Lecture 9: Functional Languages (Part I)

Preliminaries

The focus of this part of the course will be on **concepts** in declarative programming languages.

Theses concepts can be applied in almost any programming language, but they are most easily explained in declarative languages.

Examples for concepts that you know from imperative languages are loops or procedures.

Web page on programming languages:
`http://www.cee.hw.ac.uk/~hwloidl/prg-lang.html`

Installations of compilers/interpreters:
`/net/azdak/dsg/local/i386-unknown-linux/bin`

Introduction

The philosophy of *declarative languages* is

> Specify **what** should be computed, **not how** it is computed.

While imperative languages start from machine languages and develop abstractions over basic control structures, declarative languages start from a mathematical model and turn it into a concrete language.

Declarative languages are motivated by mathematical models for describing functions and relationships:

- the λ-calculus describes functions, and is the basis for **functional languages**;
- the predicate-calculus describes relationships, and is the basis for **logic languages**.

Abstracting over Machine Characteristics

Different kinds of languages provide different levels of abstraction over machine characteristics.

The programmer can choose the appropriate level of abstraction:

- Declarative Languages
- Imperative Languages
- Assembler Languages

Example program: Construct a list of square numbers from 1 to 10.

Example: Declarative Language

In a **declarative language** programs are specified as mathematical functions or relations. E.g. in Haskell (a functional language) we can write

```
[ x^2 | x <- [1..10] ]
```

This program describes only the *value* that should be computed as a function of its parameters (*functional language*).

Example: Imperative Language

In an **imperative language** programs are specified as sequences of operations. E.g. in C we can write

```
register int i;
int sqs[10];

for (i=1; i<=10; i++)
  sqs[i-1] = i*i;
```

This program describes the *procedure* of computing a value (*procedural language*).

Example: Assembler Language

In an **assembler language** programs are specified as machine-specific arithmetic and jump operations. E.g. for an i386 compatible processor

```
.L2:    cmpl $10,%ebx
        jle .L5
        jmp .L3
.L5:    leal -1(%ebx),%edx
        leal 0(,%edx,4),%eax
        leal -40(%ebp),%edx
        movl %ebx,%ecx
        imull %ebx,%ecx
        movl %ecx,(%eax,%edx)
.L4:    incl %ebx
        jmp .L2
.L3:    ...
```


Abstraction in Programming Languages

What is “abstracted”?

From Assembler to C:

- Jumps are abstracted into a set of control structures.
- Different kinds of memory access are abstracted into variable assignment.
- Machine-specific arithmetic operations (`imull`) are abstracted into familiar arithmetic operations (`*`).

From C to Haskell:

- Loop indices are abstracted into iterators over e.g. lists.
- No explicit dynamic memory allocation.
- No difference between register and normal variables.

The Church-Rosser property

The characterising property of declarative languages is the so called **Church-Rosser property**.

Intuitively, the Church-Rosser property says that the order of evaluation does not matter.

The possibility of replacing one expression with another expression that produces the same value, i.e. "replacing equals for equals", is also called **referential transparency**.

Advantages and Disadvantages of Declarative Languages

Advantages of declarative languages:

- Reasoning about correctness programs is easy.
- Programs tend to be much shorter than in imperative languages.
- Easy to do rapid prototyping.
- No inherent sequential execution.

Disadvantages of declarative languages:

- Reasoning about efficiency of programs can be hard.
- Fewer development tools available.

The Usage of Functional Languages in the Real World

Source code in Red Hat Linux 7.1[a]:

Language	Source lines of Code SLOC (in %)
C	21461450 (71.18%)
C++	4575907 (15.18%)
Shell (Bourne-like)	793238 (2.63%)
Lisp	722430 (2.40%)
Assembly	565536 (1.88%)
Perl	562900 (1.87%)
Fortran	493297 (1.64%)
Python	285050 (0.95%)
Tcl	213014 (0.71%)
Java	147285 (0.49%)

[a] From an article on slashdot (`http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`)

Books

The following are good textbooks on the declarative languages discussed in this course.

- **Haskell:** "Haskell: The Craft of Functional Programming", by Simon Thompson, Second Edition, Addison-Wesley, paperback, 1999. ISBN 0-201-34275-8.
- **LISP:** "Common Lisp the Language", 2nd edition, by Guy L. Steele, Thinking Machines, Inc. Digital Press. URL: `http://www.ida.liu.se/imported/cltl/cltl2.html`
- **SML:** "Elementary Standard ML", by Greg Michaelson, UCL Press, 1995. ISBN 1-85728-398-8 PB

Software

The following compilers and interpreters are available on the Linux lab machines.

Glasgow Haskell Compiler — interactive version:
`/net/azdak/dsg/local/i386-unknown-linux/bin/ghci`

Glasgow Haskell Compiler — documentation:
`http://haskell.cs.yale.edu/ghc/overview.html`

Haskell Home Page (including links to tutorials and textbooks):
`http://www.haskell.org/`

GNU Common LISP: `/net/azdak/dsg/local/i386-unknown-linux/bin/clisp`

LISP tutorial: `/net/azdak/dsg/local/doc/clisp/doc/LISP-tutorial.txt`

SML/NJ Compiler/Interpreter:
`http://cm.bell-labs.com/cm/cs/what/smlnj/index.html`

Haskell

Elements of Haskell

Haskell is a general purpose, purely functional, non-strict programming language.

The most important concepts shown in Haskell are:

- *Purely functional* (i.e. no side-effects),
- *algebraic data structures* with powerful operations such as pattern matching, list comprehensions,
- *higher-order functions*,
- *strong typing*,
- *lazy evaluation*;

Overall Structure of Haskell Programs

A Haskell program is a set of function definitions, partitioned into modules, with a `main` function defining the value of the program.

Modules help to structure a large application. The interface of a module is defined by the names and *types* of the function it exports.

Haskell provides *type inference*, i.e. the compiler can automatically derive the type of a function from the types of the primitives used in the function.

I. Pattern Matching and Algebraic Data Types

Function definitions often use *pattern matching*, to avoid using explicit functions to extract components of a data structure.

For example, the following function computes the square for every element of a given list (`:` is list construction, i.e. `::` in SML).

```
sqs []     = []
sqs (x:xs) = x*x : sqs xs
```

is the same as

```
sqs xs' = if (null xs') then []
                        else let
                               x  = head xs'
                               xs = tail xs'
                              in
                              x*x : sqs xs
```


Guarded Patterns

Guarded patterns allow to specify a condition that must hold for one case to match.

For example, count the number of integers greater than 0:

```
more0 [] = 0
more0 (x:xs) | x>0 = 1+more0 xs
             | otherwise = more0 xs
```

List comprehensions in Haskell

List comprehensions[a] resemble the usual mathematical notation for sets. They define an expression that constructs the elements of a list, together with iterators, and conditions.

For example, to compute the squares of all even numbers in a given list:

```
sq_even xs = [x*x | x <- xs, even x]
```

The expression `x*x` is the body of the list comprehension.

The `<-` construct is a generator, and acts like an iterator.

The condition `even x` decides whether `x` should be used in the body.

[a] Originally developed as ZF-comprehensions in Miranda

Algebraic Data Types in Haskell

Haskell provides powerful mechanisms for data abstraction.

Enumerations are defined like this:

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Pattern matching over this new data type is very useful:

```
next Mon = Tue
next Tue = Wed
next Wed = Thu
next Thu = Fri
next Fri = Sat
next Sat = Sun
next Sun = Mon
```

Pattern matching over Algebraic Data Types

Algebraic data types can have type variables as parameters.

Lists over any type, represented by the type variable `alpha`, can be defined as an algebraic data type:

```
data MyList alpha = Nil | alpha :^ (MyList alpha)
```

Parametric polymorphism means that you can define functions that work for any instance of `alpha`:

```
my_length :: MyList alpha -> Int
my_length Nil      = 0
my_length (x:^xs) = 1+(my_length xs)
```

For example:

```
> my_length (1:^(2:^Nil))
2
```

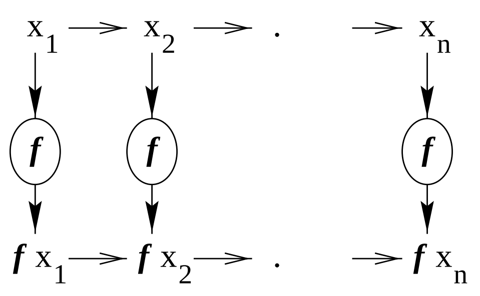

II. Higher-order Functions

Higher-order functions take functions as arguments or yield a function as result.

In general, functions are *first-class values*, i.e. they can be used wherever plain data can be used, provided the type matches.

Higher-order functions are a powerful mechanism for describing *common structures of computation*.

A predefined higher-order function is `map`, which applies a function `f` to all elements of a given list `[x1,x2,...,xn]`:



An Example of Higher-order Functions

With the pre-defined `map` function we can write our function to compute the squares of all even numbers in a different style:

```
sq_even xs = map square xs
             where square x = x*x
```

Note, that this definition separates the function operating on every list element (`square`) from the way the list is traversed (`map`).

Lambda Expressions

In the previous example the `square` function was used only once, hidden inside the definition of `sq`. In such a case it is more convenient not to use a local definition at all, but to define the function as a *lambda expression*.

A lambda expression specifies how to compute the result of a function from its input just like normal function definition, but it does not give a name to the function (*anonymous function*).

```
sqs xs = map (\ x -> x*x) xs
```

In fact, we can write it even shorter:

```
sqs = map (\ x -> x*x)
```

Function Composition

Functions are first-class objects, and therefore we can combine functions just as we can combine data.

The most powerful combinator for functions is function composition, which behaves like function composition in mathematics, i.e. `(f . g) x` is the same as `f (g x)`.

As an example, the function for computing the squares of all even numbers in a list can be written like this:

```
sq_even = map (\ x -> x*x) . filter even
```

`filter p xs` returns all elements in the list `xs` for which `p` is true.

III. Strong Typing

Haskell has a strong type system i.e. all types must be checked at compile time.

Advantages of strong typing:

- Types are a "sanity check" for the correctness of the program "*type correct programs can't go wrong*".
- With type information the compiler can perform more program optimisations "*compilation by transformation*".

Disadvantages of strong typing:

- Some programs can't be expressed in the language.
- Type inference and checking are fairly expensive.
- Error messages are sometimed obscure.

IV. Lazy Evaluation in Haskell

When evaluating a function call, Haskell uses lazy evaluation: *the arguments to the function are evaluated when they are needed*, e.g. by an arithmetic or comparison operator, not necessarily before evaluating the body of the function.

For example consider the following function:

```
foo x y z = if x<0 then abs x
                   else x+y
```

When we call this function we immediately start to evaluate the function body. In the head of the conditional we need the value of `x`, and at this point `x` is evaluated.

If `x<0` is true we return `abs x` and neither `y` nor `z` will be evaluated.

If `x<0` is false, we return `x+y`, which starts the evaluation of `y` to return the result (`z` won't be evaluated).

Some Haskell Functions over Lists

The Haskell library function `!!` extracts the n-th element from a list. Note that `!!` is an *infix* function and that counting starts from 0.

Here are the definitions of `!!` and `[_..]` from the prelude:

```
(x:_)  !! 0            =  x
(_:xs) !! n | n > 0    =  xs !! (n-1)

-- non-valid Haskell syntax
[m .. n] | m<=n       = m:[m+1 .. n]
         | otherwise  = []
```

The Haskell library function for producing a list from `m` to `n` is `enumFromTo`.

The construct `[m..n]` is "syntactic sugar" for this function.

Infinite Data Structures

With these definitions we can write:

```
> [1..10]!!2
3
```

With lazy evaluation we can use *infinite data structures*, because only as much of the data structure will be evaluated as is needed to return the result. For example

```
> [1..]!!2
3
```


An Example of Lazy Evaluation

Here is a trace of the evaluation of `[1..]!!2`:

`[1..] !! 2`	$\Rightarrow$	*is the list empty?*
`(1:[2..]) !! 2`	$\Rightarrow$	*is the index 0?*
`[2..] !! 1`	$\Rightarrow$	*is the list empty?*
`(2:[3..]) !! 1`	$\Rightarrow$	*is the index 0?*
`[3..] !! 0`	$\Rightarrow$	*is the list empty?*
`(3:[4..]) !! 0`	$\Rightarrow$	*is the index 0?*

`3`

Advanced Topics

The following advanced topics are not discussed in detail in this course:

- Modules (how to structure programs in-the-large).
- Classes (a mechanism for providing *overloading* of operators such as `+`).
- Monadic programming (enforcing a particular sequence of operations in a purely functional language).
- Exceptions (execute user-defined functions if an error occurs in a program).

Summary of Concepts in Haskell

The most important language concepts of Haskell are:

- *purely functional*: no side-effects;
- *strong type system*: types can be checked at compile-time;
- *algebraic data structures* with powerful operations;
- *(parametric) polymorphism*: the same function can operate on values of different types;
- *overloading (ad-hoc polymorphism)*: different functions can be represented by the same symbol;
- *higher-order functions*: functions can be passed to other functions or returned as values;
- *lazy evaluation*: functions may return a result even if some of its arguments aren't well-defined (*non-strict semantics*).