:
  - Various index types are possible: integers, characters, enumerations, subranges
  - Starting index chosen by the programmer
  - Ending index too: size is fixed at compile time

A Pascal example

```
type
  LetterCount =
array['a'..'z'] of Integer;
var
  Counts: LetterCount;
begin
  Counts['a'] = 1
  etc.
```

Outline

- A Type Menagerie
  - Primitive types
  - Constructed types
- **Where are types used?**
  - **Type annotations and type inference**
  - **Type equivalence issues**
  - **Type checking**

Type Annotations

- Many languages require, or at least allow, type annotations on variables, functions, …
- The programmer uses them to supply static type information to the language system
- They are also a form of documentation, and make programs easier for people to read

---

Explicit type annotations

```
- fun prod(a,b) = a * b;
val prod = fn : int * int -> int
```

Why decides ML that the type is **`int`**, rather than **`real`**?

ML's *default type* for **`*`** (and **`+`**, and **`–`**) is
`int * int -> int`

You can give an explicit *type annotation* to get **`real`** instead by using type annotations

---

Type annotations in ML

```
- fun prod(a:real,b:real):real = a*b;
val prod = fn : real * real -> real
```

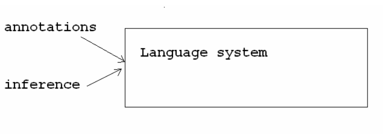
Type annotation is a colon followed by a type and can appear after any variable or expression

---

Intrinsic Types

- Some languages use naming conventions to declare the types of variables
  - Dialects of BASIC: **`S$`** is a string
  - Dialects of Fortran: **`I`** is an integer
- Like explicit annotations, these supply static type information to the language system and the human reader

- Annotations are made by programmer to help the language system to get type information
- The system itself makes inference



Simple Type Inference

- Most languages require some simple kinds of type inference
- Constants usually have types
  - Java: **`10`** has type **`int`**, **`10L`** has type **`long`**
- Expressions may have types, inferred from operators and types of operands
  - Java: if **`a`** is **`double`**, **`a*0`** is **`double`** (**`0.0`**)

Extreme Type Inference

- ML takes type inference to extremes
- Infers a static type for every expression and for every function
- Usually requires no annotations

Type Equivalence

- When are two types the same?
- An important question for static and dynamic type checking
- For instance, a language might permit **`a:=b`** if **`b`** has "the same" type as **`a`**
- Different languages decide type equivalence in different ways

Type Equivalence

- **Name equivalence:** types are the same if and only if they have the same name
- **Structural equivalence:** types are the same if and only if they are built from the same primitive types using the same type constructors in the same order
- Languages often use odd variations or combinations

Type Equivalence Example

```
type irpair1 = int * real;
type irpair2 = int * real;
fun f(x:irpair1) = #1 x;
```

- What happens if you try to pass **`f`** a parameter of type **`irpair2`**?
  - Name equivalence does not permit this: **`irpair2`** and **`irpair1`** are different names
  - Structural equivalence does permit this, since the types are constructed identically
- ML does permit it

Type Equivalence Example

```
var
  Counts1: array['a'..'z'] of Integer;
  Counts2: array['a'..'z'] of Integer;
```

- What happens if you try to assign **`Counts1`** to **`Counts2`**?
  - Name equivalence does not permit this: the types of **`Counts1`** and **`Counts2`** are unnamed
  - Structural equivalence does permit this, since the types are constructed identically
- Most Pascal systems do not permit it

- What is type checking?

Static Type Checking

- **Static** type checking determines a type for everything **before** running the program: variables, functions, expressions, etc.
- Compile-time error messages when static types are not consistent
  - Operators: **`1+"abc"`**
  - Functions: **`round("abc")`**
  - Statements: **`if "abc" then …`**
- Most modern languages are statically typed: ML. Java

Dynamic Type Checking

- In some languages, programs are type-checked **at runtime** = **dynamically** typing
- At runtime, the language system checks that operands are of suitable types for operators
- Not quite a black-and-white picture

Example: Lisp

- This Lisp function adds two numbers:

```
(defun f (a b) (+ a b))
```

- It won't work if **`a`** or **`b`** is not a number
- An improper call, like **`(f nil nil)`**, is not caught at compile time
- It is caught at runtime – that is dynamic typing

- What is the goal of type checking?

Strong Typing, Weak Typing

- The purpose of type-checking is to prevent the application of operations to incorrect types of operands
- In some languages, like ML and Java, the type-checking is thorough enough **to guarantee** this—that's **strong typing**
- Many languages (like C, Python) fall short of this: there are holes in the type system that add flexibility but weaken the guarantee- **weak typing**

Productivity vs. performance

Trade-off: Strong vs. weak typing

Let's discuss two articles:

- *Strong versus Weak Typing,*

an interview with Guido van Rossum, creator of Python, www.artima.com/intv/strongweakP.html

- *James Gosling on Java,*

a conversation with Java's creator, James Gosling, www.artima.com/intv/gosling319.html

---

Trade-off: Strong vs. weak typing

"There is a folk theorem **out there** that system with very loose typing are very easy to build prototypes with. That may be true. But the leap from a prototype that way to a real industrial-strength system is pretty vast. *James Gosling (Java creator and a strong typist)*

Funny questions

- "To what extend do you think the choice between using a strongly or weakly typed language has to do with personality?"
- "Speaking of spacecraft, would you be comfortable with the robustness of Python systems to fly an airplane in which all the control software was written in Python? "

Conclusion

- A key question for type systems: how much of the representation is exposed?
- Some programmers prefer languages like C that **expose** many implementation details
  - They offer the power to cut through type abstractions, when it is useful or efficient or fun to do so
- Others prefer languages like ML that **hide** all implementation details (*abstract types*)
  - Clean, mathematical interfaces make it easier to write correct programs, and to prove them correct

---

Part II

Functional programming. Second look to ML.

Outline

- Patterns
- Local variable definitions

Two Patterns You Already Know

```
fun f n = n*n;
fun f (a, b) = a*b;
```

- Both **n** and **`(a,b)`** are **patterns.**

`n` **:** matches to everything, creates a variable n and binds it to this everything

`(a,b)`: matches any 2-tuple, creates 2 variables a and b, and binds **a** and **b** to this tuple components

---

Other patterns: Underscore

```
- fun f _ = "yes";
val f = fn : 'a -> string
- f 34.5;
val it = "yes" : string
- f [];
val it = "yes" : string
```

It matches anything, but does not bind it to a variable

Preferred to:

`fun f x = "yes";`

`Why?`

Other patterns: Constants

```
- fun f 0 = "yes";
Warning: match nonexhaustive
         0 => ...
val f = fn : int -> string
- f 0;
val it = "yes" : string
```

- Any constant of an equality type can be used as a pattern
- But not:

`fun f 0.0 = "yes";`

Non-Exhaustive Match

- In that last example, the type of **`f`** was **`int -> string`**, but with a “match non-exhaustive” warning
- Meaning: **`f`** was defined using a pattern that didn’t cover all the domain type (**`int`**)
- So you may get runtime errors like this:

```
- f 0;
val it = "yes" : string
- f 1;
uncaught exception nonexhaustive match failure
```

Other patterns: Lists Of Patterns

```
- fun f [a,_] = a;
Warning: match nonexhaustive
          a :: _ :: nil => ...
val f = fn : 'a list -> 'a
- f [#"f",#"g"];
val it = #"f" : char
```

- This example matches any list of length 2
- It treats **a** and _ as sub-patterns, binding **a** to the first list element

Other patterns: Cons Of Patterns

```
- fun f (x::xs) = x;
Warning: match nonexhaustive
          x :: xs => ...
val f = fn : 'a list -> 'a
- f [1,2,3];
val it = 1 : int
```

- You can use a cons of patterns as a pattern
- `x::xs` matches any non-empty list, and binds **x** to the head and **xs** to the tail

ML Patterns So Far

- A variable is a pattern that matches anything, and binds to it
- A _ is a pattern that matches anything
- A constant (of an equality type) is a pattern that matches only that constant
- A tuple of patterns is a pattern that matches any tuple of the right size, whose contents match the sub-patterns
- A list of patterns is a pattern that matches any list of the right size, whose contents match the sub-patterns
- A cons (`::`) of patterns is a pattern that matches any non-empty list whose head and tail match the sub-patterns

Multiple Patterns for Functions

```
- fun f 0 = "zero"
  |   f 1 = "one";
Warning: match nonexhaustive
          0 => ...
          1 => ...
val f = fn : int -> string;
- f 1;
val it = "one" : string
```

- You can define a function by listing alternate patterns

Overlapping Patterns

```
- fun f 0 = "zero"
= |   f _ = "non-zero";
val f = fn : int -> string;
- f 0;
val it = "zero" : string
- f 34;
val it = "non-zero" : string
```

- Patterns may overlap
- ML uses the **first match** for a given argument

Pattern-Matching Style

- These definitions are equivalent:

```
fun f 0 = "zero"
|   f _ = "non-zero";
```

-

```
fun f n =
  if n = 0 then "zero"
  else "non-zero";
```

- But the pattern-matching style usually preferred in ML, because it often gives shorter and more readable code.

Pattern-Matching Example

```
fun fact n =
  if n = 0 then 1 else n * fact(n-1);
```

Rewritten using patterns:

```
fun fact 0 = 1
|   fact n = n * fact(n-1);
```

More Examples

This structure occurs frequently in recursive functions that operate on lists: one alternative for the base case (`nil`) and one alternative for the recursive case (`first::rest`).

Adding up all the elements of a list:

```
fun f nil = 0
|   f (first::rest) = first + f rest;
```

Counting the true values in a list:

```
fun f nil = 0
|   f (true::rest) = 1 + f rest
|   f (false::rest) = f rest;
```

More Examples

Making a new list of integers in which each is one greater than in the original list:

```
fun f nil = nil
|   f (first::rest) = first+1 :: f rest;
```

A Restriction

- You can't use the same variable more than once in the same pattern
- This is not legal:

```
fun f (a,a) = … for pairs of equal elements
|   f (a,b) = … for pairs of unequal elements
```

- You must use this instead:

```
fun f (a,b) =
  if (a=b) then … for pairs of equal elements
  else … for pairs of unequal elements
```

Patterns are Everywhere

```
- val (a,b) = (1,2.3);
val a = 1 : int
val b = 2.3 : real
- val a::b = [1,2,3,4,5];
Warning: binding not exhaustive
          a :: b = ...
val a = 1 : int
val b = [2,3,4,5] : int list
```

- Patterns are not just for function definition
- Here we see that you can use them in a `val`

Outline

- Patterns
- Local variable definitions

Local Variable Definitions: let

- When you use `val` at the top level to define a variable, it is visible from that point forward
- There is a way **to restrict** the scope of definitions: the `let` expression

<let-exp> `::=` **`let`** *<definitions>* **`in`** *<expression>* **`end`**

Example with `let`

```
- let val x = 1 val y = 2 in x+y end;
val it = 3 : int;
- x;
Error: unbound variable or constructor: x
```

- The value of a `let` expression is the value of the expression in the `in` part
- Variables defined with `val` between the `let` and the `in` are visible only from the point of declaration up to the `end`

Proper Indentation for `let`

```
let
  val x = 1
  val y = 2
in
  x+y
end
```

- For readability, use multiple lines and indent `let` expressions like this
- Some ML programmers put a semicolon after each `val` declaration in a `let`

Long Expressions with `let`

```
fun days2ms days =
  let
    val hours = days * 24.0
    val minutes = hours * 60.0
    val seconds = minutes * 60.0
  in
    seconds * 1000.0
  end;
```

- The `let` expression allows you to break up long expressions and name the pieces
- This can make code more readable

Example: Patterns with `let`

```
fun halve nil = (nil, nil)
|   halve [a] = ([a], nil)
|   halve (a::b::cs) =
      let
        val (x, y) = halve cs
      in
        (a::x, b::y)
      end;
```

This example takes a list argument and returns a pair of lists, with half in each

`halve` At Work

```
- fun halve nil = (nil, nil)
= |   halve [a] = ([a], nil)
= |   halve (a::b::cs) =
=       let
=         val (x, y) = halve cs
=       in
=         (a::x, b::y)
=       end;
val halve = fn : 'a list -> 'a list * 'a list
- halve [1];
val it = ([1],[]) : int list * int list
- halve [1,2];
val it = ([1],[2]) : int list * int list
- halve [1,2,3,4,5,6];
val it = ([1,3,5],[2,4,6]) : int list * int list
```

Summary

- What are types and what are they used for
- Trade-off strong vs. weak typing
- ML: pattern matching