Advanced Functional Programming, 1DL450 2012

Lecture 1, 2012-10-29

Cons T Åhs

Cons T Åhs

- Visiting teacher, i.e., really works somewhere else
- Former lecturer at IT Department (basic programming, algorithms, data structures, compiler course, logic programming, advanced programming, semantics, theory of programs etc)
- Some 10 years as a consultant (music notation, medical imaging, low level network programming, financial systems, teaching, real time video decoding, online poker, speech synthesis, ...)
- Currently at Klarna; financial company doing most development in Erlang
  - responsible for code quality, increasing competence of developers, software design and code consideration
  - mostly tool development for code analysis and metrics

Practical Issues

- Best method of contact is mail: cons.t.ahs@it.uu.se
- In person meetings best in conjunction with lectures, but set up a time over mail
- Don't expect to find me during office hours at the department
- Course assistant: Stavros Aronis (stavros.aronis@it.uu.se)
  - First contact for help
  - Marks assignments
- Please don't ask me about issues regarding course registration etc.
  - I know nothing about this
  - Talk to a student advisor or similar

Course Overview

- Advanced functional programming or Advanced programming in “functional” languages.
- Introduction to and use of three functional languages
  - Lisp - The Original, around for a long time, beautiful.
  - Haskell - The Pure Way of functional programming.
  - Erlang - Pragmatic, industrial strength language that happened to be a functional language. Ugly.
- Each language has its quirks and features, owing to history and design decisions
  - Not all quirks and features are *functional* in nature
  - These quirks and features provide added strength to the language and might be the main reason for using them

Course Overview

- 10 lectures
  - 14 in schedule - some will be canceled
  - three lectures/language
    - no in depth coverage of any language
    - learn by doing exercises and assignments
- Three assignments
  - One for each language, to be be handed in and marked by assistant
    - Will typically cover basics and some advanced feature of the language
- One larger project
  - Choose one of the three languages
  - Marked by assistant
  - Presented at end of course
- Written exam

Course Overview

- The pace will be rather high and the amount of material covered rather large
- With three different languages with their own syntax, semantic, quirks and features, there is a risk of confusing material from different lectures
- Best way to learn is to try out the concepts after each lecture
  - I will try to provide a set of exercises for each lecture
  - Do the exercises and discuss with the assistant or me
  - Exercises do not need to be submitted
  - Compare with known functional languages
- There lectures will talk about concepts and constructs in each language, but not the actual use.
  - There is some info on the course home page for this.
  - The assistant can help as well.

Preliminary Lecture Plan

1. Introduction and overview, basics, introduction to Erlang
2. Concurrent programming in Erlang
3. Advanced concepts in Erlang
4. Introduction to Common Lisp
5. Common Lisp Macros
6. Advanced concepts in Common Lisp, e.g., CLOS
7. Introduction to Haskell
8. More Haskell
9. Monads in Haskell
10. Functional Programming in the Real World, Leftovers and Summary

Prerequisites

- Basic functional programming
- Recursion
- Recursive data structures, such as lists and trees,
  - standard functions, e.g., `append`, `member`, `last`, `reverse`, `map`, `filter`, `fold`
  - The ability to define the above functions
- Simple higher order programming
- Closures
- Basic complexity issues, e.g., the complexity of the above functions
- The concept of tail recursion and the benefits of it
- Basic understanding of type inference
- Curried functions

The Common Base

- Lambda Calculus; Alonzo Church (1903-1995)
- Distill the concept of a function as a mapping from one set to another
- Separate naming/definition from the actual function
  - f(x) = g(x) + 3
  - f = λx.g(x) + 3
- A function is a value in itself and can be
  - passed as an argument to a function
  - returned as the result of a computation
- Higher order functions!
  - λfλx.f(x) = 0 -> g(x); h(x)
  - Partial application returns a closure
    - (λxλy.x+y)3 => λy.x+y (with x bound to 3)
- No side effects, i.e., names can be bound, but the value not changed
  - Single assignment variables
- In theory, this is all you need for computation
- All useful language has to be practical and deal with the real world.

Erlang Overview

- Background from Ericsson
  - a next generation language for software for telecom switches
- Requirements/background - a 24/7 business with low tolerance for downtime
  - high availability
  - robust and fault tolerant - a failing program should not take down the whole system or a node
  - distributed and concurrent; expand to more traffic by adding more processor nodes
    - easy to upgrade without downtime; new software or hardware
- Soft real time
- First implementations written as interpreters in Prolog
  - Syntax shared with Prolog
- Compiles to virtual machine (beam) or to native code
- Just happened to be functional
  - Some functional constructs bolted on afterwards
- Very much an evolved language rather than a designed language

Factorial

```
-module(fact).

-export([factorial/1]).

factorial(N) when N > 0 ->
  N*factorial(N-1);
factorial(0) -> 1.
```

```
-module(fact).

-export([factorial/1]).

factorial(N) -> factorial(N, 1).

factorial(N, F) when N > 0 ->
  factorial(N-1, F*N);
factorial(0, F) -> F.
```

- Stored in file `fact.erl` - an erlang module corresponds to a single file
- Only exported functions (`factorial/1`) are available externally
- Clauses are tested in order
- Clauses are separated with a semicolon
- Last clause ends with a period
- Variables are starts with a uppercase character
- The expression after when is called a *guard* - limited set of operators allowed, not any function call
- Why is the version on the right better?

Append lists

```
-module(append).

-export([append/2]).

append([], L) -> L;
append([X|Xs], L) -> [X|append(Xs, L)].
```

- List syntax
  - `[]` for empty list
  - `[Head | Tail]` pattern for head and tail of list
  - `[1, 2, 3]` list of three element
- Pattern matching can be used in clauses
  - Runtime error if there is no clause matching the call
- What's the complexity of this function?
- The builtin ++ operator does the same thing, so L1 ++ L2 appends the lists.

Usage

```
2> fact:factorial(10).
3628800
3> fact:factorial(100).
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
7> append:append([1,2,3],[a,b,c]).
[1,2,3,a,b,c]
8>
```

- Function call uses both module and function name
- Erlang has bignums, i.e., arbitrarily large integers
- Lists with mixed types are allowed
- Erlang is not a typed language
  - Type errors not caught at compile time

Quirk: No Strings(!)

```
9> append:append("no ", "strings").
"no strings"
10> [97, 98, 99].
"abc"
11>
```

- The normal string notation is just syntactic sugar for a list of character codes
- Lists of integers that (seem to) represent characters are printed as strings
- All list operations can be used on strings

Tuples

```
-module(tuples).

-export([build/2, first/1, second/1]).

build(X, Y) -> {X, Y}.

first({X, _}) -> X.

second({_, Y}) -> Y.
```

- You can group N (N≥0) things in a tuple
- Pattern matching can be done on tuples as well as lists
- _ is an anonymous variable, i.e., a placeholder for an ignored value

Conditional computation

- Pattern matching together with clauses is one way of doing conditional computation.
- The traditional way in a functional language is to supply a built in construct
  - C -> E1; E2
  - C is an arbitrary expression that which evaluates to true or false
  - If C evaluates to true, E1 is evaluated and returned
  - If C evaluates to false, E2 is evaluated and returned
- Why is this described with the term “construct” instead of “function”?
- Some languages got this extremely right from the start, Erlang did not..

Conditional - case

```
case lists:member(3, L) of
  true -> ... ;
  false -> ...
end

case foo(X, Y, Z) of
  ok -> ... ;
  [] -> ... ;
  {U, V} -> ...U..V
end
```

- Evaluate expression and match different results
- Cases are separated with semicolon
- Last case clause does not end with semicolon (or period)
- An `end` marks the end of the case clauses
- The result of the expression can be any type, which is reflected in the case clauses
- Variables can be bound in the patterns

Conditional - if

```
if
  integer(X) -> ... ;
  tuple(X) -> ... ;
  N > 0 -> ...
  true -> ...
end
```

- This is **not** a traditional function if!
- This is **not** similar to `COND` in Lisp!
- One can not write arbitrary expressions in the conditional, only *guards*
- Erlang's `if` is generally considered to be broken and you'll actually very seldom see it used in real programs.
- `case` and/or pattern matching is used instead
- A guard is an expression consisting only of operators and built in functions
  - A construct to make computation efficient

Terms

- Terms in Erlang are built from the bottom up using
  - atoms
    - starts with a lower case character
    - can be arbitrarily long
    - can be compared for identity in constant time - how?
    - useful for tags in tuples - simple runtime data typing
    - should not be used as strings - they are not garbage collected
  - numbers (ints and floats)
  - lists
  - tuples

Records

```
-record(person, {name, age=0, length}).

mk_person(Name) -> #person{name=Name}.

mk_person(Name, Age) ->
  #person{name=Name, age=Age}.

get_name(#person{name=Name}) -> Name.

get_age(Person) -> Person#person.age.

change_age(Person, Age) ->
  Person#person{age=Age}.
```

- Syntactic sugar for tuples with first component being the name of the record
- A somewhat abstract representation - changes in representation can be hidden
- Record syntax can be used in pattern matching
- Records were added to the language as an afterthought

Binding and matching

```
foo1(N) ->
  X = N,
  X = 1.

foo2(N) ->
  X = N,
  X == 1.
```

- Variables are single assignment, so the first occurrence of a variable will bind it
- In subsequent occurrences, the bound value is used and can not be changed
- The = operator does bind **and** matching (of patterns) and can fail, i.e., generate a runtime error
- The == operator does **only** matching (no binding) and returns true or false.
- What is the difference between `foo1/1` and `foo2/1`?

Local variables, scope

```
f(X, Y) ->
    A = X+Y,
    B = X-Y,
    {A, B}.

g(T) ->
    M = case T of
        {N} -> true;
        {N, _} -> false
    end,
    {M, N}.
```

- The scope of a local variable binding is the rest of the clause
- This is true even if a variable is introduced by pattern matching in a case clause