Functional Programming

Another representative from the *Declarative* paradigm

see Chapter Nine

Variables in Imperative Languages

- A variable in an imperative programming language can be regarded as an abstraction of the von Neumann computer store location.

- Assignment is the means by which values are changed.

- This influences the whole paradigm. In particular:
    1. Issues around assignment.
    2. Control of execution of sequences of instructions.

Assignment

- There are problems with assignment. These relate to
  - Side-effects (covered earlier)
  - Aliasing (covered earlier)
  - Referential transparency

Referential Transparency

- Referential transparency means
  - The meaning (or the effect, or the value) of an expression is the same wherever (and whenever) it occurs.
- *Imperative languages do not support referential transparency.*
- Trivial example:

```
X := 0;
Write(X);
X := X + 1;
Write(X);
```

- This problem becomes even worse when parallel execution is considered because the order of execution matters.

Sequences of Instructions

- Imperative programs are sequences of instructions because to be executed they must be stored in memory in a sequence of locations, and because of the way the standard fetch/execute cycle works.

- The control of such execution, involving loops of different kinds, is a common source of bugs in imperative programs.

Summary

- Imperative languages are based on standard computer architecture.
- The advantage of this is run-time efficiency.

- BUT…

1. It facilitates errors.
2. It makes bug-tracing difficult.
3. It makes programs hard to read and understand.
4. (Backus, 1978) It limits the programmer's thinking and inhibits the problem-solving process.

The Functional Approach

- Here are two examples of function definitions:

  Double(n) = n+n;

  Square(n) = n*n;

- A `functional program' is simply a collection of function definitions.
- A functional programming system allows us to store such function definitions, and then ask for specific expressions to be evaluated.
- For example, if we had stored the definitions above, we could give

  Double(7);

  to the system, and it would respond by giving the value of the expression, namely 14.

Functional Programming

- Note that a functional language is inherently declarative.

- Such function definitions are simply statements of what our functions are.

- There is an analogy with Prolog. But remember that in Prolog we dealt with predicates, not functions, and we achieve goals rather than evaluate expressions.

The Functional Approach

- A program consists of a collection of function definitions, and a `run' amounts to an evaluation of an expression involving these functions.
- The intention is to focus on the computations which are to be done, not on how to fit them to the structure of the machine.
- Assignment is not a part of this paradigm, nor is the idea of a sequence of instructions.
- Under this paradigm:
    1. Side-effects can be completely avoided.
    2. Aliasing is not possible.
    3. Referential transparency can be supported.
- Examples of functional languages: LISP, ML, Miranda, Haskell, Erlang

FP in the Real World

- from homepages.inf.ed.ac.uk/wadler/realworld/
- Industrial
  - Erlang, an e-commerce database, Partial evaluator for airline scheduling, Combinators for financial derivatives
- Compilers, Interpreters and Partial Evaluators, Syntax and Analysis Tools
- Theorem Provers and Reasoning Assistants
- Network Toolkits and Applications, Web, HTML, XML, XSLT
- Natural Language Processing and Speech Recognition
- Numerically Based Applications, Computer Algebra
  - MC-SYM - computes 3D shape of nucleic acid, FFTW - Fastest Fourier Transform in the West, BlurFit - model focal plane imaging
- Database Systems
- Operating Systems
- Light and sound
  - Lula: a system for theater lighting, Scheme Score

1. Programs

- A program consists of a collection of function definitions:
- For Example

```
fun Double(n)        = n+n;
fun Square(n)        = n*n;
fun Avg(x,y)         = (x+y)/2;
fun SqAvg(x,y) = Avg(Square(x),Square(y));
```

- The the following can be evaluated:

```
Square(Double(6));
SqAvg(5,7);
```

- In each case the system would respond with the value
  - (respectively 144 and 37).

2. Data Structures

- Lists are the main basic data structure.
- Notations (ML):

```
[2,6,4,5]
h :: t
nil
hd(s)
tl(s)
```

- Example function involving lists (more later):

```
fun Length(nil) = 0
  | Length(h :: t) = 1 + Length(t);
```

3. Program Control

- We have **composition of functions**:
  - apply one function to some value(s), and then apply another function to the result. (Like SqAvg above).
- There is no notion of program loop, so recursion is essential (see the Length function above).
- Functional programming also has an `if' construct, to distinguish cases.
- Example:

```
fun Max(x,y) =
    if x >= y then x else y;
fun IsEmpty(l) =
    if l = nil then true else false;
```

4. Pattern-Matching

- Two equivalent definitions:

```
fun IsEmpty(l) =
        if l = nil then true else false;

fun IsEmpty(nil) = true
  | IsEmpty(h :: t) = false;
```

- When we ask to evaluate (say) IsEmpty([4,1,7]), a matching process takes place, and as a result the second line of the definition is used.

- (See also the Length function above.)

More Pattern Matching

```
fun IsMember(x,l) = if l = nil then false
                    else if x = hd(l) then true
                    else IsMember(x,tl(l));
```

is equivalent to

```
fun IsMember(x,nil)      = false
| IsMember(x,x :: t)     = true
| IsMember(x,h :: t)     = IsMember(x,t);
```

- The word pattern here refers to the formal structure of an expression (nil and h :: t are patterns). Pattern-matching is an essential part of functional languages. (As it is also with Prolog.)

Further Pattern Matching Examples

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

fun Factorial(0) = 1
    | Factorial(n) = n * Factorial(n - 1);

fun ListAsFarAs(x,nil) = nil
    | ListAsFarAs(x,x :: t) = [x]
    |  ListAsFarAs(x,h :: t) = h :: ListAsFarAs(x,t);

fun DoubleList(nil) = nil
    | DoubleList(h :: t) = Double(h) :: DoubleList(t);

fun SumList([x]) = x
    |  SumList(h :: t) = h + SumList(t);

fun ListAvg(l) = SumList(l)/Length(l);
```

5. Evaluation

- A functional programming system simply allows evaluation of expressions. An actual evaluation is done by a process called reduction or rewriting. Here is how it works.
- Say we wish to evaluate ListAvg([3,7,2]).

ListAvg([3,7,2])

SumList([3,7,2]) / Length([3,7,2])

(3 + SumList([7,2])) / (1 + Length([7,2]))

(3 + (7 + SumList([2]))) / (1 + (1 + Length([2])))

(3 + (7 + 2)) / (1 + (1 + (1 + Length(nil))))

(3 + 9) / (1 + (1 + (1 + 0)))

12 / (1 + (1 + 1))

12 / (1 + 2)

12 / 3

4

6. Higher-order Functions

- In functional languages there is the possibility to define functions which have functions as parameters, or which return functions as results. This is what is meant by higher-order functions.
- Avoids repetition of code.
- E.g. We have a function to traverse a list and add all the elements up. We also have a function to traverse a list and multiply all the elements together. Why not have a function to traverse a list and apply some action, and make the action (function) a parameter?

- This is a natural feature of functional languages.
- Functions can have functions as parameters.

6. Higher-Order Functions (continued)

- Example

```
fun DoubleList(nil) = nil
  | DoubleList(h :: t) = Double(h) :: DoubleList(t);
fun SquareList(nil) = nil
  | SquareList(h :: t) = Square(h) :: SquareList(t);
```

- Similarities can be drawn out:

```
fun ListApply(f,nil) = nil
  | ListApply(f,h :: t) = f(h) :: ListApply(f,t);
```

- Examples become:

```
ListApply(Double,[3,1,4])
ListApply(Square,[3,1,4])
```

7. Types

(The simplistic story)

Static Types:

- All variables and parameters must have their types declared in the program, so that they can be checked by the compiler.

Dynamic Types:

- The types of program entities are not constrained at all in advance of run-time information.

Best of Both?

- Security of compile-time type checking
- Flexibility of dynamic typing / freedom from type declarations
- Make the system infer types itself!

8. Strong Typing and Type Inference

- Strong typing: all expressions have a well defined type.
- Most functional languages other than LISP have strong typing. They have a type-checking system which infers types for all objects for which a type is not specified, and checks for inconsistencies.
- It seems to give the best of both worlds:
  - security against errors, provided by type-checking,
  - freedom from the necessity to specify types.

Type Inference

- Example:

  ```
  fun Sum1To(n) = n*(n+1) div 2;
  ```

- n must be such that +1 and * and div are appropriate operations
- Sum1To must return a corresponding numerical result
- The function Sum1To thus must have type int -> int

Type Inference (continued)

- Here is another example:

```
fun AddToList(a,nil) = nil
  | AddToList(a,h :: t) =
          if (h < 0) then (a + h) :: AddToList(a,t)
          else h :: AddToList(a,t);
```

- The type of AddToList is int * (int list) -> (int list)

Type Inference (continued)

- Another example

  fun IsEmpty(l) = if l=nil then true else false;

- It is clear that this function acts on a list and returns a Boolean result. *(why?)*

- What kind of list?
- ...Any kind of list.

- *It is polymorphic . . .*

9. Polymorphism

```
fun Length(nil) = 0
  | Length(h :: t) = 1 + Length(t);
```

- The type of h may be anything. We could use this function to find the length of a list of numbers, a list of Booleans, a list of strings, a list of lists, ...
- And the value returned will be an integer (because ?).
- The type of Length would be inferred as

  ('a list) -> int

- ML uses the notation 'a to represent types which are unconstrained in this way.
  - The type of IsEmpty above is ('a list) -> bool

10. Lazy Evaluation

- Simply, this means that parameters are not evaluated until they are required.
- The opposite of lazy is strict.
- Of functional languages, LISP and ML are strict, whereas Miranda and Haskell are lazy.
- Laziness has an efficiency advantage: parameter values which are not in fact needed will not be evaluated.

Lazy Evaluation

- But laziness also brings other possibilities:

  fun CountFrom(n) = n :: CountFrom(n+1)

- An infinite list?

- Consider

  SumList(ListAsFarAs(10,CountFrom(1)))

What is used in the evaluation?

11. Abstract Data Types

- The only means of building data structures so far has been the list.
- Most functional programming languages include a facility for the user to build tailor-made data types, specific to current needs, using abstract data types.
- Example (ML):

```
datatype 'a tree = empty
                   |node of ('a * ('a tree) * ('a tree));
```

- This is a definition of a binary tree type. Here, 'a is a type variable (and the * separates types).

Abstract Data Types (continued)

- Values of the type (int tree) would be expressions such as

  empty

  node(4,empty,empty)

  node(4,empty,node(6,empty,empty))
- A binary tree either is empty, or consists of a root (at which is stored a data item) and two subtrees.
- That is exactly what we are constructing here.
- Note that the only mechanism used to build this structure is the (formal) application of functions.

Abstract Data Types (continued)

- Example (ML):

```
        datatype 'a stack = new_stack
          | push of ('a * ('a stack));
```

- Values are expressions like

```
    new_stack
    push(3,new_stack)
    push(7,push(3,new_stack))
```

- Standard operations include:

```
    exception top_err:
    fun top(new_stack) = raise top_err;
      |  top(push(a,b)) = a;
```

- (Note the mechanism for dealing with error situations.)

see page 238

Summary

- Here are the aspects of functional languages that we have considered:
    1. Composition of functions.
    2. Data structures (lists).
    3. Distinguishing cases using if.
    4. Pattern matching.
    5. Evaluation by reduction/rewriting.
    6. Static/dynamic typing.
    7. Higher-order functions.
    8. Strong typing and type inference.
    9. Polymorphism.
    10. Lazy evaluation.
    11. Data structures (abstract data types).
- And finally ...

Declarative Programming

- Both functional languages and logic languages are said to be declarative.
- Declarative programs consist of assertions and definitions, not instructions or commands.
- A consequence of this is the meaning of a declarative program should be independent of whatever run-time system is used to execute it.

Declarative Programming

- Specifically:
    1. A functional program has meaning in a mathematical sense. The forms and symbols used have precise mathematical meanings.
    2. A logic program has meaning in terms of the logical interpretation of the symbols. (Although in the case of Prolog this is somewhat compromised.)

- There are perhaps two significant benefits which derive from this:
    1. Mathematically precise meanings facilitate reasoning about programs (for example in relation to correctness).
    2. The independence from the machine makes such languages closer to specification languages, and so they lessen the gap (which is a cause of bugs) between specification and implementation.